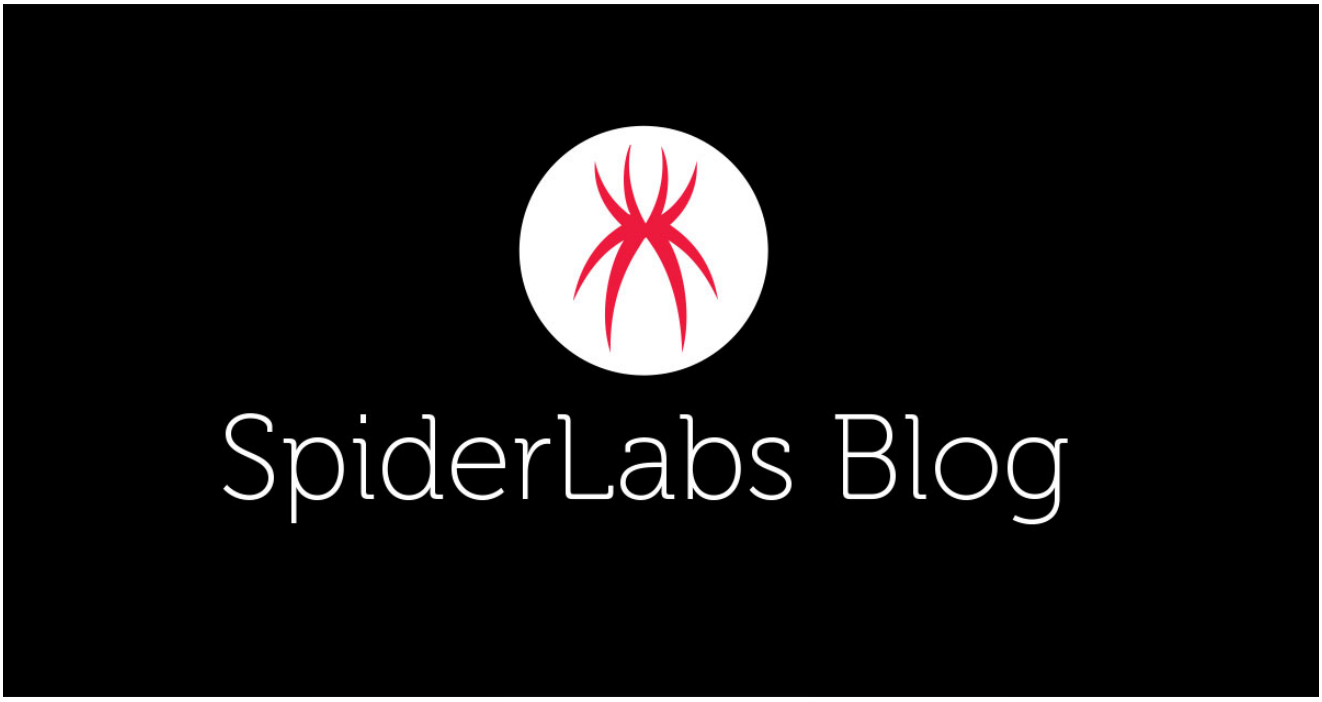
Undressing the REvil

trustwave.com/en-us/resources/blogs/spiderlabs-blog/undressing-the-revil/



Loading...

Blogs & Stories

SpiderLabs Blog

Attracting more than a half-million annual readers, this is the security community's go-to destination for technical breakdowns of the latest threats, critical vulnerability disclosures and cutting-edge research.

Contributors: Lloyd Macrohon and Rodel Mendrez

Recently, we got a chance to investigate a REvil Ransomware sample from one of our DFIR investigations. During analysis, we encountered a few stumbling blocks that made the investigation a little tricky, namely unpacking and string deobfuscation. In this blog, we will show how we manually unpacked the malware and then how we deobfuscated the strings used by the ransomware.

The particular sample we are going to investigate has the SHA256 Hash: 6ff970f1502347acd2d00e7746e40fba48995abbe26271d13102753c55694078.

Manual Unpacking

Packers are essentially tools that are used to compress a Portable Executable (PE) file. Many malware authors utilize packers with their malware to obfuscate and make it a bit harder to statically analyze code. If you want to learn more about packers, you can read our blog about this here: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/basic-packers-easy-as-pie/

We began by trying to determine what packer was used by this malware. The “Detect It Easy” tool failed to identify the packer and no signature was found. It also found no packed PE sections. Interestingly the sample had non-standard section names. We also took note of the entry point at RVA (relative virtual address) 0x9D58 which is within the *.text* section of the PE file.

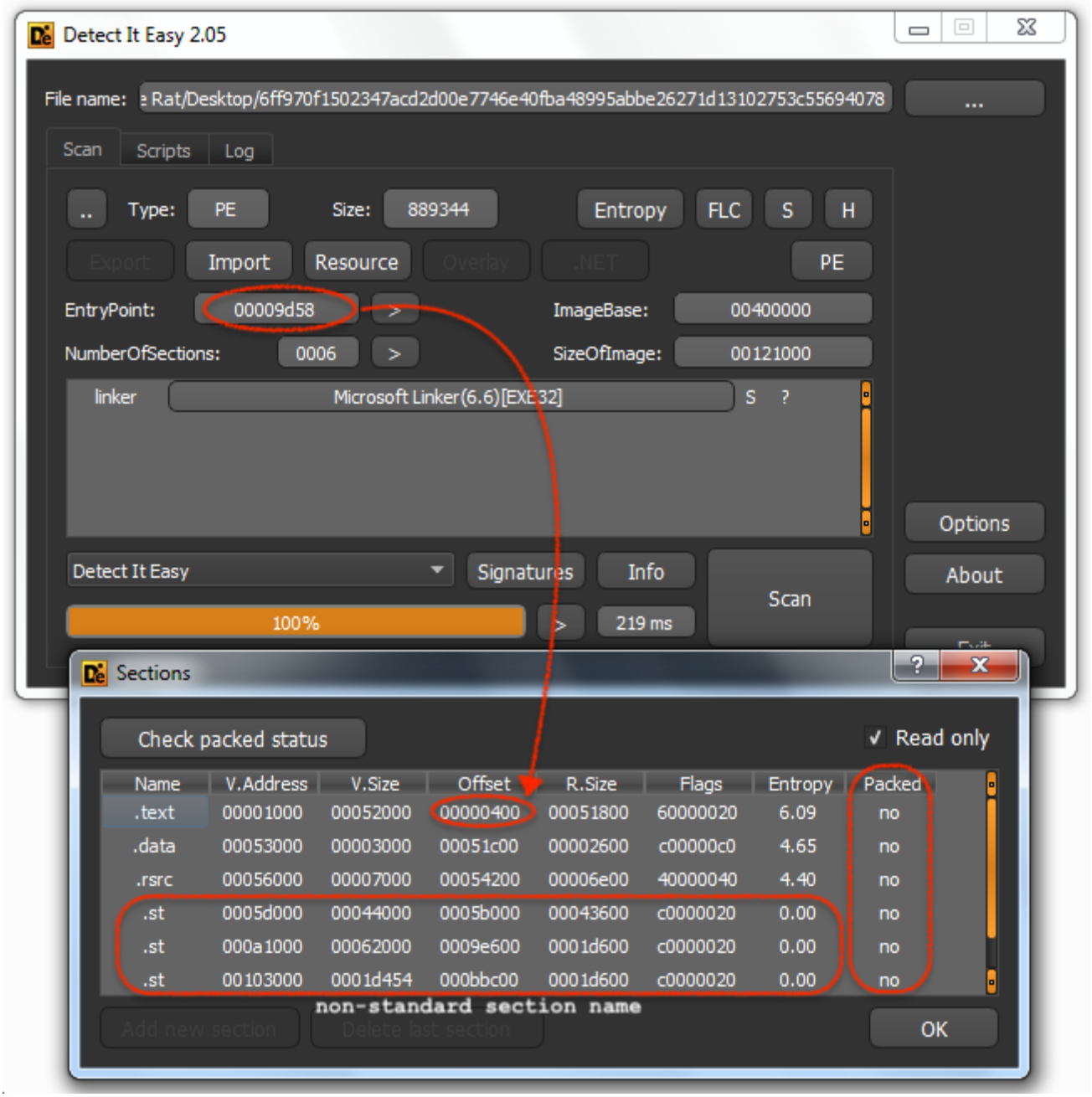
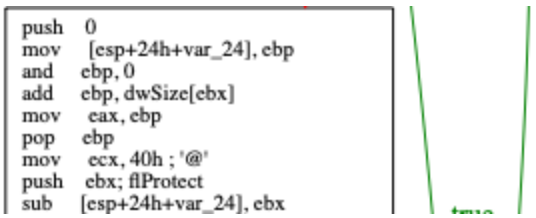
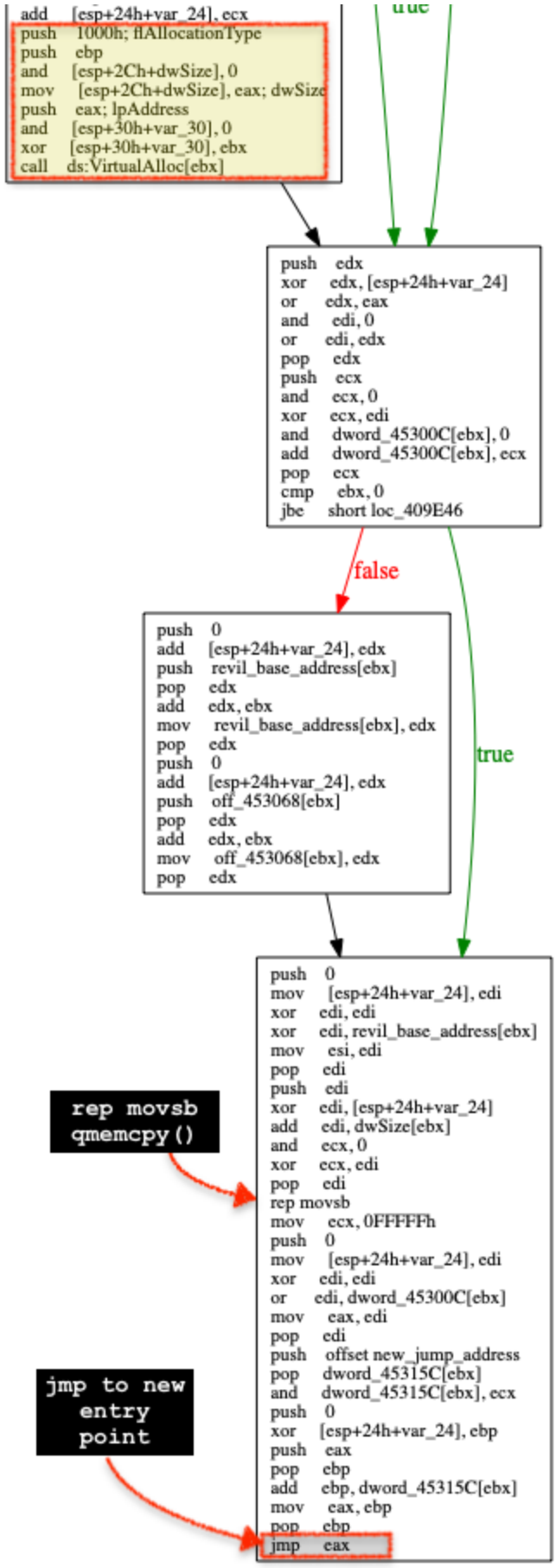


Figure 1. An overview of the malware's PE header information

Disassembling the executable, we can see right at the entry point the use of VirtualAlloc() API to allocate a new memory in the address space, then "*rep movsb*" opcode to copy data to the allocated memory, then, at the end of the function, is an opcode "*jmp eax"* that leads the instruction pointer to a new entry point in the allocated memory space. Now that looks interesting…





Here is what the code looks like when decompiled in IdaPro:

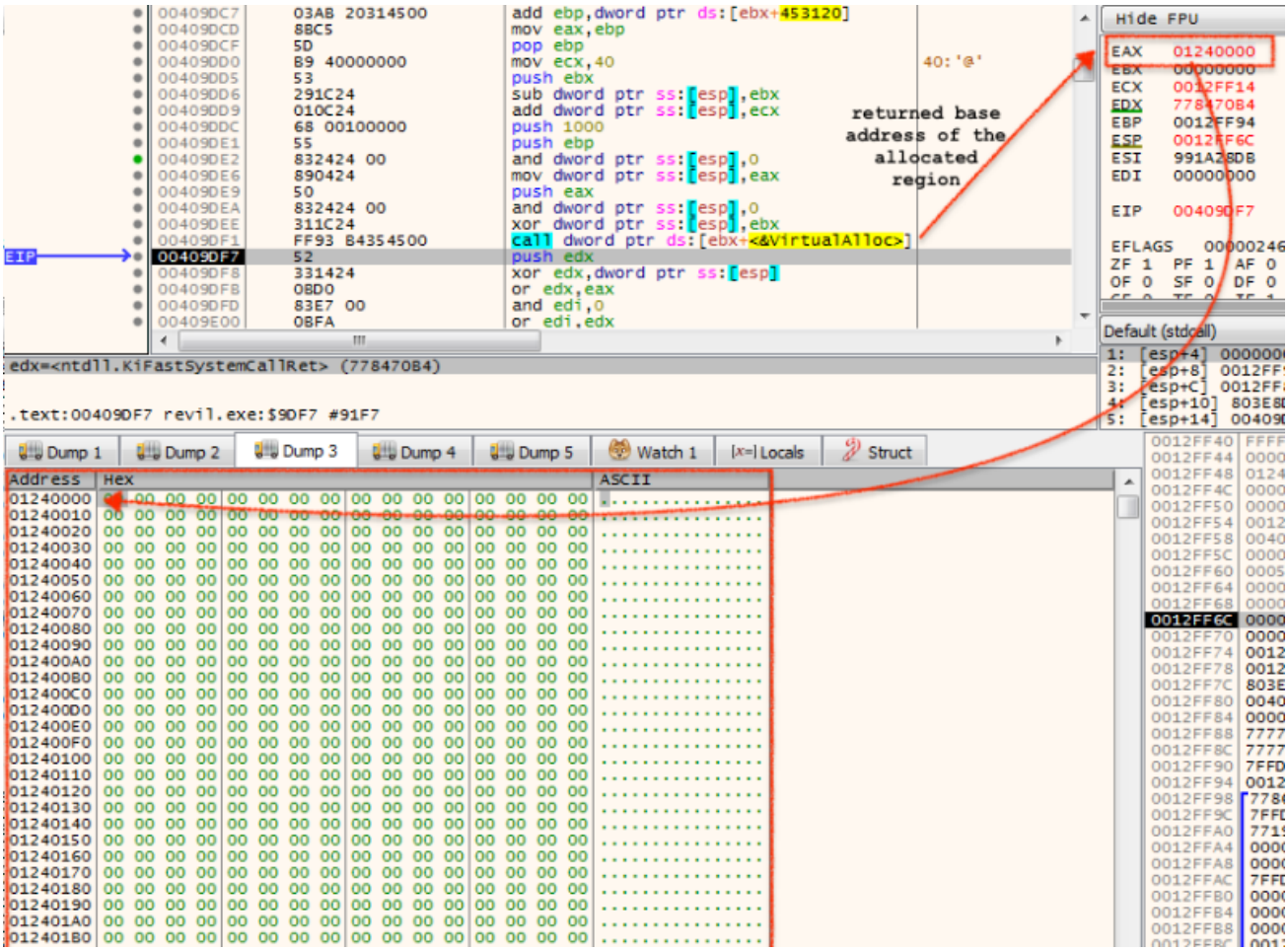
```
void __usercall start(int a1@<eax>, int a2@<edx>, int a3@<ecx>, int a4@<ebx>, int a5@<edi>, int a6@<esi>)
{
  int v6; // ebx
  int v7; // edi
  int v8; // ebx
  int v9; // esi
  int v10; // ebx
  int new_base_address; // eax
  int v12; // ebx
  int v13; // ecx
  int v14; // eax

  v6 = a5 + a2 + a4 + 1;
  v7 = a5 & 0xFFFFFF;
  v8 = v6 ^ a6;
  v9 = a6 & 0xFF00FFFF;
  v10 = v8 - a1;
  new_base_address = a1 & 0xFFFF00FF;
  v12 = a3 & v10;
  v13 = a3 & 0xFFFFFF00;
  if ( v12 )
    v12 &= v9 & v7;
  if ( v12 )
    v12 &= v13 & new_base_address;
  if ( !v12 )
  {
    new_base_address = sub_411E57(new_base_address, v13, 0, v9, 139383, 2116);
    if ( !new_base_address )
      new_base_address = (int)VirtualAlloc(0, dwSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  }
  *(int *)((char *)&dword_45300C + v12) = 0;
  *(int *)((char *)&dword_45300C + v12) += new_base_address;
  if ( v12 )
  {
    *(int *)((char *)&revil_base_address + v12) += v12;
    *(__int16 **)((char *)&off_453068 + v12) = (__int16 *)(*(char **)((char *)&off_453068 + v12) + v12);
  }
  qmemcpy(
    (void *)new_base_address,
    *(const void **)((char *)&revil_base_address + v12),
    *(int *)((char *)&dwSize + v12));
  v14 = *(int *)((char *)&dword_45300C + v12);
  *(int *)((char *)&dword_45315C + v12) = (int)&new_jump_address;
  *(int *)((char *)&dword_45315C + v12) &= 0xFFFFFu;
  __asm { jmp     eax }
}
```

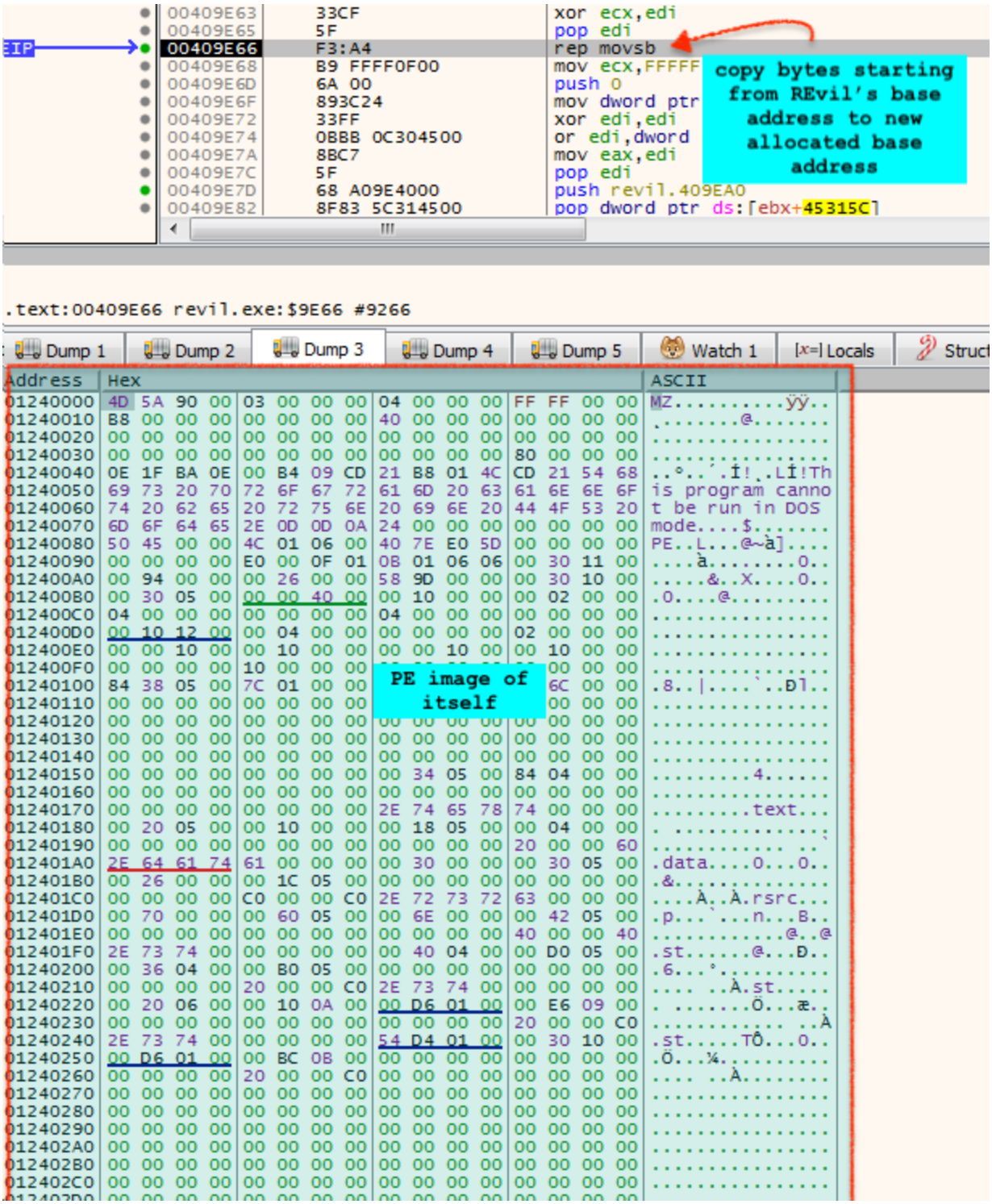
Copy data starting from base address to new allocated memory space

Jump to the new entry point

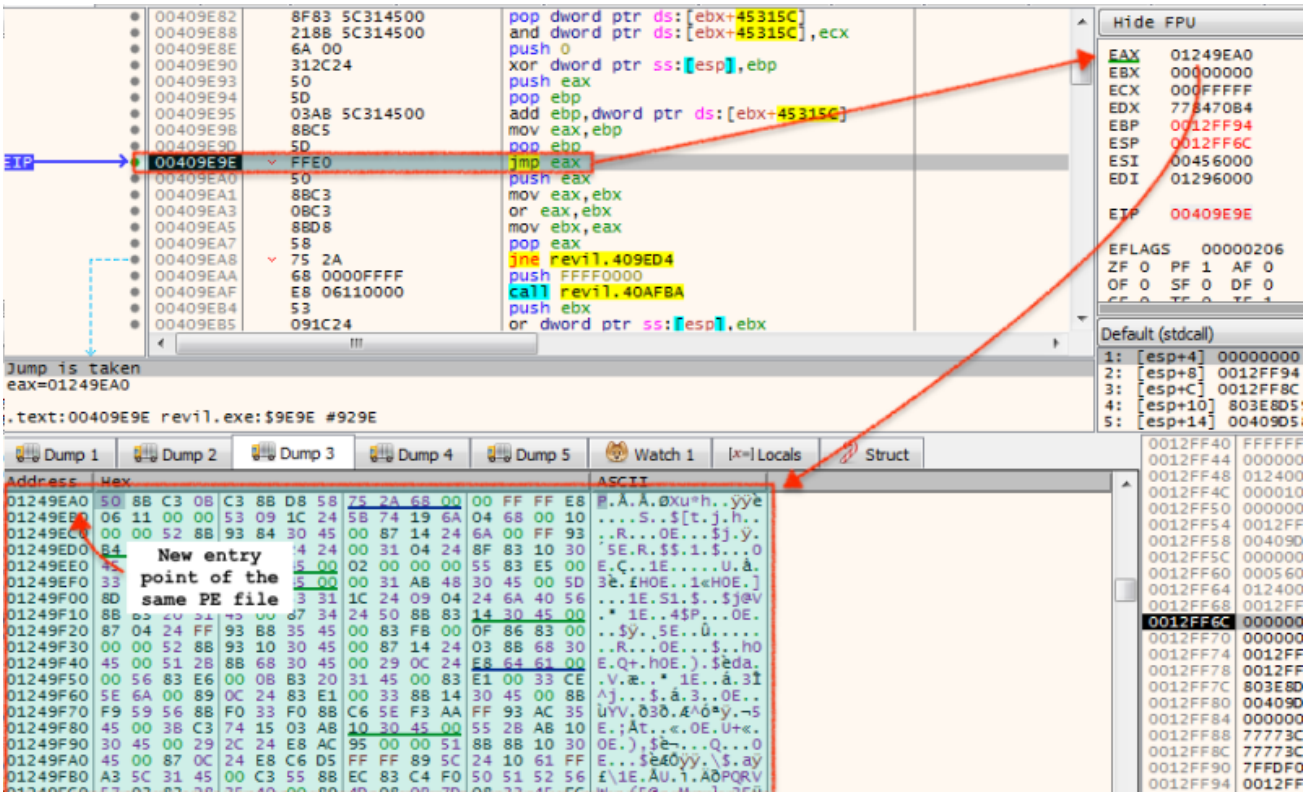
Next, we dynamically analyze the file using x64dbg (a PE debugger tool) to see what's being copied to the allocated memory. In the screenshot below, after calling VirtualAlloc() API, this instance allocated memory at the base address 0x1240000 (this memory address varies in each run).



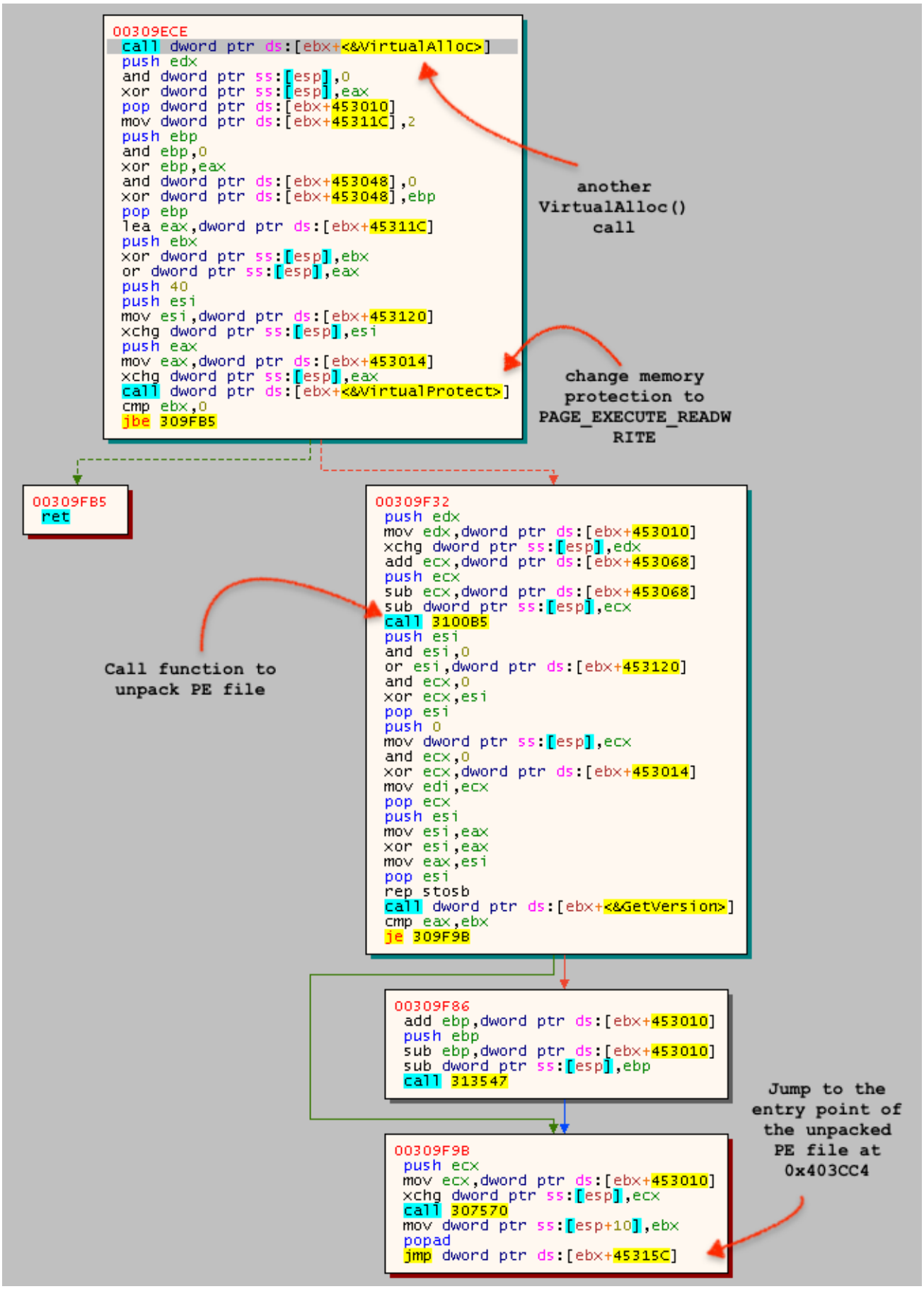
By dumping that memory address, we can visually monitor what has been copied to that address. After the malware has copied the data to the memory address, it turns out that it was the PE image of itself.



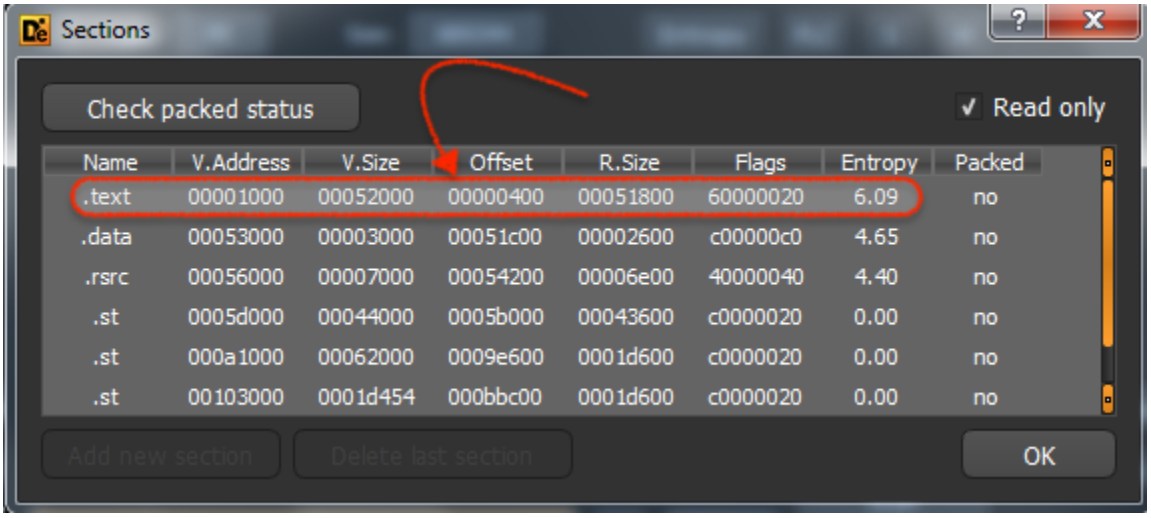
However, the *jmp eax* we mentioned earlier - jumps to a new entry point at RVA 0x9EA0 (virtual address is therefore - 0x1240000 (base address) + 0x9EA0 = 0x1249ea0):



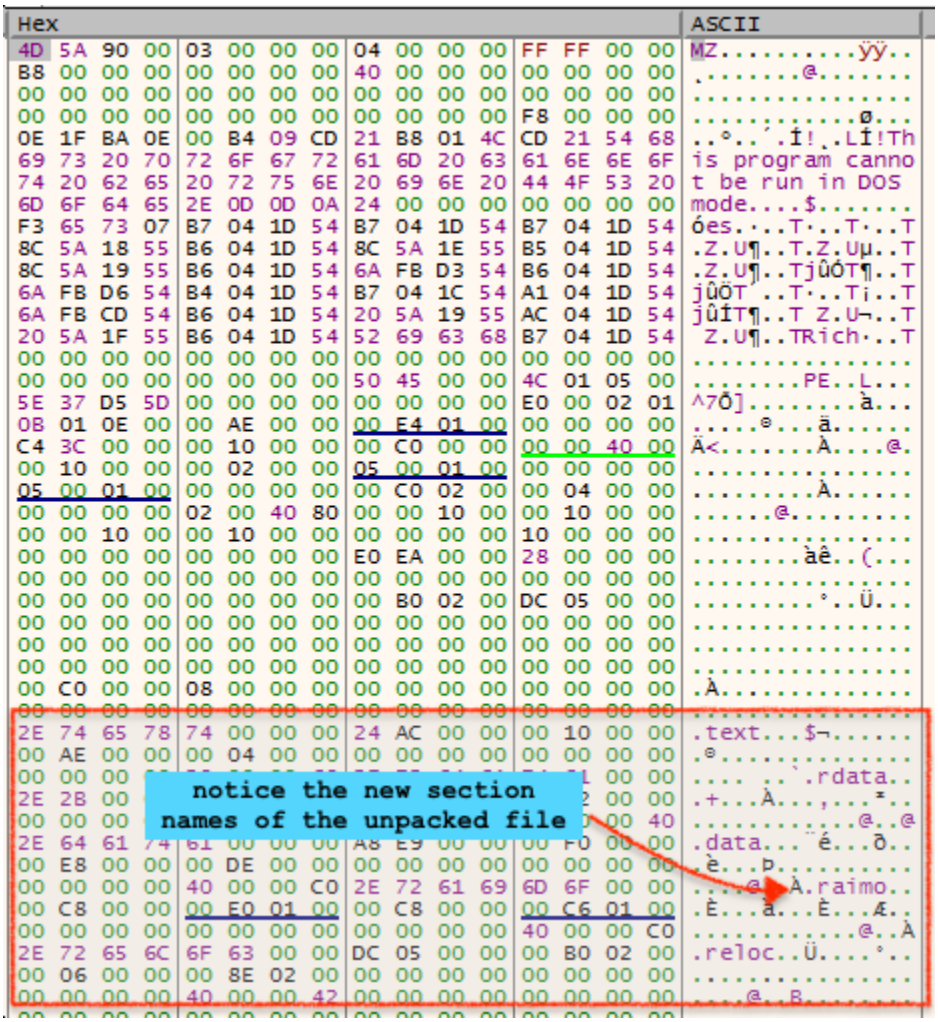
Next we follow that jump to virtual address 0x1249ea0, and yet again we encounter another VirtualAlloc() API call. So we dump the new allocated memory address and monitor it:


00309ECE
call dword ptr ds:[ebx+<&VirtualAlloc>]
push edx
and dword ptr ss:[esp],0
xor dword ptr ss:[esp],eax
pop dword ptr ds:[ebx+453010]
mov dword ptr ds:[ebx+45311C],2
push ebp
and ebp,0
xor ebp,eax
and dword ptr ds:[ebx+453048],0
xor dword ptr ds:[ebx+453048],ebp
pop ebp
lea eax,dword ptr ds:[ebx+45311C]
push ebx
xor dword ptr ss:[esp],ebx
or dword ptr ss:[esp],eax
push 40
push esi
mov esi,dword ptr ds:[ebx+453120]
xchg dword ptr ss:[esp],esi
push eax
mov eax,dword ptr ds:[ebx+453014]
xchg dword ptr ss:[esp],eax
call dword ptr ds:[ebx+<&VirtualProtect>]
cmp ebx,0
jbe 309FB5
another VirtualAlloc() call
change memory protection to PAGE_EXECUTE_READWRITE
00309FB5
ret
00309F32
push edx
mov edx,dword ptr ds:[ebx+453010]
xchg dword ptr ss:[esp],edx
add ecx,dword ptr ds:[ebx+453068]
push ecx
sub ecx,dword ptr ds:[ebx+453068]
sub dword ptr ss:[esp],ecx
call 3100B5
push esi
and esi,0
or esi,dword ptr ds:[ebx+453120]
and ecx,0
xor ecx,esi
pop esi
push 0
mov dword ptr ss:[esp],ecx
and ecx,0
xor ecx,dword ptr ds:[ebx+453014]
mov edi,ecx
pop ecx
push esi
mov esi,eax
xor esi,eax
mov eax,esi
pop esi
rep stosb
call dword ptr ds:[ebx+<&GetVersion>]
cmp eax,ebx
je 309F9B
Call function to unpack PE file
00309F86
add ebp,dword ptr ds:[ebx+453010]
push ebp
sub ebp,dword ptr ds:[ebx+453010]
sub dword ptr ss:[esp],ebp
call 313547
00309F9B
push ecx
mov ecx,dword ptr ds:[ebx+453010]
xchg dword ptr ss:[esp],ecx
call 307570
mov dword ptr ss:[esp+10],ebx
popad
jmp dword ptr ds:[ebx+45315C]
Jump to the entry point of the unpacked PE file at 0x403CC4

The malware then starts to decrypt a blob of data embedded in the PE file starting at RVA 0x13DE2 and writes it to the allocated memory. That blob of packed data (at 0ffset 0x13DE2) is within *.text* section of the PE file.



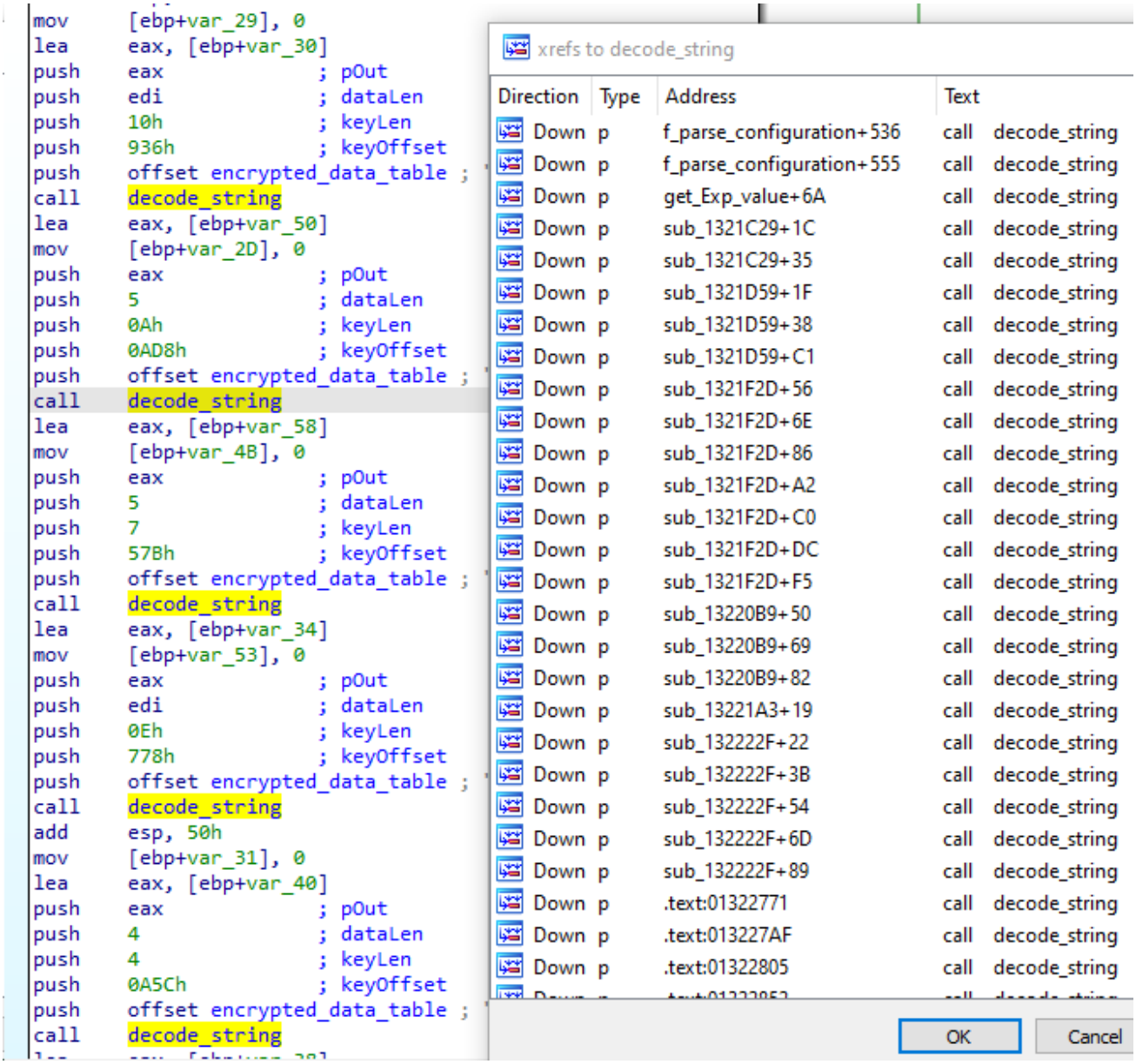
After unpacking, another PE file is revealed. This time, it is the ultimate payload – a REvil Ransomware. You will notice the new section names, and standing out is a non-standard section name *.raimo*.



We can dump this unpacked PE image and manually fix the IAT (Import Address Table) so that we can continue analyzing it statically. You can reconstruct and fix the IAT with Scylla (this is available in x64dbg) or ImportRec.

Reversing Encrypted Strings

Now that we have manually unpacked the file, we can statically analyze it. However, another stumbling block is that it leverages string obfuscation to hide the nature of what it's doing. You can see in the screenshot below a bunch of cross-references – these are calls to the decode_string function:

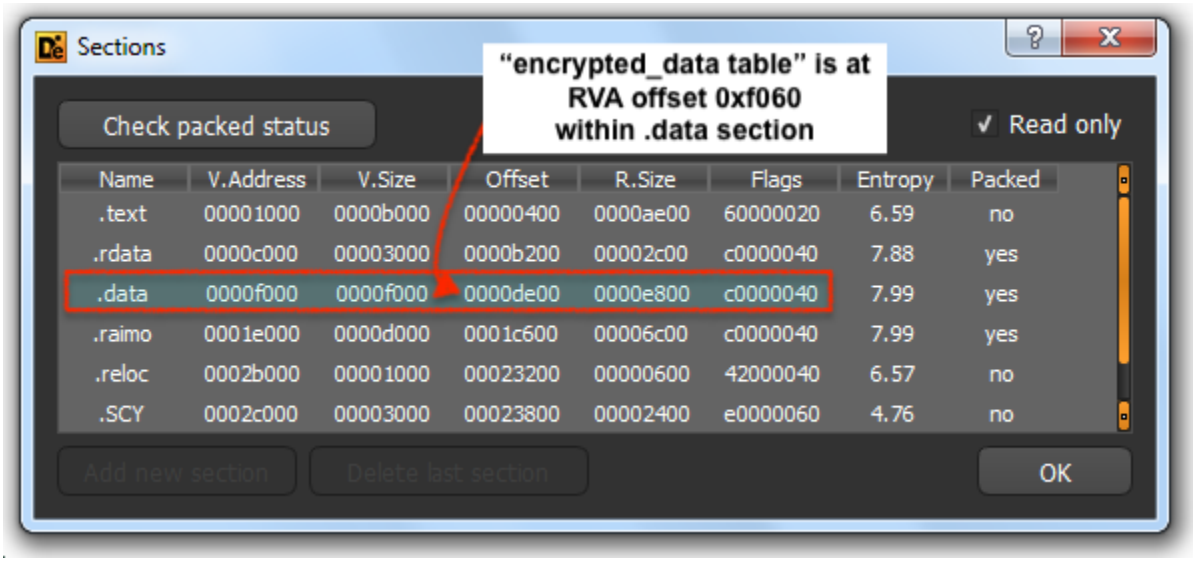


The encrypted strings are stored in parts of the binary. One part is a table of the encrypted strings that the malware uses and another part is the ransomware configuration. Each function call to decode_string is preceded by its parameters, by passing them through the

stack, these are: pointer to the output address, the key offset, key length and encoded data length. We will follow this example:

```
loc_1321E02:
lea     eax, [ebp+format]
push    eax             ; pOut
push    13Ah            ; dataLen
push    0Dh             ; keyLen
push    7B4h            ; keyOffset
push    offset encrypted_data_table ; Virtual Address = 0x132F060
call    decode_string
```

At relative virtual address 0xF060 is the data table base address which we named as **encrypted_data_table** which is 3048 bytes long. This is found in the **.data** section of the PE image.

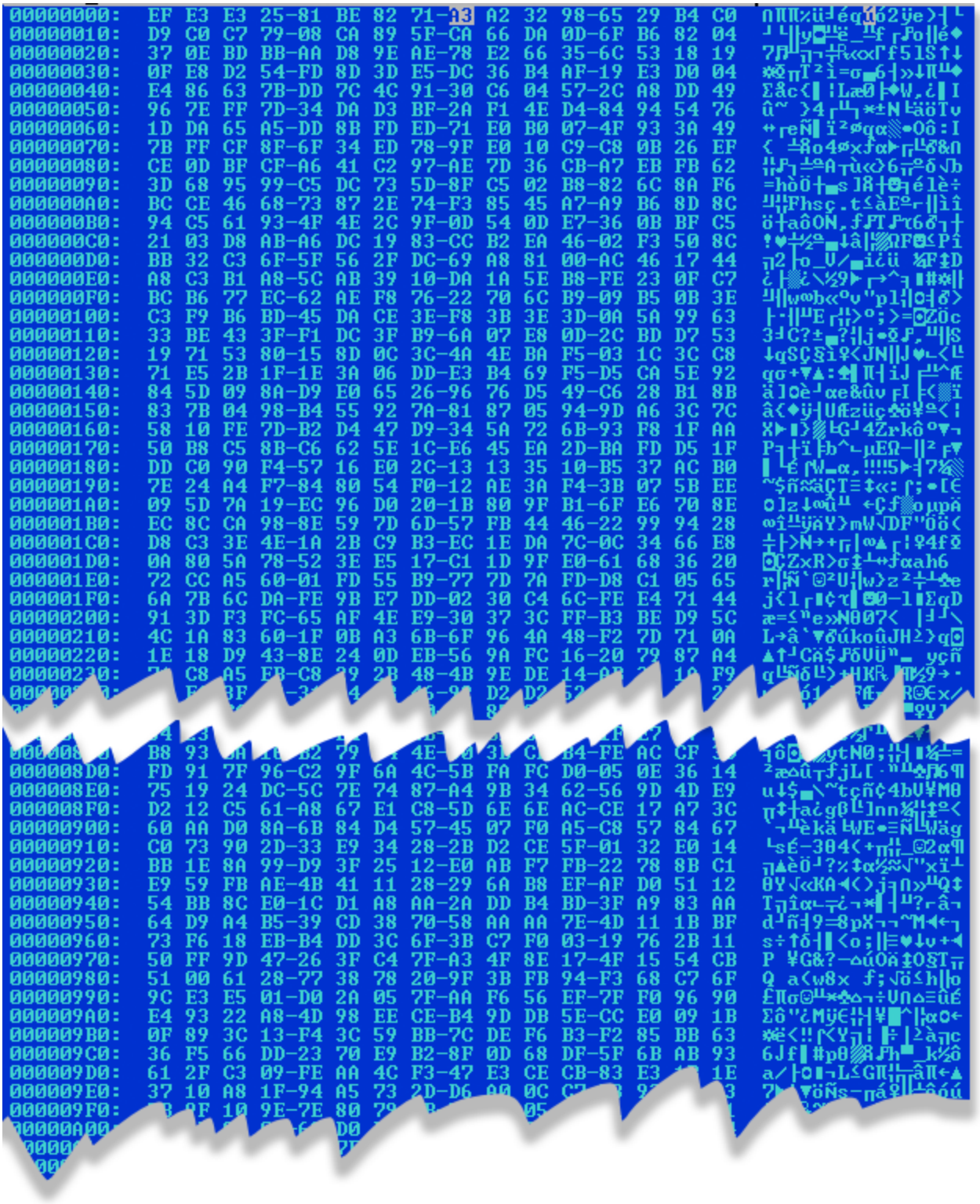


For this example, the encrypted data is at offset 0x91B from the base address of the data table, or 0xF060 + 0x7B4

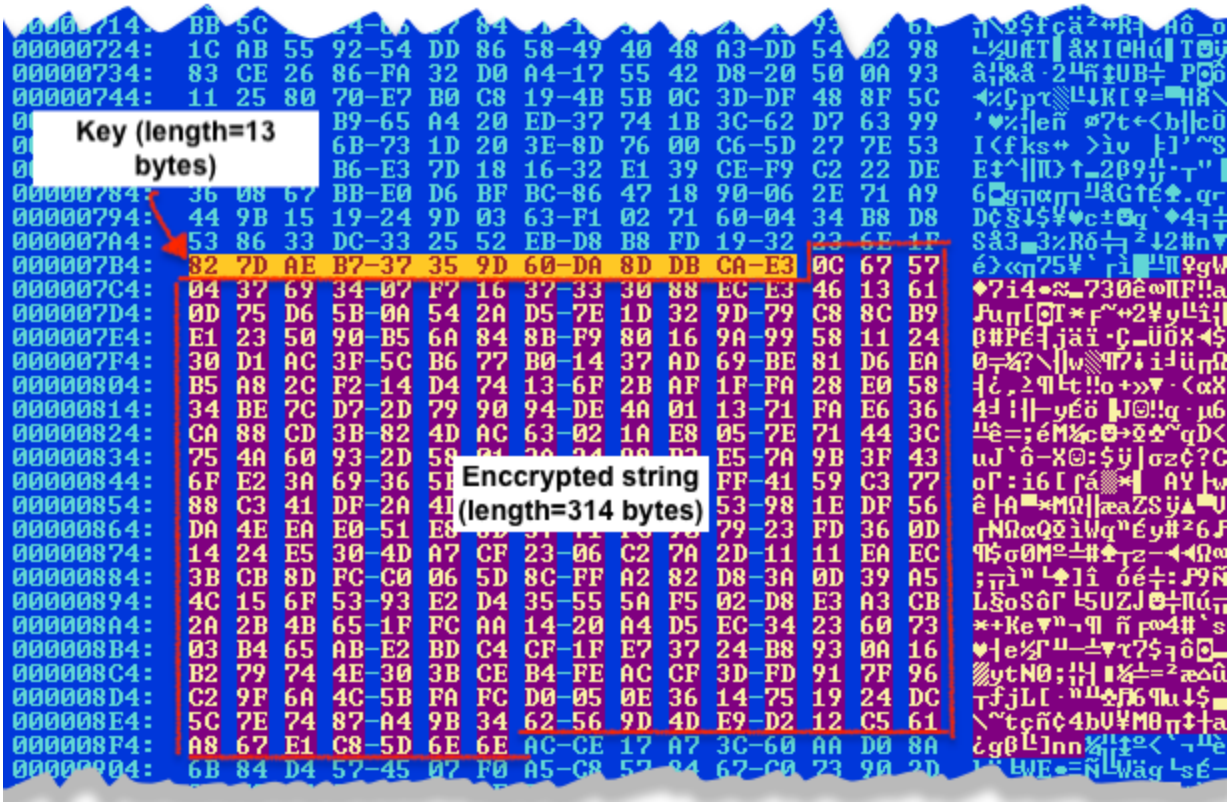
And the following parameters are hardcoded:

- offset address (from the base address 0xf060) of the encrypted string: 0x7B4
- key length: 0x0D (13 bytes)
- string length: 0x13A (314 bytes)

Here is the encrypted_data_table (truncated) after we re-based it to Zero:



The encrypted string block is therefore at 0x7B4:



The key is 13 bytes long :

```
[82 7D AE B7 37 35 9D 60 DA 8D DB CA E3]
```

And the encrypted string is 314 bytes long:

```
[0c 67 57 04 37 69 34 07 f7 16 37 33 30 88 ec e3
46 13 61 0d 75 d6 5b 0a 54 2a d5 7e 1d 32 9d 79
c8 8c b9 e1 23 50 90 b5 6a 84 8b f9 80 16 9a 99
58 11 24 30 d1 ac 3f 5c b6 77 b0 14 37 ad 69 be
81 d6 ea b5 a8 2c f2 14 d4 74 13 6f 2b af 1f fa
28 e0 58 34 be 7c d7 2d 79 90 94 de 4a 01 13 71
fa e6 36 ca 88 cd 3b 82 4d ac 63 02 1a e8 05 7e
71 44 3c 75 4a 60 93 2d 58 01 3a 24 98 b3 e5 7a
9b 3f 43 6f e2 3a 69 36 5b f4 a0 b1 2a dd ff 41
59 c3 77 88 c3 41 df 2a 4d ea d7 91 61 5a 53 98
1e df 56 da 4e ea e0 51 e8 8d 57 71 fc 90 79 23
fd 36 0d 14 24 e5 30 4d a7 cf 23 06 c2 7a 2d 11
11 ea ec 3b cb 8d fc c0 06 5d 8c ff a2 82 d8 3a
0d 39 a5 4c 15 6f 53 93 e2 d4 35 55 5a f5 02 d8
e3 a3 cb 2a 2b 4b 65 1f fc aa 14 20 a4 d5 ec 34
23 60 73 03 b4 65 ab e2 bd c4 cf 1f e7 37 24 b8
93 0a 16 b2 79 74 4e 30 3b ce b4 fe ac cf 3d fd
91 7f 96 c2 9f 6a 4c 5b fa fc d0 05 0e 36 14 75
19 24 dc 5c 7e 74 87 a4 9b 34 62 56 9d 4d e9 d2
12 c5 61 a8 67 e1 c8 5d 6e 6e]
```

Reverse engineering the decryption code in the malware shows that it's actually just the stream cipher RC4. Code snippet below is the RC4 algorithm: initialize sbox and key scheduling

```
char __cdecl sbox_init(char *sbox, char *key, unsigned int keyLen)
{
  int v3; // esi
  unsigned int i; // eax
  unsigned int v5; // ecx
  char v6; // bl
  char result; // al
  unsigned int v8; // [esp+Ch] [ebp-4h]

  LOBYTE(v3) = 0;
  for ( i = 0; i < 256; ++i )
    sbox[i] = i;
  v5 = 0;
  v8 = 0;
  do
  {
    v6 = sbox[v5];
    v3 = (unsigned __int8)(v3 + key[v5 % keyLen] + sbox[v5]);
    result = sbox[v3];
    sbox[v8] = result;
    v5 = v8 + 1;
    sbox[v3] = v6;
    v8 = v5;
  }
  while ( v5 < 0x100 );
  return result;
}
```

Code snippet below is the actual decryption of data:

```
int __cdecl data_decrypt(char *sbox, char *keyEndPtr, int end, _BYTE *output)
{
  int j; // edi
  int v5; // esi
  int result; // eax
  int v7; // ecx
  char *keyLen; // [esp+14h] [ebp+Ch]
  int enda; // [esp+18h] [ebp+10h]

  j = end;
  LOBYTE(v5) = 0;
  result = 0;
  if ( end )
  {
    keyLen = (char *)(keyEndPtr - output);
    do
    {
      v7 = (unsigned __int8)(result + 1);
      enda = v7;
      LOBYTE(v7) = sbox[v7];
      v5 = (unsigned __int8)(v5 + v7);
      sbox[(unsigned __int8)(result + 1)] = sbox[v5];
      sbox[v5] = v7;
      *output = output[(_DWORD)keyLen] ^ sbox[(unsigned __int8)(sbox[(unsigned __int8)(result + 1)] + v7)];// keystream generator
      result = enda;
      ++output;
      --j;
    }
    while ( j );
  }
  return result;
}
```

After reversing this to C, it was pretty straightforward to convert it to Python so we could run it in IDA Pro.

```
def decode(key, data):
    """ REvil string decoder, this is just RC4 """
    # initialize sbox and key scheduling
    sbox, j = [a for a in range(256)], 0
    for i in range(256):
        j = (j + key[i % len(key)] + sbox[i]) & 0xff
        sbox[i], sbox[j] = sbox[j], sbox[i]

    # this is the actual decryption of data to output
    i, j, output = 0, 0, bytearray(len(data))
    for k in range(len(data)):
        i = (i + 1) & 0xff
        j = (j + sbox[i]) & 0xff
        sbox[i], sbox[j] = sbox[j], sbox[i]
        t = (sbox[i] + sbox[j]) & 0xff
        # here is the actual decryption of the data
        output[k] = sbox[t] ^ data[k]
    return output
```

This now allows us to take the encrypted block above with the following parts:

```
key = '\x82\x7D\xAE\xB7\x37\x35\x9D\x60\xDA\x8D\xDB\xCA\xE3'
data = '\x0c\x67\x57\x04\x37\x69\x34\x07\xf7\x16\x37\x33\x30\x88\xec\xe3\x46\
print(decode(key,data))
```

This decodes a Unicode string as seen in the screenshot below:

```
{.".v.e.r.".:.%.d.,.".p.i.d.".:.".%.s.".,.".s.u.b.".:.".%.s.".,.".p.k.".:.".%.s.",
."u.i.d.".:.".%.s.".,.".s.k.".:.".%.s.".,.".u.n.m.".:.".%.s.".,.".n.e.t.".:.".%.s.
".,.".g.r.p.".:.".%.s.".,.".l.n.g.".:.".%.s.".,.".b.r.o.".:.%.s.,.".o.s.".:.".%.s."
.,.".b.i.t.".:.%.d.,.".d.s.k.".:.".%.s.".,.".e.x.t.".:.".%.s.".}.
```

Because each encoded string has its own unique key and variable length, it becomes cumbersome to decode every string. But fret not, at the end of this blog, we share the IDAPython script to aid you with the decoding process.

The second part of the obfuscated data is the ransomware configuration which basically uses the same RC4 algorithm. This encrypted configuration is stored in the non-standard named section called *.raimo.*

In the screenshot below we highlight the RC4 key "VNz47r3Wz2xT7DP1XqPa2MYcwUx8uRex", the CRC hash of the encoded data which is 0xB6C2E135, and the length of the data is 0x6B02 (27394 bytes).

```
.raimo:0133E000 key             db 'VNz47r3Wz2xT7DP1XqPa2MYcwUx8uRex'  RC4 key
.raimo:0133E000                                         ; DATA XREF: sub_1321B1C+40↑o
.raimo:0133E020 enc_conf_hash   dd 0B6C2E135h           ; DATA XREF: sub_1321B1C+17↑r
.raimo:0133E024 ; SIZE_T enc_conf_size
.raimo:0133E024 enc_conf_size   dd 6B02h                ; DATA XREF: sub_1321B1C+1↑r
.raimo:0133E024                                         ; sub_1321B1C+24↑r ...
.raimo:0133E028 ; char enc_config_data[]
.raimo:0133E028 enc_config_data db 8,'¤-ËFKæ”qÜH@',15h,'ÞƒI.i}œÄ²Zàèµò:T¦',0Fh,'ƒY–M',17h,8Fh,'$',0,'Ù'
.raimo:0133E028                                         ; DATA XREF: sub_1321B1C+7↑o
.raimo:0133E028                 db 13h,'||',16h,'ëÌ',15h,'`æäì³',27h,0Dh,'_kâ',17h,'*…Ù',1Eh,'³®',6,'è'
.raimo:0133E028                 db 'Õ•',90h,'T',3,'z',90h,'Ð',17h,'+š',18h,'†Ì',14h,'Ý_Æ•ZŒ',16h,'†²¢'
.raimo:0133E028                 db 8Fh,'zïO/çÇ…5Šù}à Encrypted COnfiguration Fh,'~r&¡',8Fh,'‹vý8S¾³´'
.raimo:0133E028                 db 'A²',1,';öi',1,'áMqhç€w+†´²U#÷V=O&~1j',1Eh,18h,90h,1Eh,'Ò’',9,0Eh,'-'
.raimo:0133E028                 db 'xŸ«i`ž,Ð®Ša',6,'L0',27h,'A_‰‰',7,'7<v',8Dh,'¼C£',0Ah
.raimo:0133E028                 db '^¡',1Bh,'äwß;d?_Ü¥ ßa}Jx',8Fh,'v',1Bh,11h,1Ah,'«AÜ³NÁ†',8Fh,'+Y‹E'
.raimo:0133E028                 db '?Õùžl]š',1Eh,'™÷[7üYýÃ',16h,'jÑê¬æŠ[€',16h,'<',12h,'e',6,'V”ÏÞœÉÝ'
.raimo:0133E028                 db 'f~ð',3,90h,'°"axÝ*xFk3^Rz',9,'ÊHV&ÿ',2,'†bBµ^Ë&Oœh‘(„ñ–XÍÈ€íØD¹ø¿'
.raimo:0133E028                 db '¬éžžš',13h,'ÜêO1ó',27h,'¼',7Fh,'/',1Eh,'r=:>âÌÐW',0Ah
```

The resulting decrypted configuration file looks like this:

```
{
    "pk": "/SCIwBFa1lAeOgTv32PoxS2LkBk3Bs6aDnd5JExyG3g=",
    "pid": "26",
    "sub": "2095",
    "dbg": false,
    "fast": false,
    "wipe": true,
    "wht": {
        "fld": [ ...
        ],
        "fls": [ ...
        ],
        "ext": [ ...
        ]
    },
    "wfld": [ ...
    ],
    "prc": [ ...
    ],
    "dmn": "mgimalta.com;makingmillionaires.net;walterman.es;eurethicsport.eu;smartspeak.com;ar
    "net": false,
    "svc": [ ...
    ],
    "nbody": "LQAtAC0AIAA9AD0APQAgAFcAZQBsAGMAbwBtAGUAIABDAHkAcgB1AHMATwBuAGUAIABhAG4AZAAgAEQAZ
    "nname": "{EXT}-readme.txt",
    "exp": false,
    "img": "QQBsAGwAIABvAGYAIAB5AG8AdQByACAAZgBpAGwAZQBzACAAYQByAGUAIABlAG4AYwByAHkAcAB0AGUAZAA
    "arn": false
}
```

When we finally unpack the file and deobfuscate the string, the process of reversing the code statically is so much easier. We won't, however, go into further detail about the Ransomware itself as there are very good analyses on this malware elsewhere, such as: https://www.acronis.com/en-eu/articles/sodinokibi-ransomware/

As mentioned earlier, we also wrote an IDAPython script to help deobfuscate strings hidden by this malware which may aid in the analysis process. You can find it here: https://github.com/bizdak/malware-analysis/blob/master/revil/revil.py

Decoded string after running the IdaPython script:

```
decode_string(encrypted_data, 1319, 12, 20, s_wpContent);// wp-content
zwpContent = 0;
decode_string(encrypted_data, 1450, 7, 12, s_static);// static
null0 = 0;
decode_string(encrypted_data, 2707, 9, 14, s_content);// content
v21 = 0;
decode_string(encrypted_data, 197, 7, 14, s_include);// include
v19 = 0;
decode_string(encrypted_data, 231, 5, 14, s_uplo);// uploads
v17 = 0;
decode_string(encrypted_data, 1826, 7, 8, s_news);// news
v41 = 0;
decode_string(encrypted_data, 2845, 16, 8, s_data);// data
v39 = 0;
decode_string(encrypted_data, 2514, 7, 10, s_admin);// admin
v31 = 0;
```

Happy reversing!