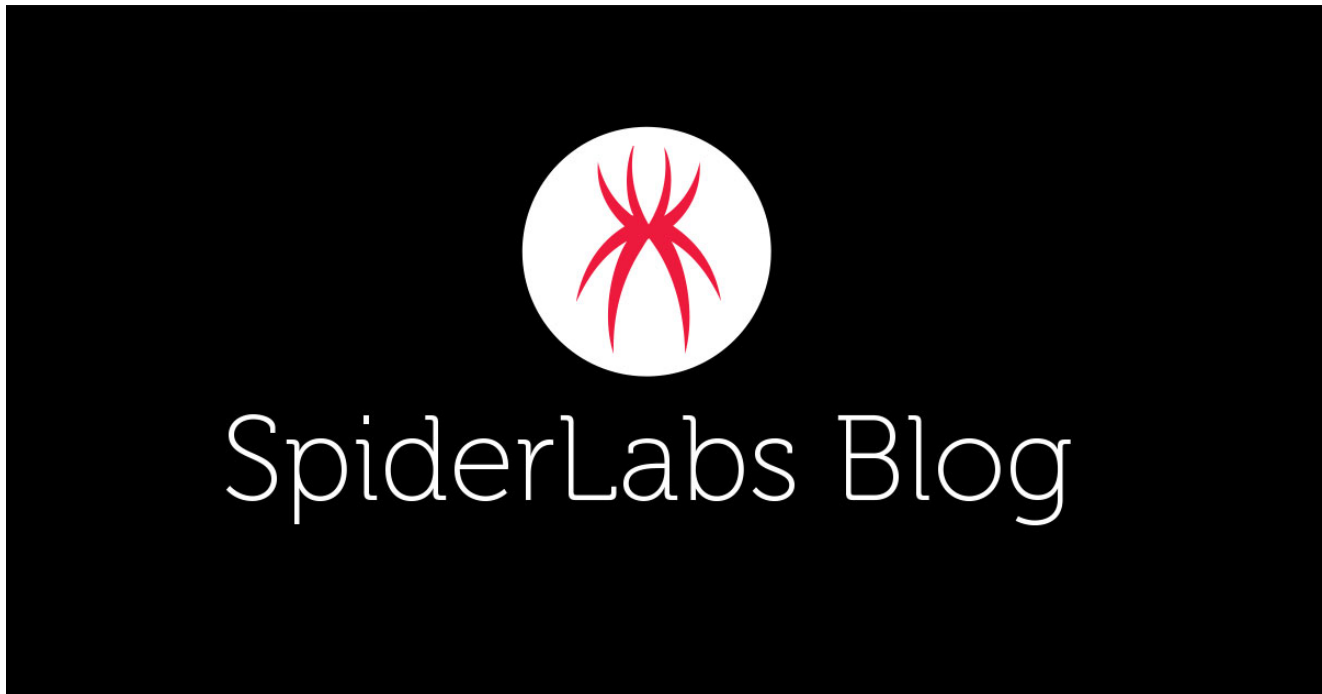


# Undressing the REvil

---

 [trustwave.com/en-us/resources/blogs/spiderlabs-blog/undressing-the-revil/](https://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.

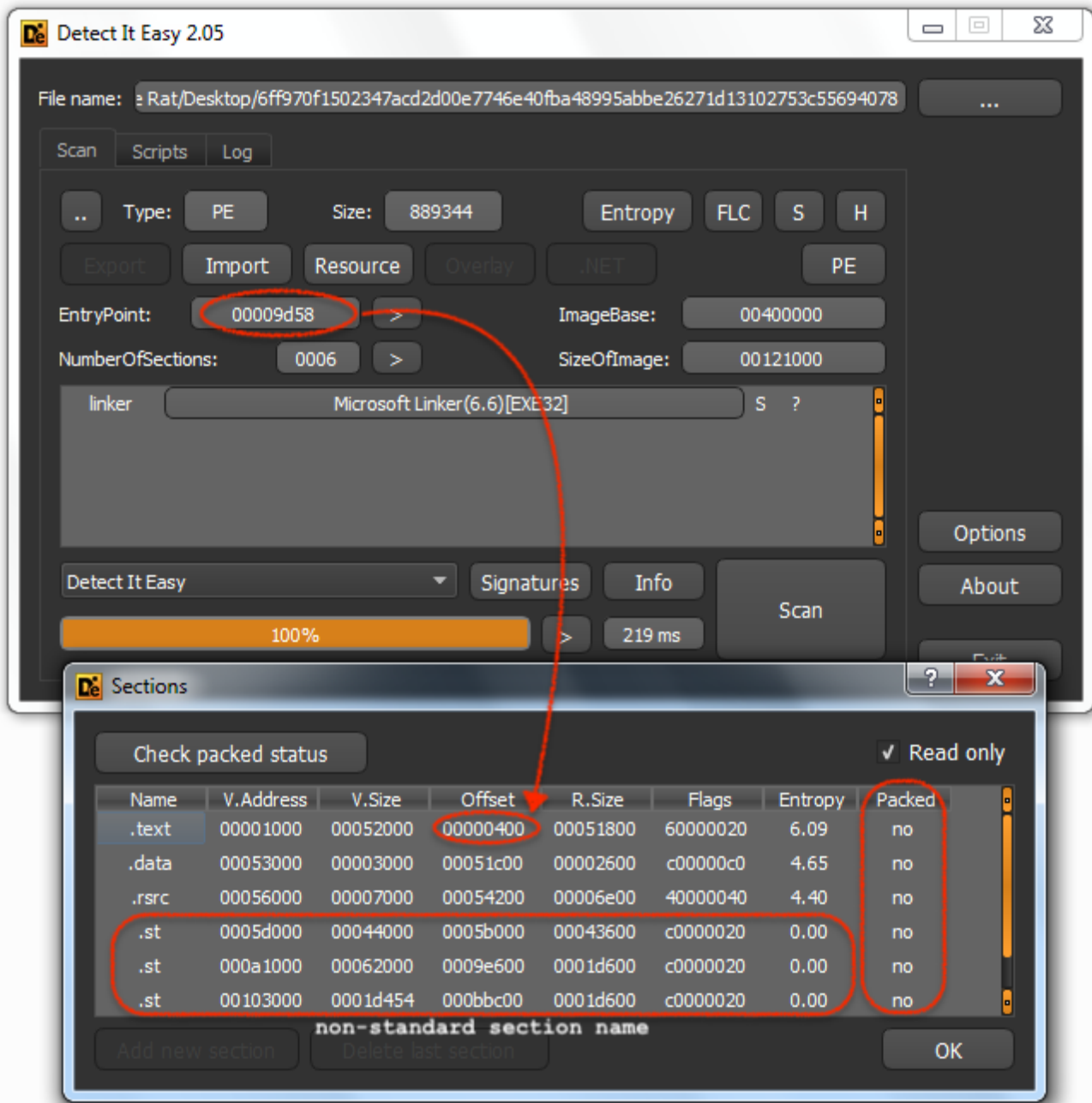


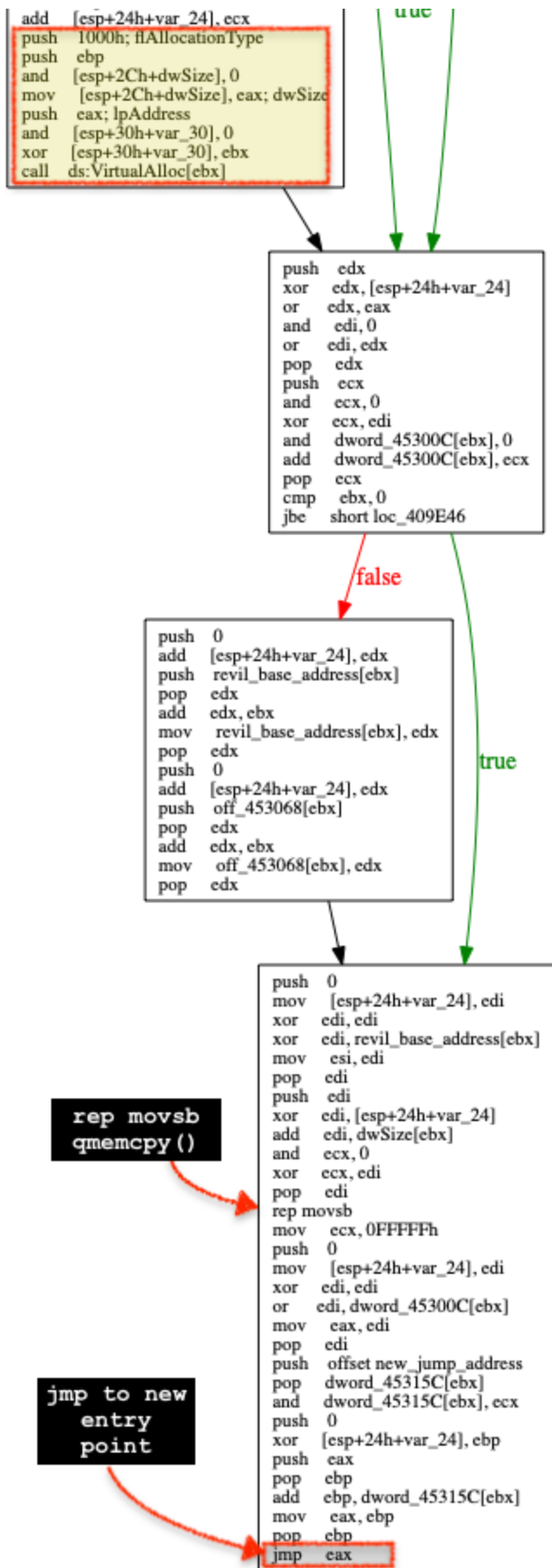
Figure 1. An overview of the malware's PE header information

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...

```

push 0
mov [esp+24h+var_24], ebp
and ebp, 0
add ebp, dwSize[ebx]
mov eax, ebp
pop ebp
mov ecx, 40h ; '@'
push ebx; flProtect
sub [esp+24h+var_24], ebx

```



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

```

1 void __usercall start(int a1@<eax>, int a2@<edx>, int a3@<ecx>, int a4@<ebx>, int a5@<edi>, int a6@<esi>)
2 {
3     int v6; // ebx
4     int v7; // edi
5     int v8; // ebx
6     int v9; // esi
7     int v10; // ebx
8     int new_base_address; // eax
9     int v12; // ebx
10    int v13; // ecx
11    int v14; // eax
12
13    v6 = a5 + a2 + a4 + 1;
14    v7 = a5 & 0xFFFFFFFF;
15    v8 = v6 ^ a6;
16    v9 = a6 & 0xFF00FFFF;
17    v10 = v8 - a1;
18    new_base_address = a1 & 0xFFFF00FF;
19    v12 = a3 & v10;
20    v13 = a3 & 0xFFFFFFFF00;
21    if ( v12 )
22        v12 &= v9 & v7;
23    if ( v12 )
24        v12 &= v13 & new_base_address;
25    if ( !v12 )
26    {
27        new_base_address = sub_411E57(new_base_address, v13, 0, v9, 139383, 2116);
28        if ( !new_base_address )
29            new_base_address = (int)VirtualAlloc(0, dwSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
30    }
31    *(int *)((char *)& dword_45300C + v12) = 0;
32    *(int *)((char *)& dword_45300C + v12) += new_base_address;
33    if ( v12 )
34    {
35        *(int *)((char *)& revil_base_address + v12) += v12;
36        *(__int16 *)((char *)& off_453068 + v12) = (__int16 *)((char *)& off_453068 + v12) + v12;
37    }
38    memcpy(
39        (void *)new_base_address,
40        *(const void *)((char *)& revil_base_address + v12),
41        *(int *)((char *)& dwSize + v12));
42    v14 = *(int *)((char *)& dword_45300C + v12);
43    *(int *)((char *)& dword_45315C + v12) = (int)&new_jump_address;
44    *(int *)((char *)& dword_45315C + v12) &= 0xFFFFFu;
45    __asm { jmp     eax }
46 }

```

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).

The screenshot displays a debugger window with the following components:

- Assembly View:** Shows instructions starting from 004090C7. The instruction at 004090F7 is `push edx`, which is highlighted with a blue arrow pointing to the EIP register.
- Registers:** The EAX register is highlighted with a red box and contains the value 01240000. Other registers like ECX, EDX, and ESP are also visible.
- Memory Dump:** A table showing memory addresses from 01240000 to 012401B0. The first row (01240000) is highlighted with a red box and contains the hex value 00 00 00 00. A red arrow points from the EAX register to this row.
- Registers Window:** Shows the current state of registers, with EAX at 01240000.
- Registers Window (Default (stdcall)):** Shows stack frames with addresses like [esp+4] and [esp+8].

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.

00409E63	33CF	xor ecx,edi
00409E65	5F	pop edi
<b>00409E66</b>	<b>F3:A4</b>	<b>rep movsb</b>
00409E68	B9 FFFF0F00	mov ecx,FFFFFF
00409E6D	6A 00	push 0
00409E6F	893C24	mov dword ptr
00409E72	33FF	xor edi,edi
00409E74	0B8B 0C304500	or edi,dword
00409E7A	8BC7	mov eax,edi
00409E7C	5F	pop edi
00409E7D	68 A09E4000	push revil.409EA0
00409E82	8F83 5C314500	pop dword ptr ds:[ebx+45315C]

copy bytes starting from REvil's base address to new allocated base address

.text:00409E66 revil.exe:\$9E66 #9266

Address	Hex	ASCII
01240000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
01240010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	.....@.....
01240020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240030	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00	.....80.....
01240040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°...!!.Li!Th
01240050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
01240060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
01240070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$......
01240080	50 45 00 00 4C 01 06 00 40 7E E0 5D 00 00 00 00	PE..L...[~a]...
01240090	00 00 00 00 E0 00 0F 01 08 01 06 06 00 30 11 00	....à.....0..
012400A0	00 94 00 00 00 26 00 00 58 9D 00 00 00 30 10 00	....&..X...0..
012400B0	00 30 05 00 00 00 40 00 00 10 00 00 00 02 00 00	.0....@.....
012400C0	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	.....
012400D0	00 10 12 00 00 04 00 00 00 00 00 00 02 00 00 00	.....
012400E0	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	.....
012400F0	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00	.....
01240100	84 38 05 00 7C 01 00 00 00 00 00 00 6C 00 00 00	.8.. .....Dl..
01240110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240150	00 00 00 00 00 00 00 00 00 34 05 00 84 04 00 00	.....4.....
01240160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240170	00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00	.....text...
01240180	00 20 05 00 00 10 00 00 00 18 05 00 00 04 00 00	.....text...
01240190	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	.....0.....
012401A0	2E 64 61 74 61 00 00 00 00 30 00 00 00 30 05 00	.data...0...0..
012401B0	00 26 00 00 00 1C 05 00 00 00 00 00 00 00 00 00	&.....A.rsrc..
012401C0	00 00 00 00 C0 00 00 C0 2E 72 73 72 63 00 00 00	...A..A..n...B..
012401D0	00 70 00 00 00 60 05 00 00 6E 00 00 00 42 05 00	...A.st...@...D..
012401E0	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	...A.st...@...D..
012401F0	2E 73 74 00 00 00 00 00 00 40 04 00 00 D0 05 00	...A.st...@...D..
01240200	00 36 04 00 00 B0 05 00 00 00 00 00 00 00 00 00	6.....A.st...@...D..
01240210	00 00 00 00 20 00 00 C0 2E 73 74 00 00 00 00 00	6.....A.st...@...D..
01240220	00 20 06 00 00 10 0A 00 00 D6 01 00 00 E6 09 00	.....0...æ..
01240230	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 C0	.....0...A
01240240	2E 73 74 00 00 00 00 00 54 D4 01 00 00 30 10 00	.st...T0...0..
01240250	00 D6 01 00 00 BC 08 00 00 00 00 00 00 00 00 00	.0...%.....
01240260	00 00 00 00 20 00 00 C0 00 00 00 00 00 00 00 00	.....A.....
01240270	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240280	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01240290	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012402A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012402B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012402C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012402D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

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):

The screenshot displays a debugger interface with the following components:

- Assembly View:** Shows instructions from address 00409E82 to 00409E85. Instruction 00409E9E is highlighted with a red box and labeled as the current instruction. It is a `jmp eax` instruction. A red arrow points from this instruction to the register window.
- Register Window:** Shows the state of registers. `EAX` is highlighted in red and contains the value `01249EA0`. Other registers like `EBX`, `ECX`, `EDX`, `EBP`, `ESP`, `ESI`, and `EDI` are also visible.
- Stack Window:** Shows stack frames. The current frame is at `00409E9E`. It lists arguments for a `stdcall` function: `[esp+4]` (00000000), `[esp+8]` (0012FF94), `[esp+C]` (0012FF8C), `[esp+10]` (803E8D5), and `[esp+14]` (00409D05).
- Memory Dump:** Shows a dump of memory starting at address `01249EA0`. A red box highlights a section of memory starting at `01249E00` with the text:
 

```

      01249E00  50 8B C3 08 C3 8B D8 58 75 2A 68 00 00 FF FF E8  0 .A.A.0xu^h..yye
      01249E01  06 11 00 00 53 09 1C 24 58 74 19 6A 04 68 00 10  ...S..[t.j.h..
      01249E02  00 00 52 88 93 84 30 45 00 87 14 24 6A 00 FF 93  ..R...0E...$.y.
      01249E03  84          New entry          4 24 00 31 04 24 8F 83 10 30  SE.R.$$.1.$...0
      01249E04  33          point of the          5 00 02 00 00 00 55 83 E5 00  E.Ç..1E...U.ä.
      01249E05  8D          same PE file          3 31 1C 24 09 04 24 6A 40 56  ..1E.S1.$..$j@v
      01249E06  88          3 20 24 75 00 07 34 24 50 88 83 14 30 45 00  * 1E..45P...0E.
      01249E07  87 04 24 FF 93 88 35 45 00 83 FB 00 0F 86 83 00  ..5y..SE..ü....
      01249E08  00 00 52 88 93 10 30 45 00 87 14 24 03 88 68 30  ..R...0E...$.ho
      01249E09  45 00 51 28 88 68 30 45 00 29 0C 24  E8 64 61 00  E.Q+.h0E.).$eda.
      01249E0A  00 56 83 E6 00 08 B3 20 31 45 00 83 E1 00 33 CE  .V.e..* 1E..ä.3I
      01249E0B  5E 6A 00 89 0C 24 83 E1 00 33 88 14 30 45 00 8B  A).$.ä.3..0E..
      01249E0C  F9 59 56 88 F0 33 F0 88 C6 5E F3 AA FF 93 AC 35  uYV.030.40^y.-5
      01249E0D  45 00 38 C3 74 15 03 AB 10 30 45 00 55 28 AB 10  E.;At..«.0E.U+«.
      01249E0E  30 45 00 29 2C 24 E8 AC 95 00 00 51 88 88 10 30  OE.)$.e...Q...0
      01249E0F  45 00 87 0C 24 E8 C6 D5 FF FF 89 5C 24 10 61 FF  E...$e0yy. $.ay
      01249E10  A3 5C 31 45 00 C3 55 88 EC 83 C4 F0 50 51 52 56  \1E.AU.1.A0PQRV
      
```

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\_READW  
RITE

```

00309FB5
ret

```

Call function to  
unpack PE file

```

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

```

```

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

```

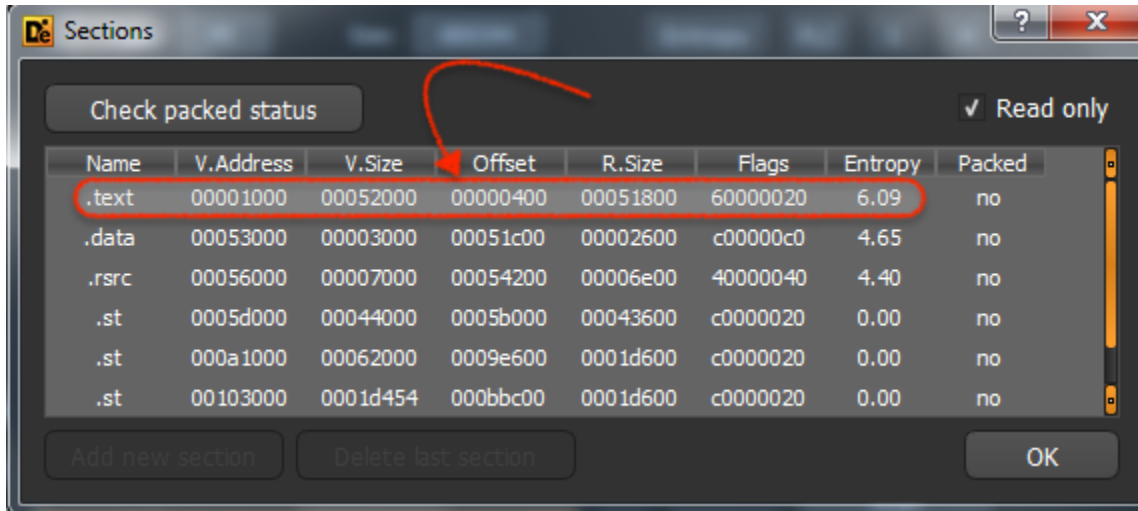
Jump to the  
entry point of  
the unpacked  
PE file at  
0x403CC4

```

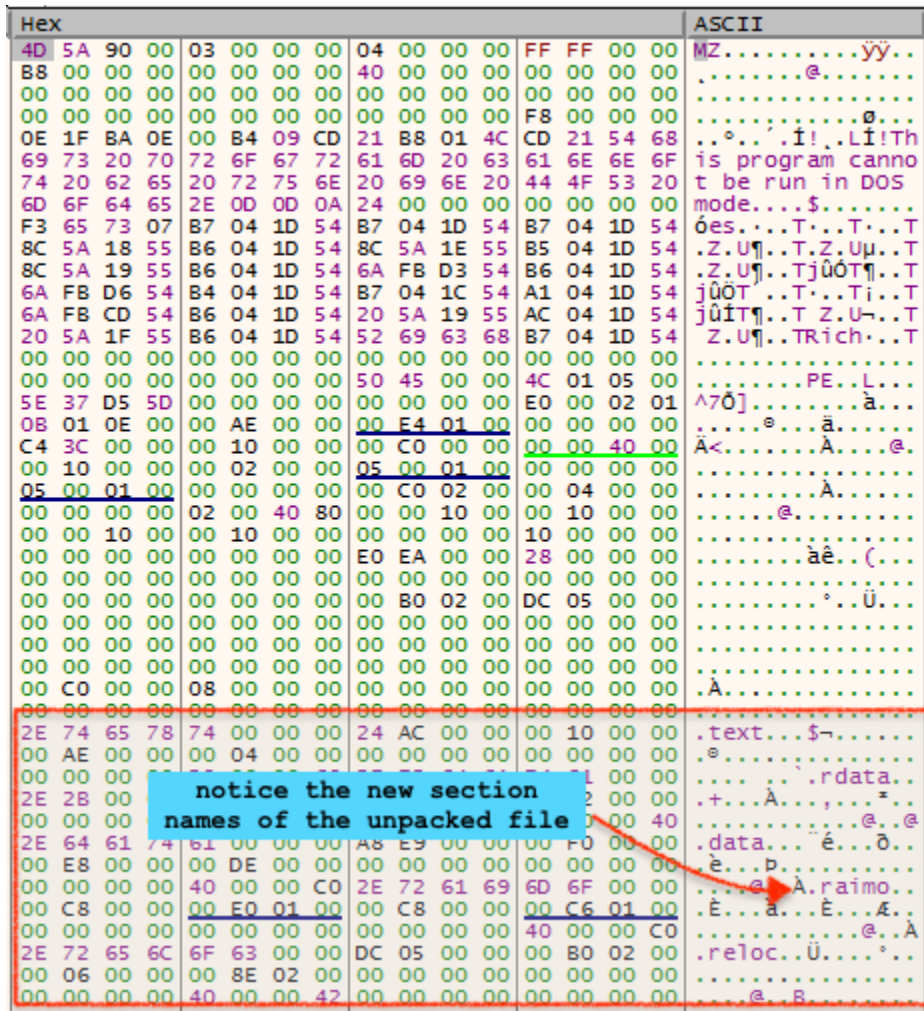
00309F9B
push ecx
mov ecx,dword ptr ds:[ebx+453010]
xchg dword ptr ss:[esp],ecx
call 307570
mov dword ptr ss:[esp+10],ebx
popad
jmp dword ptr ds:[ebx+45315C]

```

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 Offset 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 screenshot displays assembly code on the left and a cross-reference window titled "xrefs to decode\_string" on the right. The assembly code shows multiple calls to `decode_string` with various parameters. The cross-reference window lists the following entries:

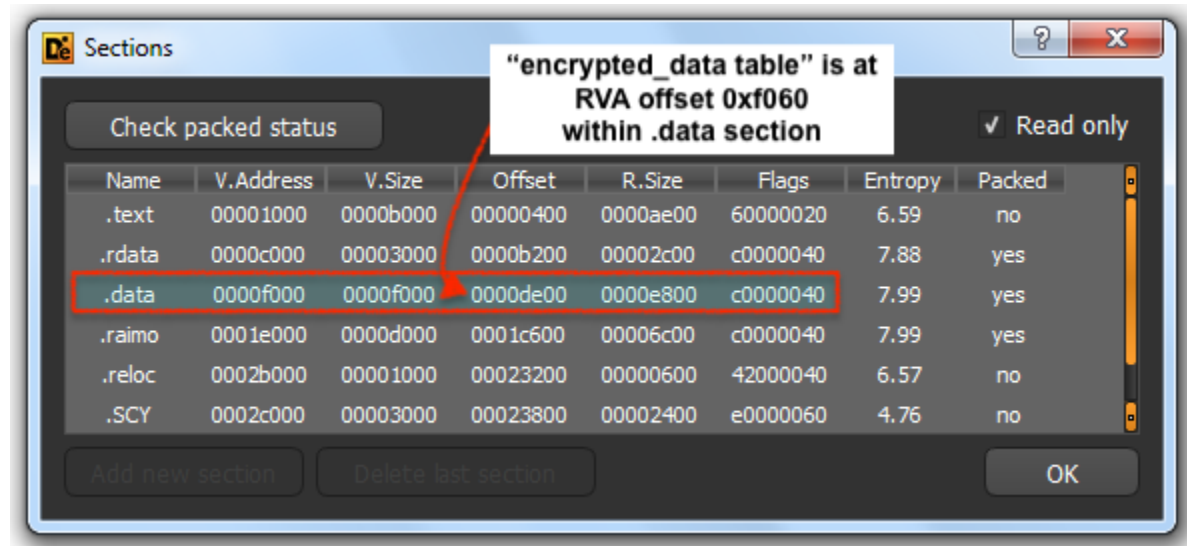
Direction	Type	Address	Text
Down	p	f_parse_configuration+536	call decode_string
Down	p	f_parse_configuration+555	call decode_string
Down	p	get_Exp_value+6A	call decode_string
Down	p	sub_1321C29+1C	call decode_string
Down	p	sub_1321C29+35	call decode_string
Down	p	sub_1321D59+1F	call decode_string
Down	p	sub_1321D59+38	call decode_string
Down	p	sub_1321D59+C1	call decode_string
Down	p	sub_1321F2D+56	call decode_string
Down	p	sub_1321F2D+6E	call decode_string
Down	p	sub_1321F2D+86	call decode_string
Down	p	sub_1321F2D+A2	call decode_string
Down	p	sub_1321F2D+C0	call decode_string
Down	p	sub_1321F2D+DC	call decode_string
Down	p	sub_1321F2D+F5	call decode_string
Down	p	sub_13220B9+50	call decode_string
Down	p	sub_13220B9+69	call decode_string
Down	p	sub_13220B9+82	call decode_string
Down	p	sub_13221A3+19	call decode_string
Down	p	sub_132222F+22	call decode_string
Down	p	sub_132222F+3B	call decode_string
Down	p	sub_132222F+54	call decode_string
Down	p	sub_132222F+6D	call decode_string
Down	p	sub_132222F+89	call decode_string
Down	p	.text:01322771	call decode_string
Down	p	.text:013227AF	call decode_string
Down	p	.text:01322805	call decode_string
Down	p	.text:01322857	call decode_string

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

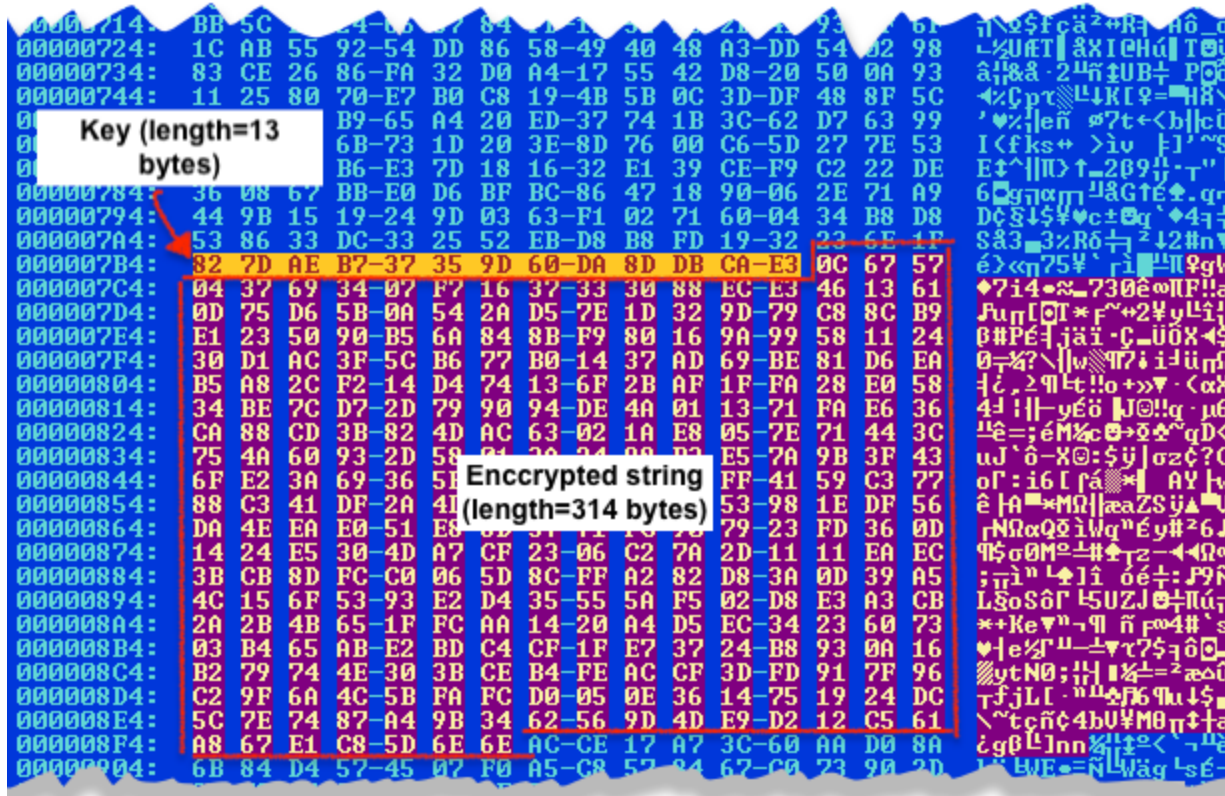
And the following parameters are hardcoded:

- 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:

```
00000000: EF E3 E3 25-81 BE 82 71-18 A2 32 98-65 29 B4 C0 nllziid eq02je>|L
00000010: D9 C0 C7 79-08 CA 89 5F-CA 66 DA 0D-6F B6 82 04 |lyuqz f rfo|le
00000020: 37 0E BD BB-AA D8 9E AE-78 E2 66 35-6C 53 18 19 7月 77-ffwod'f51S↑↓
00000030: 0F E8 D2 54-FD 8D 3D E5-DC 36 B4 AF-19 E3 D0 04 *8π'z i=σ_6-1>>llll
00000040: E4 86 63 7B-DD 7C 4C 91-30 C6 04 57-2C A8 DD 49 Σac<|!Lae0 fW, z| I
00000050: 96 7E FF 7D-34 DA D3 BF-2A F1 4E D4-84 94 54 76 ū~ >4 ll *eN kōTu
00000060: 1D DA 65 A5-DD 8B FD ED-71 E0 B0 07-4F 93 3A 49 +reN| i'z qo 00: I
00000070: 7B FF CF 8F-6F 34 ED 78-9F E0 10 C9-C8 0B 26 EF < =8o4xofa rL68n
00000080: CE 0D BF CF-A6 41 C2 97-AE 7D 36 CB-A7 EB FB 62 11P1 =A rü<0 6rδ-δ
00000090: 3D 68 95 99-C5 DC 73 5D-8F C5 02 B8-82 6C 8A F6 =h00+ms 18+01 lè÷
000000A0: BC CE 46 68-73 87 2E 74-F3 85 45 A7-A9 B6 8D 8C 21P Fhsc. t.±àE²-r||i i
000000B0: 94 C5 61 93-4F 4E 2C 9F-0D 54 0D E7-36 0B BF C5 ò+adON. fPT Pr667+
000000C0: 21 03 D8 AB-A6 DC 19 83-CC B2 EA 46-02 F3 50 8C !+Pz = lã|lllRF0P i
000000D0: BB 32 C3 6F-5F 56 2F DC-69 A8 81 00-AC 46 17 44 72 to U/ m icü xP dD
000000E0: A8 C3 B1 A8-5C AB 39 10-DA 1A 5E B8-FE 23 0F C7 z|lll\>9P r-^3 #x|
000000F0: BC B6 77 EC-62 AE F8 76-22 70 6C B9-09 B5 0B 3E 4|lll00<0u "p1|0f<0>
00000100: C3 F9 B6 BD-45 DA CE 3E-F8 3B 3E 3D-0A 5A 99 63 卜-||UE r'0; >=0z0c
00000110: 33 BE 43 3F-F1 DC 3F B9-6A 07 E8 0D-2C BD D7 53 3<C?± m'z|j>0P. 4|lS
00000120: 19 71 53 80-15 8D 0C 3C-4A 4E BA F5-03 1C 3C C8 ↓qS0S iP<JN||j|w<L
00000130: 71 E5 2B 1F-1E 3A 06 DD-E3 B4 69 F5-D5 CA 5E 92 qσ+VA: 4|ll i j ll'fe
00000140: 84 5D 09 8A-D9 E0 65 26-96 76 D5 49-C6 28 B1 8B ä l0è-J ae8üv fI f0i
00000150: 83 7B 04 98-B4 55 92 7A-81 87 05 94-9D A6 3C 7C ä<0?Ullfeziic 88P<I
00000160: 58 10 FE 7D-B2 D4 47 D9-34 5A 72 6B-93 F8 1F AA X|P|lllEG-4Zr:k0 0v-
00000170: 50 B8 C5 8B-C6 62 5E 1C-E6 45 EA 2D-BA FD D5 1F Pz+i fb'~j dE0-||z fV
00000180: DD C0 90 F4-57 16 E0 2C-13 13 35 10-B5 37 AC B0 |lE Pwα, !!!5=7%
00000190: 7E 24 A4 F7-84 80 54 F0-12 AE 3A F4-3B 07 5B EE ~n88CTE tcc: P; •LE
000001A0: 09 5D 7A 19-EC 96 D0 20-1B 80 9F B1-6F E6 70 8E o lz loüll +cJf0 jupã
000001B0: EC 8C CA 98-8E 59 7D 6D-57 FB 44 46-22 99 94 28 ωi±jãY)mM'DF"00<
000001C0: D8 C3 3E 4E-1A 2B C9 B3-EC 1E DA 7C-0C 34 66 E8 ±|>N+>r| ωΔ r|P4f 0
000001D0: 0A 80 5A 78-52 3E E5 17-C1 1D 9F E0-61 68 36 20 0CZxR>σ±+foah6
000001E0: 72 CC A5 60-01 FD 55 B9-77 7D 7A FD-D8 C1 05 65 r|l'ñ| 0?U|llw> z'±ll'ae
000001F0: 6A 7B 6C DA-FE 9B E7 DD-02 30 C4 6C-FE E4 71 44 j<ll rll'c r| 00-1lΣqD
00000200: 91 3D F3 FC-65 AF 4E E9-30 37 3C FF-B3 BE D9 5C ae=±"e>N007< |J'
00000210: 4C 1A 83 60-1F 0B A3 6B-6F 96 4A 48-F2 7D 71 0A L->â V6ÜkoüJH2>q0
00000220: 1E 18 D9 43-8E 24 0D EB-56 9A FC 16-20 79 87 A4 Δ↑Cã$ P0Uj" = ycn
00000230: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000240: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000250: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000260: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000270: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000280: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000290: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000300: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000310: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000320: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000330: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000340: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000350: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000360: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000370: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000380: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000390: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000400: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000410: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000420: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000430: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000440: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000450: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000460: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000470: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000480: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000490: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
00000500: 7A C8 A5 FB-C8 29 2B 48-4B 9E DE 14-0B 7 10 F9 qL60L>+HKR. 7P29->
```

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\x'
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+40f0
.raimo:0133E020 enc_conf_hash dd 0B6C2E135h ; DATA XREF: sub_1321B1C+17f0
.raimo:0133E024 ; SIZE_T enc_conf_size
.raimo:0133E024 enc_conf_size dd 6B02h ; DATA XREF: sub_1321B1C+1f0
.raimo:0133E024 ; sub_1321B1C+24f0 ...
.raimo:0133E028 ; char enc_config data[]
.raimo:0133E028 enc_config_data db 8,'H-ĒFKæ"qŪH@',15h,'PFI,i}æÄ²ZàèµÒ:T|',0Fh,'fY-M',17h,8Fh,'$',0,'Ū'
.raimo:0133E028 ; DATA XREF: sub_1321B1C+7f0
.raimo:0133E028 db 13h,'||',16h,'èì',15h,'`æäiæ',27h,0Dh,'_kâ',17h,'*...Ū',1Eh,'³0',6,'è'
.raimo:0133E028 db 'Ō•',90h,'T',3,'z',90h,'Đ',17h,'+š',18h,'+İ',14h,'Ÿ_Æ•ZÆ',16h,'+²ç'
.raimo:0133E028 db 8Fh,'zi0/çç...5Šù}ã Encrypted ConfigurationFh,'~r&i',8Fh,'<vý8S¼æ'
.raimo:0133E028 db 'A²',1,';öi',1,'ámqñçēw+T"=U#÷V=0æ~ij',1Eh,18h,90h,1Eh,'Ō',9,0Eh,'-'
.raimo:0133E028 db 'xŸ«i`ž,Đ@ša',6,'L0',27h,'A_šš',7,'7<v',8Dh,'¼çf',0Ah
.raimo:0133E028 db '^j',1Bh,'āwß;d?_Ū¥ ßa}Jx',8Fh,'v',1Bh,11h,1Ah,'«AŪ#NÁ+',8Fh,'+Y<E'
.raimo:0133E028 db '?Ōüžl]š',1Eh,'"÷[7üYyĀ',16h,'jñē-æš[€',16h,'<',12h,'e',6,'V"İPæÉŸ'
.raimo:0133E028 db 'f~ð',3,90h,'°"axŸ*xFk3"Rz',9,'ĒHV&y',2,'+tbBµ^Ē&0æh'(„ñ-XİĒĒiŌD²ð;
.raimo:0133E028 db '₌₌₌₌' 13h,'üä01ó',27h,'%' 7Fh,'/' 1Fh,'r=:>āİĐW' 0Ah

```

The resulting decrypted configuration file looks like this:

```

1 {
2   "pk": "/SCIwBFa1lAe0gTv32PoxS2LkBk3Bs6aDnd5JExyG3g=",
3   "pid": "26",
4   "sub": "2095",
5   "dbg": false,
6   "fast": false,
7   "wipe": true,
8   "wht": {
9     "fld": [
10      ],
11     "fls": [
12      ],
13     "ext": [
14      ]
15    ],
16   "wfld": [
17   ],
18   "prc": [
19   ],
20   "dmn": "mgimalta.com;makingmillionaires.net;walterman.es;eurethicsport.eu;smartspeak.com;ar
21   "net": false,
22   "svc": [
23   ],
24   "nbody": "LQAtAC0AIAA9AD0APQAgAFcAZQBzAGMAbwBtAGUAIABDAHkAcgB1AHMATwBuAGUAIABhAG4AZAAgAEQAZ
25   "nname": "{EXT}-readme.txt",
26   "exp": false,
27   "img": "QQBzAGwAIABvAGYAIAB5AG8AdQBvACAAZgBpAGwAZQBzACAAYQByAGUAIABLAG4AYwByAHkAcAB0AGUAZAA
28   "arn": false
29 }

```

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!