A Case Study into solving Crypters/Packers in Malware Obfuscation using an SMT approach

Jason Reaves

ABSTRACT

Obfuscation in malware is commonly employed for any number of reasons but it's purpose is ultimately the same, to make the underlying malicious entity go unnoticed. Crypters and Packers both are heavily employed to bypass common security measures so ultimately these are just tools. Tools that are utilizing algorithms in order to take data and turn it into some other data while being able to reverse the process later, obviously these reversible algorithms can be chained together as well into 'layers'. In this paper I explore the idea that it is easier to think of these layers as a math equation which can be solved. This has the potential of turning something that can be overwhelming at first, like writing an unpacker, into a much more manageable problem.

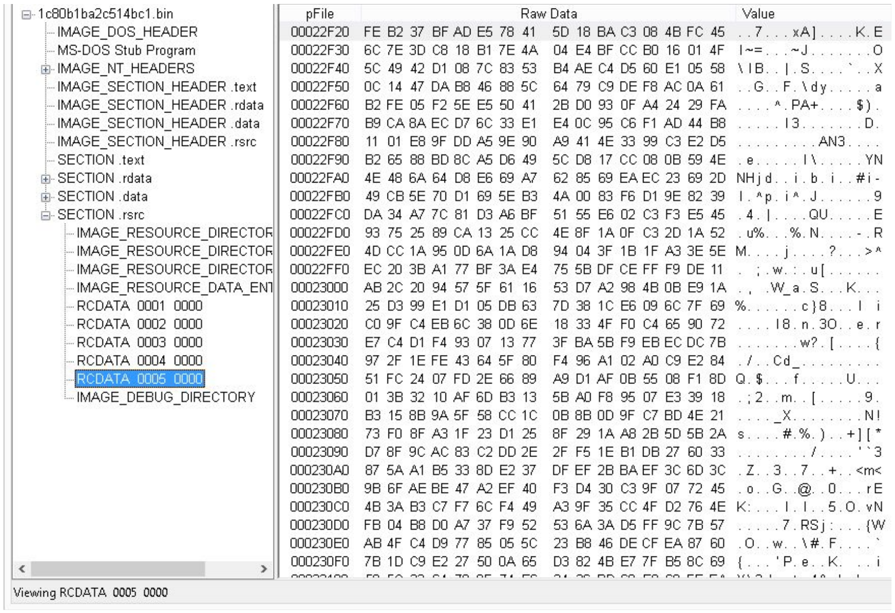
For the purpose of this paper I will refer to packers[9] and crypters[9] both as packers, the reason being that in the world of malware both are used for obfuscating the underlying code that is to be executed.

1. Introduction

Packers have evolved greatly over the years, especially with malware needing to utilize crypters and packers that can bypass any number of obstacles depending on their targets. For brevity we will focus specifically on crypters that utilize multiple binary operators to obfuscate their payloads, it seems a natural progression that researchers will usually move towards finding ways to pivot from other data in these scenarios such as finding ways to rip out starting values through various techniques including bruting, select-bruting, regex matching, nearby static data pivoting or any number of other process for basically finding values. Instead of finding values I always yearned to be able to instead lean on math in regards to solving a problem, if I can reverse this routine and describe it in an adequate manner to write it in a higher level language then I should be able to describe this routine as a problem that can either be simplified or best case solved. This line of thought is what eventually led me to find Z3[6] and its usefulness in subsets of malware research.

2. Finding the problem

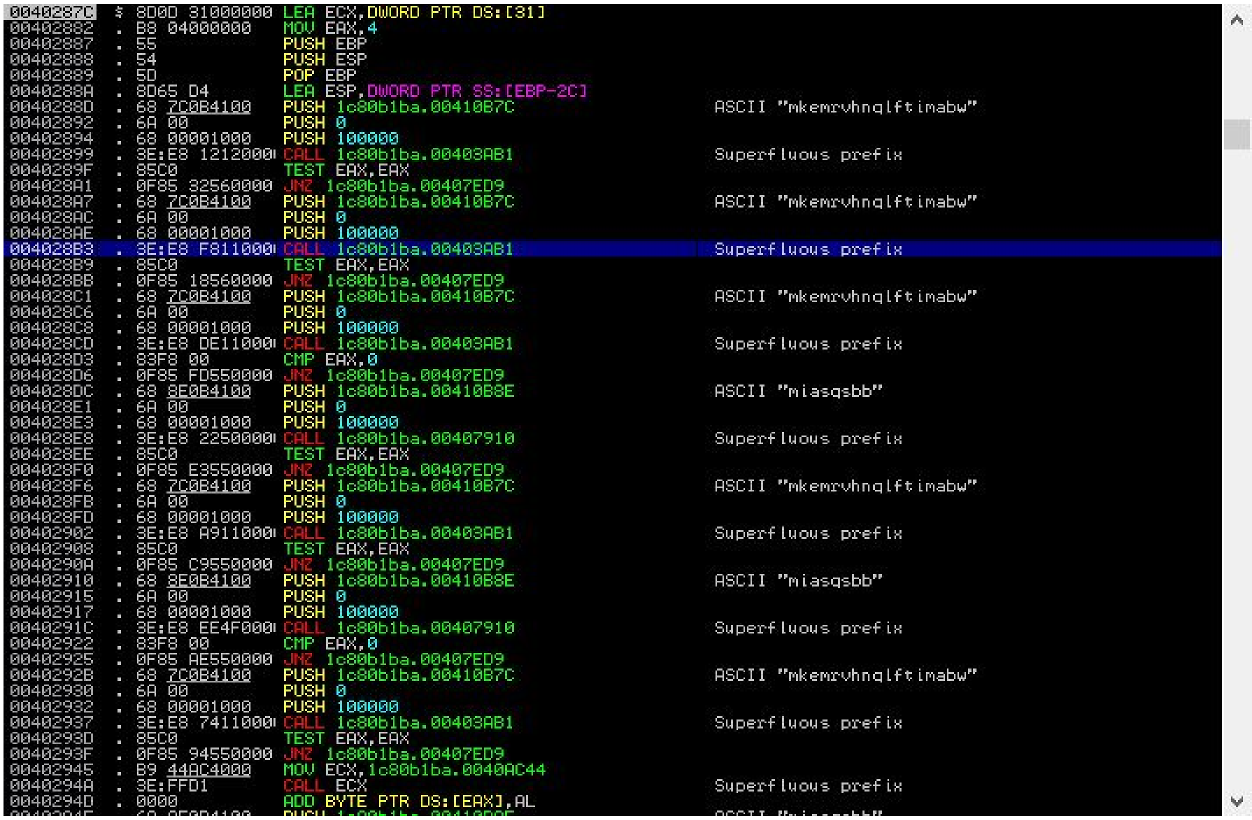
The sample we'll be looking at is specifically the crypter being used by the latest Locky Ransomware campaigns in late August 2017, 1c80b1ba2c514bc1d32eb5b9909d79812ab8f2944548bc96757c1d992ce6d8ac. While the object of this paper is not to show how to reverse engineer routines or malware, we will simply walk through to the relevant portion of code in order to begin describing our problem. Basically we're going to find where the routine that is responsible for decoding the payload. For this crypter a quick glance at the PE file shows a potentially encoded resource section.



```
pFile     Raw Data                                           Value
00022F20  FE B2 37 BF AD E5 78 41  5D 18 BA C3 08 4B FC 45  ..7...xA]....K.E
00022F30  6C 7E 3D C8 18 B1 7E 4A  04 E4 BF CC B0 16 01 4F  l~=...~J.......O
00022F40  5C 49 42 D1 08 7C 83 53  B4 AE C4 D5 60 E1 05 58  \IB..|.S....`..X
00022F50  0C 14 47 DA B8 46 88 5C  64 79 C9 DE F8 AC 0A 61  ..G..F.\dy.....a
00022F60  B2 FE 05 F2 5E E5 50 41  2B D0 93 0F A4 24 29 FA  ....^.PA+....$).
00022F70  B9 CA 8A EC D7 6C 33 E1  E4 0C 95 C6 F1 AD 44 B8  .....l3.......D.
00022F80  11 01 E8 9F DD A5 9E 90  A9 41 4E 33 99 C3 E2 D5  .........AN3....
00022F90  B2 65 88 BD 8C A5 D6 49  5C D8 17 CC 08 0B 59 4E  .e.....I\.....YN
00022FA0  4E 48 6A 64 D8 E6 69 A7  62 85 69 EA EC 23 69 2D  NHjd..i.b.i..#i-
00022FB0  49 CB 5E 70 D1 69 5E B3  4A 00 83 F6 D1 9E 82 39  I.^p.i^.J......9
00022FC0  DA 34 A7 7C 81 D3 A6 BF  51 55 E6 02 C3 F3 E5 45  .4.|....QU.....E
00022FD0  93 75 25 89 CA 13 25 CC  4E 8F 1A 0F C3 2D 1A 52  .u%...%.N....-.R
00022FE0  4D CC 1A 95 0D 6A 1A D8  94 04 3F 1B 1F A3 3E 5E  M....j....?...>^
00022FF0  EC 20 3B A1 77 BF 3A E4  75 5B DF CE FF F9 DE 11  . ;.w.:.u[......
00023000  AB 2C 20 94 57 5F 61 16  53 D7 A2 98 4B 0B E9 1A  ., .W_a.S...K...
00023010  25 D3 99 E1 D1 05 DB 63  7D 38 1C E6 09 6C 7F 69  %......c}8...l.i
00023020  C0 9F C4 EB 6C 38 0D 6E  18 33 4F F0 C4 65 90 72  ....l8.n.3O..e.r
00023030  E7 C4 D1 F4 93 07 13 77  3F BA 5B F9 EB EC DC 7B  .......w?.[....{
00023040  97 2F 1E FE 43 64 5F 80  F4 96 A1 02 A0 C9 E2 84  ./..Cd_.........
00023050  51 FC 24 07 FD 2E 66 89  A9 D1 AF 0B 55 08 F1 8D  Q.$...f.....U...
00023060  01 3B 32 10 AF 6D B3 13  5B A0 F8 95 07 E3 39 18  .;2..m..[.....9.
00023070  B3 15 8B 9A 5F 58 CC 1C  0B 8B 0D 9F C7 BD 4E 21  ...._X........N!
00023080  73 F0 8F A3 1F 23 D1 25  8F 29 1A A8 2B 5D 5B 2A  s....#.%.)..+][*
00023090  D7 8F 9C AC 83 C2 DD 2E  2F F5 1E B1 DB 27 60 33  ......../....'`3
000230A0  87 5A A1 B5 33 8D E2 37  DF EF 2B BA EF 3C 6D 3C  .Z..3..7..+..<m<
000230B0  9B 6F AE BE 47 A2 EF 40  F3 D4 30 C3 9F 07 72 45  .o..G..@..0...rE
000230C0  4B 3A B3 C7 F7 6C F4 49  A3 9F 35 CC 4F D2 76 4E  K:...l.I..5.O.vN
000230D0  FB 04 B8 D0 A7 37 F9 52  53 6A 3A D5 FF 9C 7B 57  .....7.RSj:...{W
000230E0  AB 4F C4 D9 77 85 05 5C  23 B8 46 DE CF EA 87 60  .O..w..\#.F....`
000230F0  7B 1D C9 E2 27 50 0A 65  D3 82 4B E7 7F B5 8C 69  {...'P.e..K....i
```

Figure 1 Resource sections

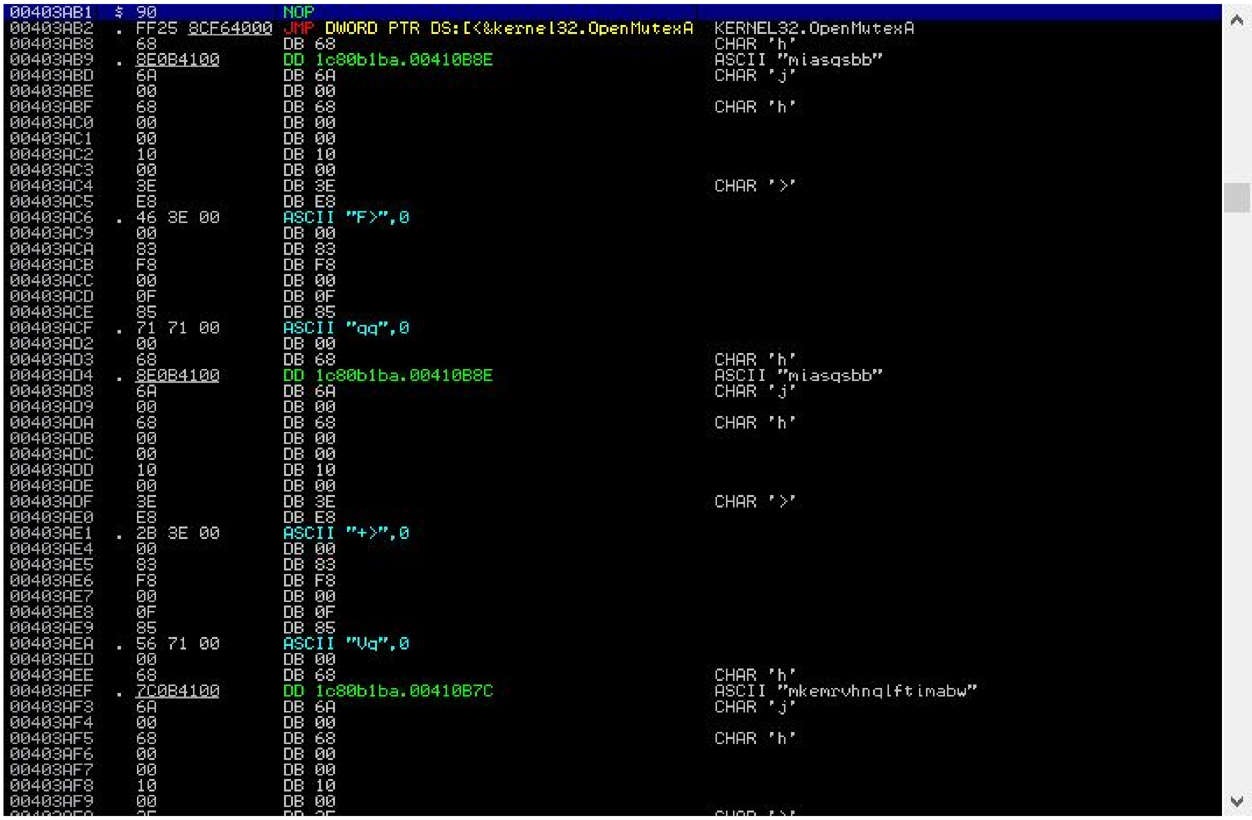
Opening up the file in a debugger shows a bunch of very similar calls at the main entrypoint[10].



```
0040287C  $ 8D0D 31000000 LEA ECX,DWORD PTR DS:[31]
00402882  . B8 04000000   MOV EAX,4
00402887  . 55            PUSH EBP
00402888  . 54            PUSH ESP
00402889  . 5D            POP EBP
0040288A  . 8D65 D4       LEA ESP,DWORD PTR SS:[EBP-2C]
0040288D  . 68 7C0B4100   PUSH 1c80b1ba.00410B7C            ASCII "mkemrvhnqlftimabw"
00402892  . 6A 00         PUSH 0
00402894  . 68 00001000   PUSH 100000
00402899  . 3E:E8 1212000 CALL 1c80b1ba.00403AB1            Superfluous prefix
0040289F  . 85C0          TEST EAX,EAX
004028A1  . 0F85 32560000 JNZ 1c80b1ba.00407ED9
004028A7  . 68 7C0B4100   PUSH 1c80b1ba.00410B7C            ASCII "mkemrvhnqlftimabw"
004028AC  . 6A 00         PUSH 0
004028AE  . 68 00001000   PUSH 100000
004028B3  . 3E:E8 F811000 CALL 1c80b1ba.00403AB1            Superfluous prefix
004028B9  . 85C0          TEST EAX,EAX
004028BB  . 0F85 18560000 JNZ 1c80b1ba.00407ED9
004028C1  . 68 7C0B4100   PUSH 1c80b1ba.00410B7C            ASCII "mkemrvhnqlftimabw"
004028C6  . 6A 00         PUSH 0
004028C8  . 68 00001000   PUSH 100000
004028CD  . 3E:E8 DE11000 CALL 1c80b1ba.00403AB1            Superfluous prefix
004028D3  . 83F8 00       CMP EAX,0
004028D6  . 0F85 FD550000 JNZ 1c80b1ba.00407ED9
004028DC  . 68 8E0B4100   PUSH 1c80b1ba.00410B8E            ASCII "miasqsbb"
004028E1  . 6A 00         PUSH 0
004028E3  . 68 00001000   PUSH 100000
004028E8  . 3E:E8 2250000 CALL 1c80b1ba.00407910            Superfluous prefix
004028EE  . 85C0          TEST EAX,EAX
004028F0  . 0F85 E3550000 JNZ 1c80b1ba.00407ED9
004028F6  . 68 7C0B4100   PUSH 1c80b1ba.00410B7C            ASCII "mkemrvhnqlftimabw"
004028FB  . 6A 00         PUSH 0
004028FD  . 68 00001000   PUSH 100000
00402902  . 3E:E8 A911000 CALL 1c80b1ba.00403AB1            Superfluous prefix
00402908  . 85C0          TEST EAX,EAX
0040290A  . 0F85 C9550000 JNZ 1c80b1ba.00407ED9
00402910  . 68 8E0B4100   PUSH 1c80b1ba.00410B8E            ASCII "miasqsbb"
00402915  . 6A 00         PUSH 0
00402917  . 68 00001000   PUSH 100000
0040291C  . 3E:E8 EE4F000 CALL 1c80b1ba.00407910            Superfluous prefix
00402922  . 83F8 00       CMP EAX,0
00402925  . 0F85 AE550000 JNZ 1c80b1ba.00407ED9
0040292B  . 68 7C0B4100   PUSH 1c80b1ba.00410B7C            ASCII "mkemrvhnqlftimabw"
00402930  . 6A 00         PUSH 0
00402932  . 68 00001000   PUSH 100000
00402937  . 3E:E8 7411000 CALL 1c80b1ba.00403AB1            Superfluous prefix
0040293D  . 85C0          TEST EAX,EAX
0040293F  . 0F85 94550000 JNZ 1c80b1ba.00407ED9
00402945  . B9 44AC4000   MOV ECX,1c80b1ba.0040AC44
0040294A  . 3E:FFD1       CALL ECX                          Superfluous prefix
0040294D  . 0000          ADD BYTE PTR DS:[EAX],AL
```

Figure 2 Entry point code

Peeking inside one of these calls shows that they are just jump commands to OpenMutex. So it is trying to open a mutex with the desired access of SYNCHRONIZE[8].



```
00403AB1  $ 90           NOP
00403AB2  . FF25 8CF64000 JMP DWORD PTR DS:[<&kernel32.OpenMutexA  KERNEL32.OpenMutexA
00403AB8    68           DB 68                                     CHAR 'h'
00403AB9  . 8E0B4100     DD 1c80b1ba.00410B8E                      ASCII "miasqsbb"
00403ABD    6A           DB 6A                                     CHAR 'j'
00403ABE    00           DB 00
00403ABF    68           DB 68                                     CHAR 'h'
00403AC0    00           DB 00
00403AC1    00           DB 00
00403AC2    10           DB 10
00403AC3    00           DB 00
00403AC4    3E           DB 3E                                     CHAR '>'
00403AC5    E8           DB E8
00403AC6  . 46 3E 00     ASCII "F>",0
00403AC9    00           DB 00
00403ACA    83           DB 83
00403ACB    F8           DB F8
00403ACC    00           DB 00
00403ACD    0F           DB 0F
00403ACE    85           DB 85
00403ACF  . 71 71 00     ASCII "qq",0
00403AD2    00           DB 00
00403AD3    68           DB 68                                     CHAR 'h'
00403AD4  . 8E0B4100     DD 1c80b1ba.00410B8E                      ASCII "miasqsbb"
00403AD8    6A           DB 6A                                     CHAR 'j'
00403AD9    00           DB 00
00403ADA    68           DB 68                                     CHAR 'h'
00403ADB    00           DB 00
00403ADC    00           DB 00
00403ADD    10           DB 10
00403ADE    00           DB 00
00403ADF    3E           DB 3E                                     CHAR '>'
00403AE0    E8           DB E8
00403AE1  . 2B 3E 00     ASCII "+>",0
00403AE4    00           DB 00
00403AE5    83           DB 83
00403AE6    F8           DB F8
00403AE7    00           DB 00
00403AE8    0F           DB 0F
00403AE9    85           DB 85
00403AEA  . 56 71 00     ASCII "Vq",0
00403AED    00           DB 00
00403AEE    68           DB 68                                     CHAR 'h'
00403AEF  . 7C0B4100     DD 1c80b1ba.00410B7C                      ASCII "mkemrvhnqlftimabw"
00403AF3    6A           DB 6A                                     CHAR 'j'
00403AF4    00           DB 00
00403AF5    68           DB 68                                     CHAR 'h'
00403AF6    00           DB 00
00403AF7    00           DB 00
00403AF8    10           DB 10
00403AF9    00           DB 00
```

Figure 3 Jump command

As long as all the calls return 0 then the code will come to a different function call that takes us to a different section of the binary that starts to make some LoadLibrary calls.

```
. B9 44AC4000   MOV ECX,1c80b1ba.0040AC44
. 3E:FFD1       CALL ECX
```

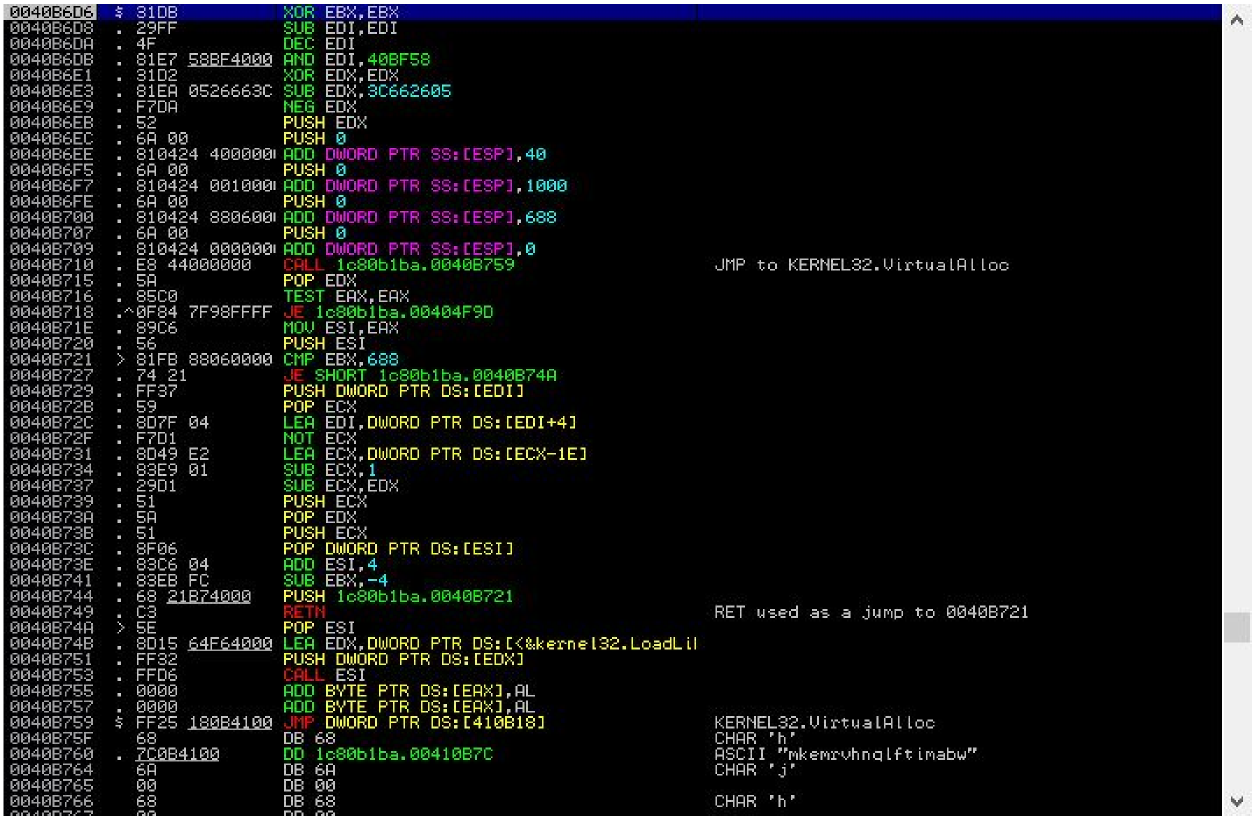
Figure 4 Execute next code block

A quick stop at a loop that calls WaitForSingleObject over and over, potentially a custom sleep routine. Sleep routines are commonly leveraged in malware to defeat sandbox analysis which will normally only execute a piece of malware for a set time amount[11].

```
57A2FI JE 1c80b1ba.00404F9D                          Superfluous prefix
       PUSH 1E
000    PUSH 3E8
       PUSH -1
54000  LEA EAX,DWORD PTR DS:[<&kernel32.WaitFo:
       CALL DWORD PTR DS:[EAX]                       KERNEL32.WaitForSingleObject
       DEC DWORD PTR SS:[ESP]
       JNZ SHORT 1c80b1ba.0040AD48                   Superfluous prefix
```

Figure 5 Custom sleep routine

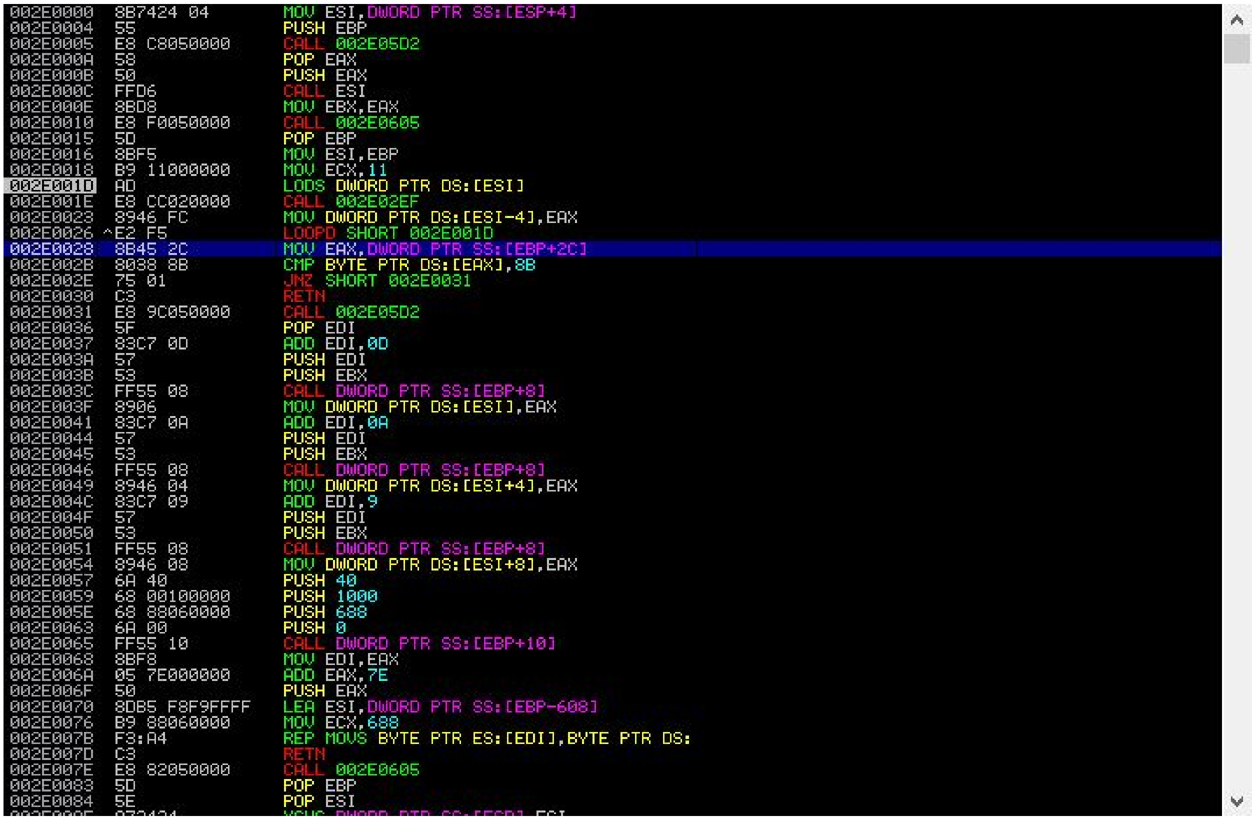
Moving on a little later we see a call to VirtualAlloc followed by a loop utilizing a push->ret technique. Unfortunately this isn't our routine for decoding the payload but instead the routine for decoding the bytecode layer that will be called next[12], do we need this layer? Possibly, whether or not we need to decode out that layer will depend on how the final routine is implemented for decoding out the payload. If you'd like an example of a slightly more advanced example of a crypter where we end up having to decode out some of the layers of a crypter I have a write up on one such crypter[1] where the decoding routine is dynamically generated and needs to be decoded.



```
0040B6D6  $ 31DB          XOR EBX,EBX
0040B6D8  . 29FF          SUB EDI,EDI
0040B6DA  . 4F            DEC EDI
0040B6DB  . 81E7 58BF4000 AND EDI,40BF58
0040B6E1  . 31D2          XOR EDX,EDX
0040B6E3  . 81EA 0526663C SUB EDX,3C662605
0040B6E9  . F7DA          NEG EDX
0040B6EB  . 52            PUSH EDX
0040B6EC  . 6A 00         PUSH 0
0040B6EE  . 810424 400000 ADD DWORD PTR SS:[ESP],40
0040B6F5  . 6A 00         PUSH 0
0040B6F7  . 810424 001000 ADD DWORD PTR SS:[ESP],1000
0040B6FE  . 6A 00         PUSH 0
0040B700  . 810424 880600 ADD DWORD PTR SS:[ESP],688
0040B707  . 6A 00         PUSH 0
0040B709  . 810424 000000 ADD DWORD PTR SS:[ESP],0
0040B710  . E8 44000000   CALL 1c80b1ba.0040B759                     JMP to KERNEL32.VirtualAlloc
0040B715  . 5A            POP EDX
0040B716  . 85C0          TEST EAX,EAX
0040B718  .^0F84 7F98FFFF JE 1c80b1ba.00404F9D
0040B71E  . 89C6          MOV ESI,EAX
0040B720  . 56            PUSH ESI
0040B721  > 81FB 88060000 CMP EBX,688
0040B727  . 74 21         JE SHORT 1c80b1ba.0040B74A
0040B729  . FF37          PUSH DWORD PTR DS:[EDI]
0040B72B  . 59            POP ECX
0040B72C  . 8D7F 04       LEA EDI,DWORD PTR DS:[EDI+4]
0040B72F  . F7D1          NOT ECX
0040B731  . 8D49 E2       LEA ECX,DWORD PTR DS:[ECX-1E]
0040B734  . 83E9 01       SUB ECX,1
0040B737  . 29D1          SUB ECX,EDX
0040B739  . 51            PUSH ECX
0040B73A  . 5A            POP EDX
0040B73B  . 51            PUSH ECX
0040B73C  . 8F06          POP DWORD PTR DS:[ESI]
0040B73E  . 83C6 04       ADD ESI,4
0040B741  . 83EB FC       SUB EBX,-4
0040B744  . 68 21B74000   PUSH 1c80b1ba.0040B721
0040B749  . C3            RETN                                       RET used as a jump to 0040B721
0040B74A  > 5E            POP ESI
0040B74B  . 8D15 64F64000 LEA EDX,DWORD PTR DS:[<&kernel32.LoadLi
0040B751  . FF32          PUSH DWORD PTR DS:[EDX]
0040B753  . FFD6          CALL ESI
0040B755  . 0000          ADD BYTE PTR DS:[EAX],AL
0040B757  . 0000          ADD BYTE PTR DS:[EAX],AL
0040B759  $ FF25 180B4100 JMP DWORD PTR DS:[410B18]                  KERNEL32.VirtualAlloc
0040B75F    68            DB 68                                      CHAR 'h'
0040B760  . 7C0B4100      DD 1c80b1ba.00410B7C                       ASCII "mkemrvhnqlftimabw"
0040B764    6A            DB 6A                                      CHAR 'j'
0040B765    00            DB 00
0040B766    68            DB 68                                      CHAR 'h'
```

Figure 6 Decode and execute next layer

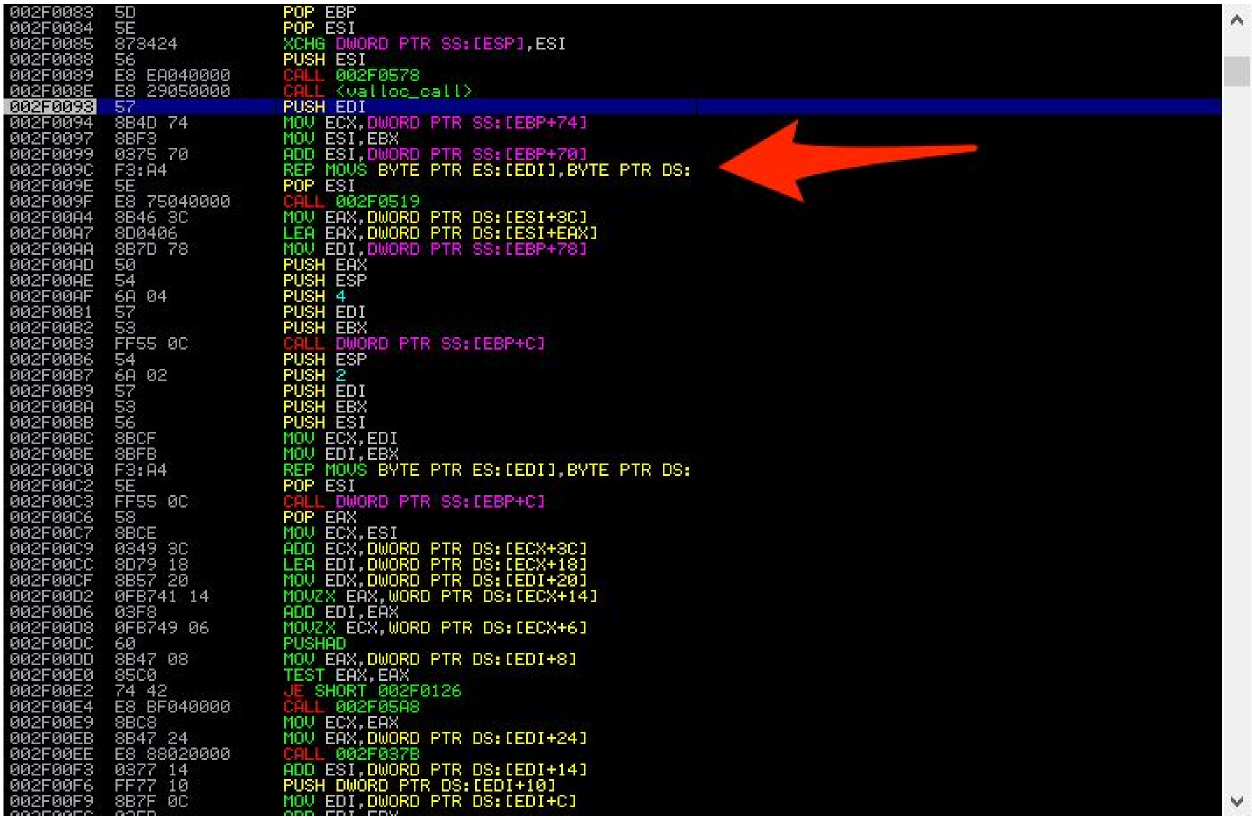
Heading into that next layer is just your normal code resolving any dependencies that it needs at runtime[13].



```
002E0000  8B7424 04       MOV ESI,DWORD PTR SS:[ESP+4]
002E0004  55              PUSH EBP
002E0005  E8 C8050000     CALL 002E05D2
002E000A  58              POP EAX
002E000B  50              PUSH EAX
002E000C  FFD6            CALL ESI
002E000E  8BD8            MOV EBX,EAX
002E0010  E8 F0050000     CALL 002E0605
002E0015  5D              POP EBP
002E0016  8BF5            MOV ESI,EBP
002E0018  B9 11000000     MOV ECX,11
002E001D  AD              LODS DWORD PTR DS:[ESI]
002E001E  E8 CC020000     CALL 002E02EF
002E0023  8946 FC         MOV DWORD PTR DS:[ESI-4],EAX
002E0026 ^E2 F5           LOOPD SHORT 002E001D
002E0028  8B45 2C         MOV EAX,DWORD PTR SS:[EBP+2C]
002E002B  8038 8B         CMP BYTE PTR DS:[EAX],8B
002E002E  75 01           JNZ SHORT 002E0031
002E0030  C3              RETN
002E0031  E8 9C050000     CALL 002E05D2
002E0036  5F              POP EDI
002E0037  83C7 0D         ADD EDI,0D
002E003A  57              PUSH EDI
002E003B  53              PUSH EBX
002E003C  FF55 08         CALL DWORD PTR SS:[EBP+8]
002E003F  8906            MOV DWORD PTR DS:[ESI],EAX
002E0041  83C7 0A         ADD EDI,0A
002E0044  57              PUSH EDI
002E0045  53              PUSH EBX
002E0046  FF55 08         CALL DWORD PTR SS:[EBP+8]
002E0049  8946 04         MOV DWORD PTR DS:[ESI+4],EAX
002E004C  83C7 09         ADD EDI,9
002E004F  57              PUSH EDI
002E0050  53              PUSH EBX
002E0051  FF55 08         CALL DWORD PTR SS:[EBP+8]
002E0054  8946 08         MOV DWORD PTR DS:[ESI+8],EAX
002E0057  6A 40           PUSH 40
002E0059  68 00100000     PUSH 1000
002E005E  68 88060000     PUSH 688
002E0063  6A 00           PUSH 0
002E0065  FF55 10         CALL DWORD PTR SS:[EBP+10]
002E0068  8BF8            MOV EDI,EAX
002E006A  05 7E000000     ADD EAX,7E
002E006F  50              PUSH EAX
002E0070  8DB5 F8F9FFFF   LEA ESI,DWORD PTR SS:[EBP-608]
002E0076  B9 88060000     MOV ECX,688
002E007B  F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:
002E007D  C3              RETN
002E007E  E8 82050000     CALL 002E0605
002E0083  5D              POP EBP
002E0084  5E              POP ESI
```

Figure 7 Resolve dependencies

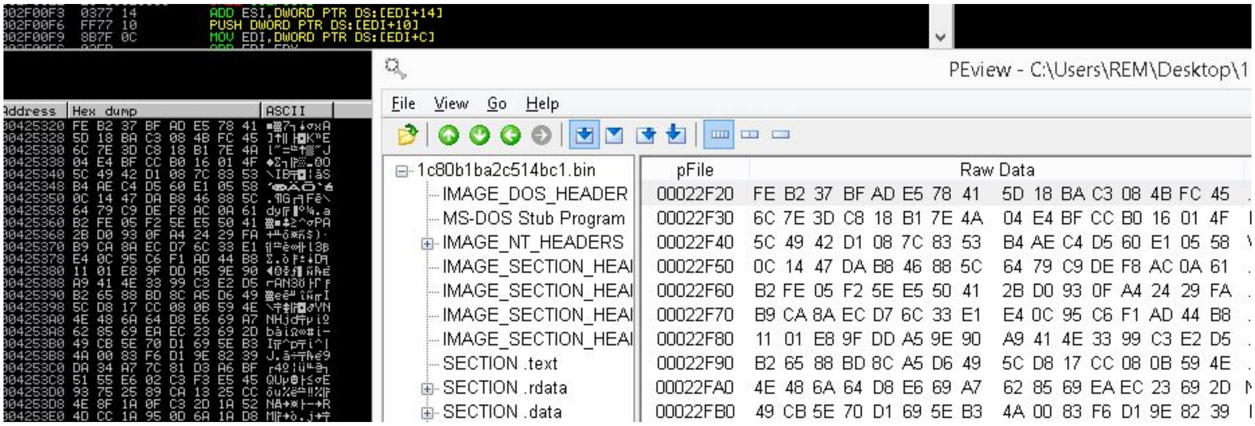
You might notice with this next picture the address change, simply because the bytecode layer fixes its own dependencies and then allocates a new memory section and copies itself over before calling the next section of code to be executed from within itself, kind of an odd way to do it but if you're the type that sets breakpoints everywhere you might find yourself with messed up code.
In this next code however we have a call to VirtualAlloc followed by some data being moved into our newly created memory.



```
002F0083  5D              POP EBP
002F0084  5E              POP ESI
002F0085  873424          XCHG DWORD PTR SS:[ESP],ESI
002F0088  56              PUSH ESI
002F0089  E8 EA040000     CALL 002F0578
002F008E  E8 29050000     CALL <valloc_call>
002F0093  57              PUSH EDI
002F0094  8B4D 74         MOV ECX,DWORD PTR SS:[EBP+74]
002F0097  8BF3            MOV ESI,EBX
002F0099  0375 70         ADD ESI,DWORD PTR SS:[EBP+70]
002F009C  F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:
002F009E  5E              POP ESI
002F009F  E8 75040000     CALL 002F0519
002F00A4  8B46 3C         MOV EAX,DWORD PTR DS:[ESI+3C]
002F00A7  8D0406          LEA EAX,DWORD PTR DS:[ESI+EAX]
002F00AA  8B7D 78         MOV EDI,DWORD PTR SS:[EBP+78]
002F00AD  50              PUSH EAX
002F00AE  54              PUSH ESP
002F00AF  6A 04           PUSH 4
002F00B1  57              PUSH EDI
002F00B2  53              PUSH EBX
002F00B3  FF55 0C         CALL DWORD PTR SS:[EBP+C]
002F00B6  54              PUSH ESP
002F00B7  6A 02           PUSH 2
002F00B9  57              PUSH EDI
002F00BA  53              PUSH EBX
002F00BB  56              PUSH ESI
002F00BC  8BCF            MOV ECX,EDI
002F00BE  8BFB            MOV EDI,EBX
002F00C0  F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:
002F00C2  5E              POP ESI
002F00C3  FF55 0C         CALL DWORD PTR SS:[EBP+C]
002F00C6  58              POP EAX
002F00C7  8BCE            MOV ECX,ESI
002F00C9  0349 3C         ADD ECX,DWORD PTR DS:[ECX+3C]
002F00CC  8D79 18         LEA EDI,DWORD PTR DS:[ECX+18]
002F00CF  8B57 20         MOV EDX,DWORD PTR DS:[EDI+20]
002F00D2  0FB741 14       MOVZX EAX,WORD PTR DS:[ECX+14]
002F00D6  03F8            ADD EDI,EAX
002F00D8  0FB749 06       MOVZX ECX,WORD PTR DS:[ECX+6]
002F00DC  60              PUSHAD
002F00DD  8B47 08         MOV EAX,DWORD PTR DS:[EDI+8]
002F00E0  85C0            TEST EAX,EAX
002F00E2  74 42           JE SHORT 002F0126
002F00E4  E8 BF040000     CALL 002F05A8
002F00E9  8BC8            MOV ECX,EAX
002F00EB  8B47 24         MOV EAX,DWORD PTR DS:[EDI+24]
002F00EE  E8 88020000     CALL 002F037B
002F00F3  0377 14         ADD ESI,DWORD PTR DS:[EDI+14]
002F00F6  FF77 10         PUSH DWORD PTR DS:[EDI+10]
002F00F9  8B7F 0C         MOV EDI,DWORD PTR DS:[EDI+C]
```

Figure 8 Next execution block to copy data over

Whenever I see something like this in a crypter the first thought that comes to my mind is "where is this data located in the binary". A quick check shows it's the resource section we had noticed when we were doing our precursory inspection.



```
002F00F3  0377 14         ADD ESI,DWORD PTR DS:[EDI+14]
002F00F6  FF77 10         PUSH DWORD PTR DS:[EDI+10]
002F00F9  8B7F 0C         MOV EDI,DWORD PTR DS:[EDI+C]
```

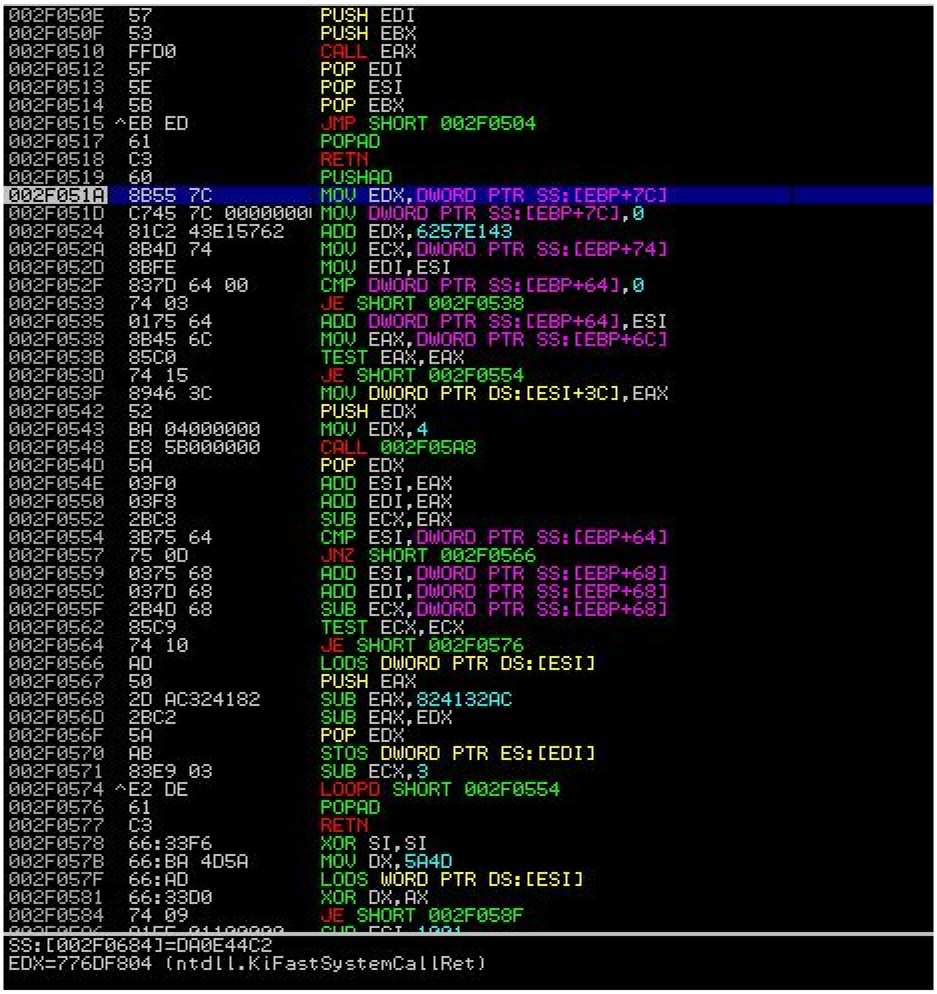
```
Address  Hex dump                 ASCII
00425320 FE B2 37 BF AD E5 78 41
00425328 5D 18 BA C3 08 4B FC 45
00425330 6C 7E 3D C8 18 B1 7E 4A
00425338 04 E4 BF CC B0 16 01 4F
00425340 5C 49 42 D1 08 7C 83 53
00425348 B4 AE C4 D5 60 E1 05 58
00425350 0C 14 47 DA B8 46 88 5C
00425358 64 79 C9 DE F8 AC 0A 61
00425360 B2 FE 05 F2 5E E5 50 41
00425368 2B D0 93 0F A4 24 29 FA
00425370 B9 CA 8A EC D7 6C 33 E1
00425378 E4 0C 95 C6 F1 AD 44 B8
00425380 11 01 E8 9F DD A5 9E 90
00425388 A9 41 4E 33 99 C3 E2 D5
00425390 B2 65 88 BD 8C A5 D6 49
00425398 5C D8 17 CC 08 0B 59 4E
004253A0 4E 48 6A 64 D8 E6 69 A7
004253A8 62 85 69 EA EC 23 69 2D
004253B0 49 CB 5E 70 D1 69 5E B3
004253B8 4A 00 83 F6 D1 9E 82 39
004253C0 DA 34 A7 7C 81 D3 A6 BF
004253C8 51 55 E6 02 C3 F3 E5 45
004253D0 93 75 25 89 CA 13 25 CC
004253D8 4E 8F 1A 0F C3 2D 1A 52
004253E0 4D CC 1A 95 0D 6A 1A D8
```

PEview - C:\Users\REM\Desktop\1

```
1c80b1ba2c514bc1.bin      pFile     Raw Data
  IMAGE_DOS_HEADER        00022F20  FE B2 37 BF AD E5 78 41  5D 18 BA C3 08 4B FC 45
  MS-DOS Stub Program     00022F30  6C 7E 3D C8 18 B1 7E 4A  04 E4 BF CC B0 16 01 4F
  IMAGE_NT_HEADERS        00022F40  5C 49 42 D1 08 7C 83 53  B4 AE C4 D5 60 E1 05 58
  IMAGE_SECTION_HEA       00022F50  0C 14 47 DA B8 46 88 5C  64 79 C9 DE F8 AC 0A 61
  IMAGE_SECTION_HEA       00022F60  B2 FE 05 F2 5E E5 50 41  2B D0 93 0F A4 24 29 FA
  IMAGE_SECTION_HEA       00022F70  B9 CA 8A EC D7 6C 33 E1  E4 0C 95 C6 F1 AD 44 B8
  IMAGE_SECTION_HEA       00022F80  11 01 E8 9F DD A5 9E 90  A9 41 4E 33 99 C3 E2 D5
  SECTION .text           00022F90  B2 65 88 BD 8C A5 D6 49  5C D8 17 CC 08 0B 59 4E
  SECTION .rdata          00022FA0  4E 48 6A 64 D8 E6 69 A7  62 85 69 EA EC 23 69 2D
  SECTION .data           00022FB0  49 CB 5E 70 D1 69 5E B3  4A 00 83 F6 D1 9E 82 39
```

Figure 9 Copied data location

The next call after the data is moved is interesting, some hardcoded dword values, two sub instructions with a load and store in a loop? That looks like an encoding loop of some kind.



```
002F050E   57               PUSH EDI
002F050F   53               PUSH EBX
002F0510   FFD0             CALL EAX
002F0512   5F               POP EDI
002F0513   5E               POP ESI
002F0514   5B               POP EBX
002F0515 ^EB ED             JMP SHORT 002F0504
002F0517   61               POPAD
002F0518   C3               RETN
002F0519   60               PUSHAD
002F051A   8B55 7C          MOV EDX,DWORD PTR SS:[EBP+7C]
002F051D   C745 7C 0000000  MOV DWORD PTR SS:[EBP+7C],0
002F0524   81C2 43E15762    ADD EDX,6257E143
002F052A   8B4D 74          MOV ECX,DWORD PTR SS:[EBP+74]
002F052D   8BFE             MOV EDI,ESI
002F052F   837D 64 00       CMP DWORD PTR SS:[EBP+64],0
002F0533   74 03            JE SHORT 002F0538
002F0535   0175 64          ADD DWORD PTR SS:[EBP+64],ESI
002F0538   8B45 6C          MOV EAX,DWORD PTR SS:[EBP+6C]
002F053B   85C0             TEST EAX,EAX
002F053D   74 15            JE SHORT 002F0554
002F053F   8946 3C          MOV DWORD PTR DS:[ESI+3C],EAX
002F0542   52               PUSH EDX
002F0543   BA 04000000      MOV EDX,4
002F0548   E8 5B000000      CALL 002F05A8
002F054D   5A               POP EDX
002F054E   03F0             ADD ESI,EAX
002F0550   03F8             ADD EDI,EAX
002F0552   2BC8             SUB ECX,EAX
002F0554   3B75 64          CMP ESI,DWORD PTR SS:[EBP+64]
002F0557   75 0D            JNZ SHORT 002F0566
002F0559   0375 68          ADD ESI,DWORD PTR SS:[EBP+68]
002F055C   037D 68          ADD EDI,DWORD PTR SS:[EBP+68]
002F055F   2B4D 68          SUB ECX,DWORD PTR SS:[EBP+68]
002F0562   85C9             TEST ECX,ECX
002F0564   74 10            JE SHORT 002F0576
002F0566   AD               LODS DWORD PTR DS:[ESI]
002F0567   50               PUSH EAX
002F0568   2D AC324182      SUB EAX,824132AC
002F056D   2BC2             SUB EAX,EDX
002F056F   5A               POP EDX
002F0570   AB               STOS DWORD PTR ES:[EDI]
002F0571   83E9 03          SUB ECX,3
002F0574 ^E2 DE             LOOPD SHORT 002F0554
002F0576   61               POPAD
002F0577   C3               RETN
002F0578   66:33F6          XOR SI,SI
002F057B   66:BA 4D5A       MOV DX,5A4D
002F057F   66:AD            LODS WORD PTR DS:[ESI]
002F0581   66:33D0          XOR DX,AX
002F0584   74 09            JE SHORT 002F058F
SS:[002F0684]=DA0E44C2
EDX=776DF804 (ntdll.KiFastSystemCallRet)
```

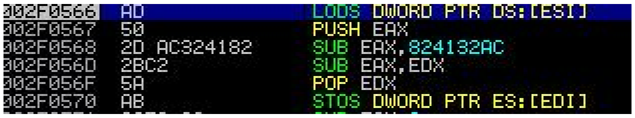
Figure 10 Decoding routine

It's good in these situations to keep track of what and where any hardcoded values are, such as the one loaded into EDX immediately and then added to a hardcoded value, turns out both values are hardcoded in the bytecode layer.

```
AX 01210000
CX 00000000
DX 3C662605
BX 00400000 1c80b1ba.004000
SP 0013FF28
```

Figure 11 Hardcoded value

Further down we see the previously mentioned loop that if you step through a few times you'll notice the PE file emerge, so this is the loop that we are concerned with since we know the file is in a resource section for this particular sample.



```
002F0566  AD                 LODS DWORD PTR DS:[ESI]
002F0567  50                 PUSH EAX
002F0568  2D AC324182        SUB EAX,824132AC
002F056D  2BC2               SUB EAX,EDX
002F056F  5A                 POP EDX
002F0570  AB                 STOS DWORD PTR ES:[EDI]
```

Figure 12 Decoding values

We have one hardcoded value and the previous two hardcoded values added together to get us the first two values subtracted, afterwords you can see EDX which contained one of those values is replaced with the previous dword value from our encoded data. We can construct this as a math routine:

$$f(x) = x - 0x824132AC - \Delta$$

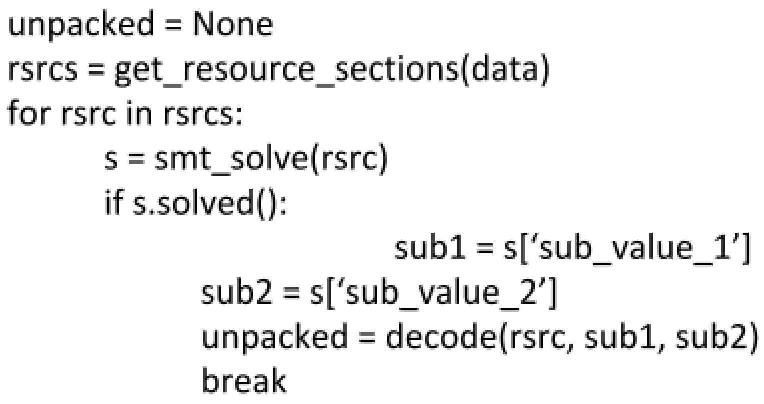
Figure 13 Proposed initial function

We know that delta is 0x3c662605 for the first iteration and delta becomes the previous x as it loops over the data. However when we are looking to decode out the binary we won't know the hardcoded value 0x824132ac and we also won't know the starting delta value. Simple enough to think of bruting out the values but that could be a pain, you would need to brute out one value from what you would expect to see in the first four bytes of a PE file and then try to figure out what the hardcoded value is from the next 4 bytes. Possible but could take a few cycles to brute, so instead you could decode out the bytecode layer and then use YARA[7] and regex patterns to try to find possible values instead to simplify this process but this approach can be error prone and end up being just as slow as bruting depending on how you implement it. The other option is to use an SMT solver, they can solve these types of problems very quickly because we know the endgame is a PE file and a PE file has a bunch of header data that we can predict.

3. SMT solving an unpacker

We basically did a walkthrough of the routine that decodes out the payload in the previous section. Up next we're going to go through how to turn this decoding routine, which is basically a math problem, into something that can be solved by an SMT.

Since we know the encoded data is in a resource section we can setup our overall program pseudocode as thus:

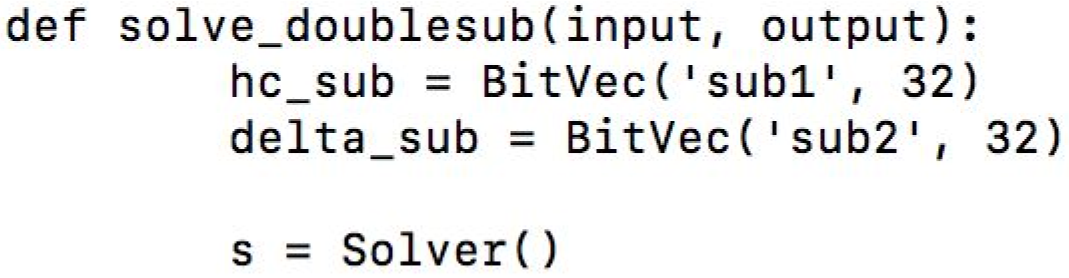


```
unpacked = None
rsrcs = get_resource_sections(data)
for rsrc in rsrcs:
        s = smt_solve(rsrc)
        if s.solved():
                                sub1 = s['sub_value_1']
                sub2 = s['sub_value_2']
                unpacked = decode(rsrc, sub1, sub2)
                break
```

Figure 14 Pseudocode

The gist is we will call a function on every resource section which will handle setting up our SMT solver by adding in necessary constraints. What are our constraints? Simply that we know what the output of the decoding should be, a PE file, and we know the routine involved. This means the process of setting up our solver and adding constraints is basically just describing a problem and then letting it solve the problem for us.

Ultimately we have two values that we need to find, a hardcoded subtract DWORD value and another DWORD value that only gets used for the first iteration and then replaced with the previous encoded DWORD value. For our problem these values basically become variables and for simplicity we can use the first 12 bytes of the unpacked PE file of the sample we just went through, '\x4d\x5a\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00'. If you're asking "why 12 bytes", the answer is basically that the more data you have the more likely you are able to solve the problem for the actual variables. If you only add a constraint for the first 4 bytes, given that the values we are trying to find are DWORD values this gives us many possibilities to satisfy the problem. The more data we can add for constraints then the narrower we can make the list of possible values to satisfy our problem which in turn gives us a better chance of producing the actual values we need instead of a possible range of values. Let's begin setting up our solver function.



```
def solve_doublesub(input, output):
        hc_sub = BitVec('sub1', 32)
        delta_sub = BitVec('sub2', 32)

        s = Solver()
```

Figure 15 Initial solver function

Here we’ve setup the beginning of our solver function and declared them as BitVecs which are basically variables, in this case 32 bit variables named sub1 and sub2 respectively. These will represent our hardcoded subtraction variable and our initial delta variable that we are trying to find out. The code can seem a bit weird at first, I found it best to think of these are your variable declarations. How you use your variables is by taking our math function above and unrolling a few iterations of the function into their equivalent y=x version which let’s us find the unknown values we are searching for because we know the y and x or the output and the input. Let’s take another look at our function.

$$f(x) = x - 0x824132AC - \Delta$$

Figure 16 Function

Now let’s replace some of the data to make it use our variables and turn it into the y=x form.

$$y = x - hcsub - deltasub$$

Figure 17 Function in yx form

We mentioned earlier that we know the inputs and the outputs already, the inputs are bytes that can be found inside our sample. The outputs are the first few bytes of a normal windows executable(‘\x4d\x5a\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00’), with this along with the inputs which as aforementioned are bytes that can be found inside the malware sample we are able to use our mathematical algorithm to find the values we need. We set this up by adding constraints which are conditions that will constraint our SMT solver.

```
    second_delta = struct.unpack_from('<I', input)[0]
    s.add((BitVecVal(struct.unpack_from('<I',input)[0], 32) - hc_sub) - delta_sub
== BitVecVal(struct.unpack_from('<I',output)[0], 32))
    s.add((BitVecVal(struct.unpack_from('<I',input[4:])[0], 32) - hc_sub) -
second_delta == BitVecVal(struct.unpack_from('<I',output[4:])[0], 32))
    s.add((BitVecVal(struct.unpack_from('<I',input[8:])[0], 32) - hc_sub) -
BitVecVal(struct.unpack_from('<I', input[4:])[0], 32) ==
BitVecVal(struct.unpack_from('<I',output[8:])[0], 32))
    return(s)
```

Figure 18 Setup SMT constraints

Now we can loop through every resource section and look for one that satisfies our constraints in the solver.

```
for rsrc in rsrcs:
 #Try z3 solvers
 a = bytearray(rsrc)
 for poss_decode in possible_decodes:
         s = solve_doublesub(a, poss_decode)
         if s.check() == sat:
                 m = s.model()
                 for d in m.decls():
                         if d.name() == 'sub1':
                                 sub1 = m[d].as_long()
                         elif d.name() == 'sub2':
                                 sub2 = m[d].as_long()
                 print("Satisfied!")
                 print("Sub1 Value: "+hex(sub1))
                 print("Sub2 Value: "+hex(sub2))
```

Figure 19 Try to find the encoded file

4. Conclusion and Future Work

In this paper I detailed a concept on how to approach looking at packers but creating an unpacker in this manner is not something feasible for every variant that exists, it is possible to do in one off scenarios where a researcher is tracking a specific malware family using a packer and wants to find what other families might either be used by the same group(for example a private packer or not sold) or perhaps what other groups are using the same packer(for example a public packer that is sold as a service). Either discovery tells a different story that can be useful for a researcher trying to paint a better picture over the workings of a threat group that might be leveraging malware[1,2]

There are some ways that current software could leverage some of the concepts presented in this paper however, auto unpacking solutions have existed for a number of years and they normally rely on a combination of sandboxes or virtual machines with specific loaded modules or software designed to look for binaries that are decoded and rebuilt into memory sections[3,4,5,14]. The concept here specifically of leveraging the decoding routines themselves could be used to expand the usefulness of these existing automated systems for finding interesting code sections that might not be detected via normal means. The main idea being that if the malware is decoding something then it’s potentially useful to someone so if you can find that specific routine you can harvest everything it decodes in an automated manner without having to guess later what was or wasn’t decoded. Such a system would need a way to heuristically detect where these routines are, a way to detect when they are executed and a way to dump or store the decoded data after they have finished running.

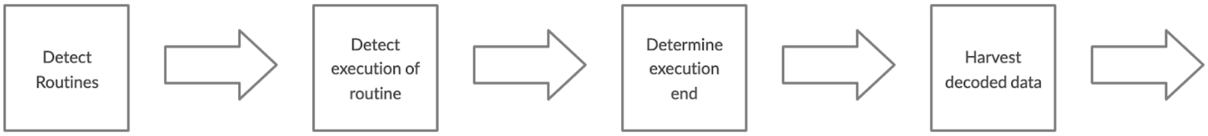


Figure 20 Overview of preprocessing packed files

It might also be beneficial for automated systems to detect signatures for the decoded data and then in the event of a miss for the decode routine but a detection on a decoded data signature determining in the execution where this signature fired so the address of the routine can be stored and potentially the entire execution restarted while monitoring this routine to harvest all decoded data.

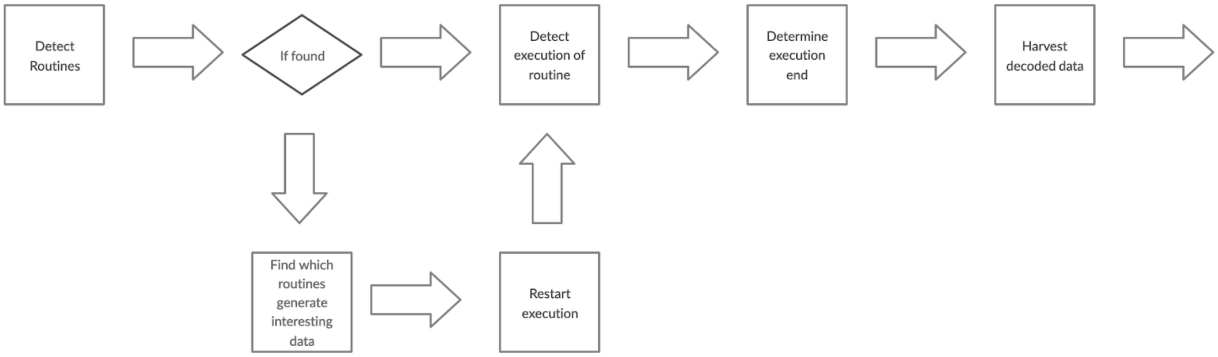


Figure 21 Overview of preprocessing packed files with finding routine during initial execution

This allows the possibility of harvesting more data from malware in an automated fashion but also being able to determine the most interesting routines to harvest which are the ones responsible for decoding data, these routines could then be passed to an engine designed for auto generating detecting binary detection rules or also stored for further review by researchers/analysts.

Future research will involve the heuristic static detection of malware and interesting routines that this concept could be leveraged against and a process of auto generating SMT solvers.

5. References

1: Reaves, Jason. *MAN1: Tracking the Crypter and the Actor,* https://vixra.org/abs/1902.0257.
2: Reaves, Jason. "GandCrab Overview and Crypter Reuse." *Random RE*, 1 Feb. 2018, sysopfb.github.io/malware/2018/02/01/gandcrab-overview.html.
3: https://cuckoosandbox.org/
4: https://www.unpac.me/
5: https://github.com/hasherezade/pe-sieve
6: https://github.com/Z3Prover/z3
7: https://virustotal.github.io/yara/
8: https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw
9: Nachreiner, Corey. "How Hackers Hide Their Malware: The Basics." *Dark Reading*, Dark Reading, 30 Aug. 2017, www.darkreading.com/how-hackers-hide-their-malware-the-basics/a/d-id/1329722.
10: Karl-Bridge-Microsoft. "PE Format - Win32 Apps." *Win32 Apps | Microsoft Docs*, docs.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-image-only.
11: https://attack.mitre.org/techniques/T1497/
12: https://attack.mitre.org/techniques/T1406/

13: Sikorski, Michael, and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
14: https://cape.contextis.com/