Breaking a Multi-Layer Crypter through Reverse-Engineering, A case study into the MAN1 Crypter

Jason Reaves

February 21, 2016

Abstract

Crypters and packers are common in the malware world, lots of techniques have been invented over the years to help people bypass security measures commonly used. One such technique where a crypter will use multiple, sometimes dynamically generated, layers to decode and unpack the protected executable allows a crypter to bypass common security measures such as Antivirus. While at the end of this paper we will have constructed a working proof of concept for an unpacker it is by no means meant as a production level mechanism, the goal is simply to show the reversing of routines found in a crypter while using a reverse-engineering framework that is geared towards shellcode analysis to our benefit for malware analysis.

1 Introduction

A Multi-Layer crypter is simply a crypter or packer that uses multiple stages of executable layers to achieve it's purpose of loading a final protected executable. Reversing these things is usually considered pointless by researchers unless they're employing some sort of a new anti-analysis trick or exploit for privilege escalation as normally the interesting thing to look at will be the unpacked data.
This crypter which will be nicknamed MAN1 for the purpose of this paper is one that I have followed for a long time. It's used by a particular group/actor and has been used by them as they moved around large malware families. The group/actor in question has been involved in Vawtrak, Dyre, Nymaim, ZeusVM and ReactorBot to name a few.

2 First and Second Layers

The first layer isn't really hidden as well as the others, it's basically a block of code that is XORd. This section however contains most of the more important routines that will be used during the unpacking process. This layer will decode and pass execution to the second layer, but both of these layers use the same routines so we will only go through the routines in a general overview manner since our primary concern is pulling out the unpacked payload.

The main purpose of these layers is to reconstruct the next layer and then call it. By 'reconstruct' I'm referring to a process by which the data that has been broken into chunks, encoded and sometimes compressed with a header of encoded values on the top of the data will be decoded and put back together. This routine will find these chunks and piece them back together, I refer to these chunks as a set for the purpose of this paper because there is an identifying number assigned to each set of chunks with each chunk having an index number into the set.

2.1 Main Loop

Each layer begins in roughly the same fashion with a loop that will enumerate through each index of the set that corresponds to the next layer of data it is to unpack(Figure 1). The biggest difference with the last layer being that it has code to check and load the payload after it has been unpacked as opposed to just jumping to the next layer.

2.2 Finding a Set Chunk

The first notable routine is used very frequently used, it's to find the first occurrence in a given data blob for a header with the proper set number. Some psuedo C code for this can be see in Figure 2. After all the checks pass an int value is passed off to a function for decoding and compared against the set number that was passed in.

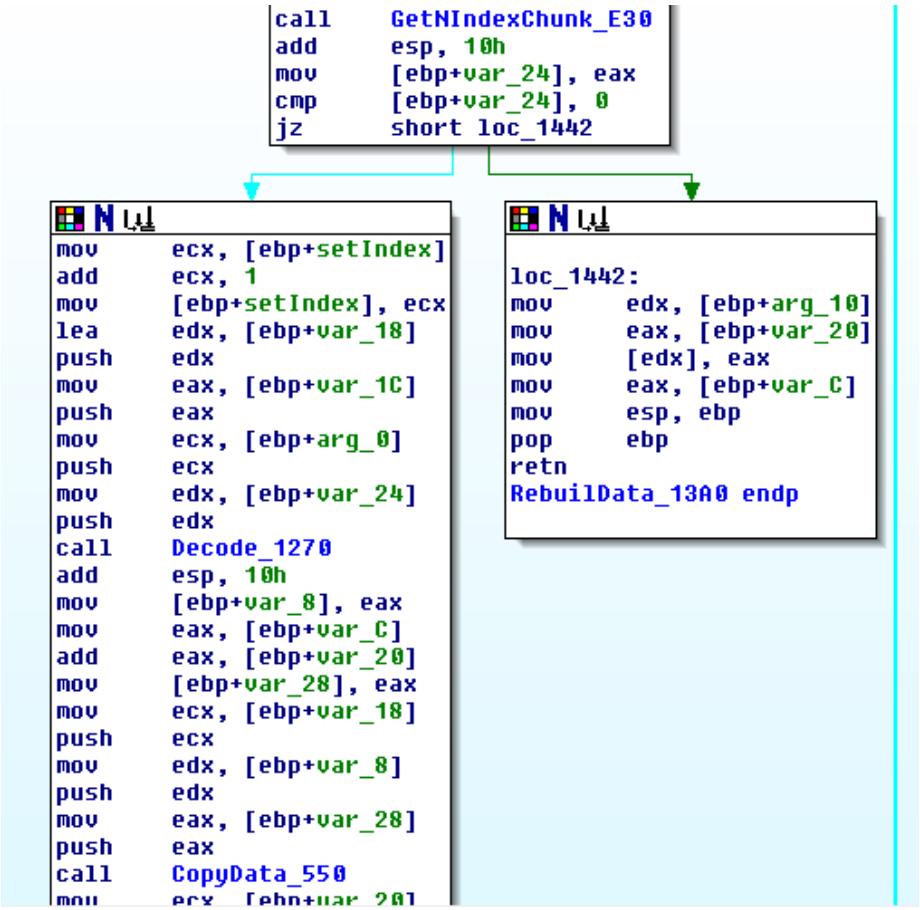

Figure 1: Main Loop from Crypter Layer

2.3 Decoding the Header

The function mentioned for decoding a header value can be seen as psuedo C code in Figure 3. There's some obvious areas where this function could throw a division by 0 error which could be the reason for the previous check values and code. As can be seen from the main loop of the program this routine is used to decode all the values of the header and using this plus following out the values as they're parsed we can start to put together a structure for the header(Figure 4).

2.4 Decoding the Data

After a piece of the next section is found in the set it will be decoded. The part of the code that does the decoding is just another smaller set to be found that is

```
int FindSetChunk(char *data, int setNum)
{
  int i = 0;
  while(i < strlen(data)-1)
  {
    int increment = 1;
    unsigned short CheckVal1 = *((unsigned short*)data);
    if(CheckVal1 > 1500 && CheckVal1 < 4000)
    {
      unsigned short CheckVal2 = *((unsigned short*)(data+2));
      if(CheckVal2 >= CheckVal1+700 &&
         CheckVal2 <= CheckVal1+CheckVal1+1967)
      {
        unsigned short offset = *((unsigned short*)(data+4));
        if(i + offset + 6 < strlen(data) &&
          *((unsigned short*)(data+offset+6)) ==
                        offset + CheckVal1 + 2*CheckVal2)
        {
          increment += offset+8;
          if(DecodeHdrVal(*((unsigned int*)(data+6))) == setNum)
            return i+6;
        }
      }
    }
    i += increment;
    data += increment;
  }
  return 0;
}
```

Figure 2: Find Set Chunk

the byte code of a function. The interesting part is that this byte code appears to change between samples which means it's probably dynamically generated as part of the stub which will then probably encode the layers and payload as part of the build process. Using Miasm[3] which is a reverse-engineering python framework focused on shellcode analysis we can disassemble this byte code after finding it(this code comes from a Pony sample) to see the relevant decoding section of this code in Figure 5.

3 Putting it all together

As I mentioned at the beginning the layers all use the same methods for retrieving data so in order to automate pulling out the final unpacked payload we only

```
int DecodeHdrVal(int val)
{
  int result = 0;
  int temp = val;
  if(LOWORD(val) > 3200)
    temp = val - 3200;

  int rem1 = LOWORD(temp) % 100;
  int rem2 = LOWORD(temp) % 100;
  if(rem1 > 50)
    rem2 = rem2 >> 1;

  result = LOWORD(temp - rem1 - HIWORD(val) / rem2) / rem2;
  result += (LOWORD(temp - rem1 - HIWORD(val) / rem2) % rem2) << 16;

  return result;
}
```

Figure 3: Decode Header Values

```
struct DataBlob {
  unsigned short CheckVal1;
  unsigned short CheckVal2;
  unsigned short CheckOffset;
  struct ChunkHeader {
    unsigned int SetNum;
    unsigned int length;
    unsigned int SetIndex;
    unsigned int check;
    unsigned int key;
    unsigned int compressedflag;
    unsigned int uncompressedSize;
  }chunk;
  char data[chunk.length];
}
```

Figure 4: Header Structure

need the routines that we have mentioned thus far and the set number of the payload data which is 0 for the samples I went through.
Finding the data and pulling out the relevant header values we will need isn't a difficult process(Figure 6). However, how do we go about accounting for the embedded byte code function that changes between samples? There are many

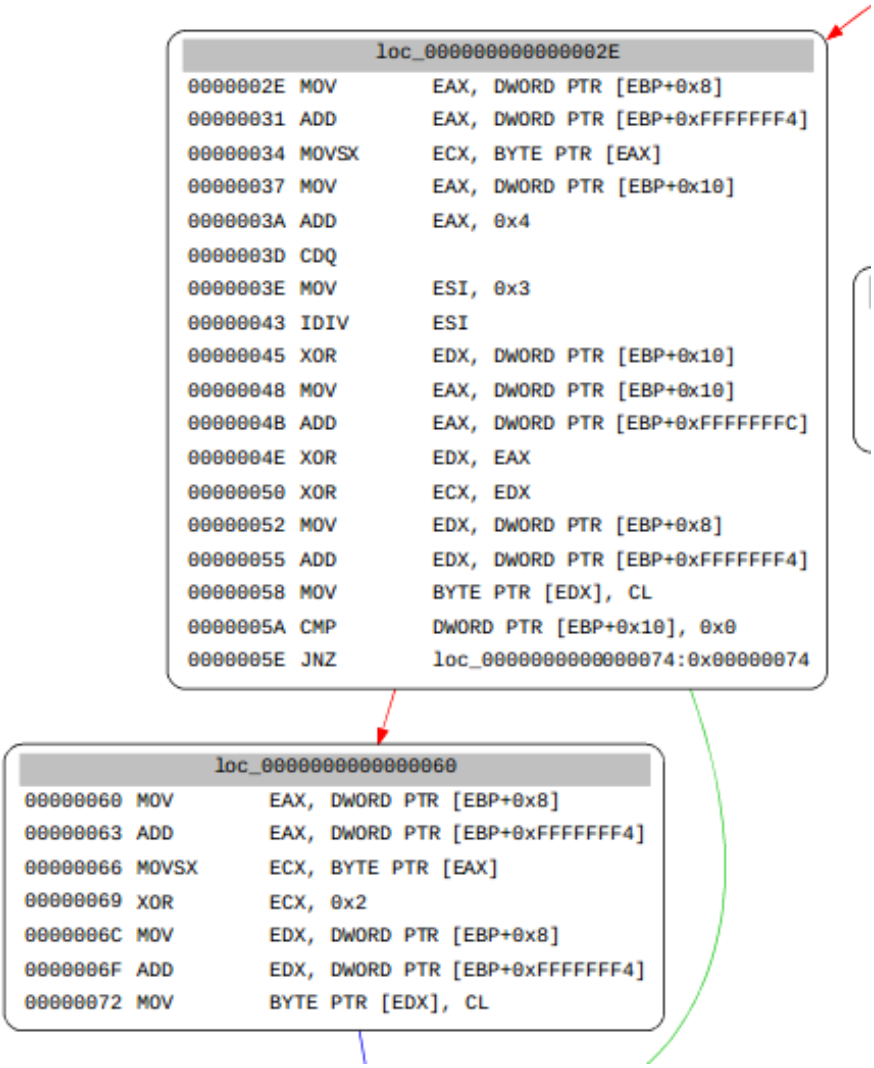

Figure 5: Disassembled Byte Code from Miasm

ways to go about solving the problem by using other executables but I decided to keep it in python as much as I could so I went back to Miasm because I had read an article[4] where someone had walked through shellcode in memory with it through dynamic analysis. Miasm comes with an example script that performs pretty much what we're looking for(`example/jitter/x86_32.py`) and studying how they capture the end by pushing a bogus address on the stack we can figure out how to call our byte code with parameters(Figure 7)!
Armed with all of this constructing an unpacker is very possible, I've put one

```
i = 1
t = 1
ret = ""
while t != 0:
        t = main_loop(0, i, data, False)
        if t != 0:
                temp = scode(str(t[2]), len(t[2]), t[0], binascii.unhexlify(shellcode))
                if t[3] == 1 or t[3] == 2:
                        temp = LZNT_decompress(temp, 0)
                ret += str(temp[:t[1]])
                if t[1] > len(temp):
                        ret += '\x00' * (t[1] - len(temp))

        i += 1
```

Figure 6: Loop Through Payload Set Overview

```
myjit = Machine("x86_32").jitter("tcc")
myjit.init_stack()

run_addr = 0x40000000
myjit.vm.add_memory_page(run_addr, PAGE_READ | PAGE_WRITE, shellcode)

#myjit.jit.log_regs = True
#myjit.jit.log_mn = True
#myjit.jit.log_newbloc = True

myjit.add_breakpoint(0x1337beef, code_sentinelle)
myjit.vm.add_memory_page(0x10000000, PAGE_READ | PAGE_WRITE, data)
myjit.push_uint32_t(key)
myjit.push_uint32_t(len(data))
myjit.push_uint32_t(0x10000000)
myjit.push_uint32_t(0x1337beef)
myjit.init_run(run_addr)
myjit.continue_run()
return myjit.cpu.get_mem(0x10000000,len(data))
```

Figure 7: Decoding with Miasm

on github that follows the flow of the packer pretty closely and left some of my comments in it from when I was reversing the flow of the routines (`https://github.com/sysopfb/Unpackers/blob/master/Man1/man1unpack.py`.) As I mentioned this is not production code as it is a highly unoptimized approach using python, making it faster is entirely possible in many different ways.

4 Conclusions

Pony Sample SHA256:
f51d9f113a2935fe4663ef4f4f8db2c26c7426e3859ed94afdd3115e69b4091d
Nymaim Samples SHA256:
214349a9c898377a0dc9680cbe2ca13f7bcc0cd59d551769a5461514db7520e0

fd759e89749603dbbce2ae2dafaa8e89238c174ffa392fd3af78809a965cf692
Vawtrak Sample SHA256:
616db219c96f8f39709e3356ec242f8dbdbe181fe92022c015b2fb270c6ee18c

References

[1] IDA, https://www.hex-rays.com/products/ida/

[2] Python, https://www.python.org/

[3] Miasm - Reverse engineering framework in Python, https://github.com/cea-sec/miasm

[4] Dynamic shellcode analysis - Miasm's blog, *http* : *//www.miasm.re/blog/*2016/02/12/*dynamic*$_s$*hellcode*$_a$*nalysis.html*