A Quick Solution to an Ugly Reverse Engineering Problem

msreverseengineering.com/blog/2019/1/14/a-quick-solution-to-an-ugly-reverse-engineering-problem

January 14, 2019

January 14, 2019 Rolf Rolles

Reverse engineering tools tend to be developed against fundamental assumptions, for example, that binaries will more or less conform to the standard patterns generated by compilers; that instructions will not jump into other instructions; perhaps that symbols are available, etc. As any reverse engineer knows, your day can get worse if the assumptions are violated. Your tools may work worse than usual, or even stop working entirely. This blog post is about one such minor irritation, and the cheap workaround that I used to fix it.

In particular, the binary I was analyzing -- one function in particular -- made an uncommon use of an ordinary malware subterfuge technique, which wound up violating ordinary assumptions about the sizes of functions. In particular, malware authors quite often build data that they need -- strings, most commonly -- in a dynamic fashion, so as to obscure the data from analysts using tools such as "strings" or a hex editor. (Malware also commonly enciphers its strings somehow, though that is not the feature that I'll focus on in this entry.) As such, we see a lot of the following in the function in question.

```
.text:004048C7 C6 85 FC E5 FF FF 4D      mov     [ebp+var_1A04], 4Dh ; 'M'
.text:004048CE C6 85 FD E5 FF FF 5A      mov     [ebp+var_1A03], 5Ah ; 'Z'
.text:004048D5 C6 85 FE E5 FF FF 90      mov     [ebp+var_1A02], 90h
.text:004048DC C6 85 FF E5 FF FF 00      mov     [ebp+var_1A01], 0
.text:004048E3 C6 85 00 E6 FF FF 03      mov     [ebp+var_1A00], 3
.text:004048EA C6 85 01 E6 FF FF 00      mov     [ebp+var_19FF], 0
.text:004048F1 C6 85 02 E6 FF FF 00      mov     [ebp+var_19FE], 0
.text:004048F8 C6 85 03 E6 FF FF 00      mov     [ebp+var_19FD], 0
.text:004048FF C6 85 04 E6 FF FF 04      mov     [ebp+var_19FC], 4
.text:00404906 C6 85 05 E6 FF FF 00      mov     [ebp+var_19FB], 0
.text:0040490D C6 85 06 E6 FF FF 00      mov     [ebp+var_19FA], 0
.text:00404914 C6 85 07 E6 FF FF 00      mov     [ebp+var_19F9], 0
.text:0040491B C6 85 08 E6 FF FF FF      mov     [ebp+var_19F8], 0FFh
.text:00404922 C6 85 09 E6 FF FF FF      mov     [ebp+var_19F7], 0FFh
.text:00404929 C6 85 0A E6 FF FF 00      mov     [ebp+var_19F6], 0
.text:00404930 C6 85 0B E6 FF FF 00      mov     [ebp+var_19F5], 0
.text:00404937 C6 85 0C E6 FF FF B8      mov     [ebp+var_19F4], 0B8h ; '¸'
```

What made this binary's use of the technique unusual was the scale at which it was applied. Typically the technique is used to obscure strings, usually no more than a few tens of bytes apiece. This binary, on the other hand, used the technique to build two embedded executables, totaling about 16kb in data -- hence, there are about 16,000 writes like the one in the previous figure, each implemented by a 7-byte instruction. The function pictured above comprises about 118KB of code -- over 25% of the total size of the binary. The function would have been large even without this extra subterfuge, as it has about 7kb of compiled code apart from the instructions above.

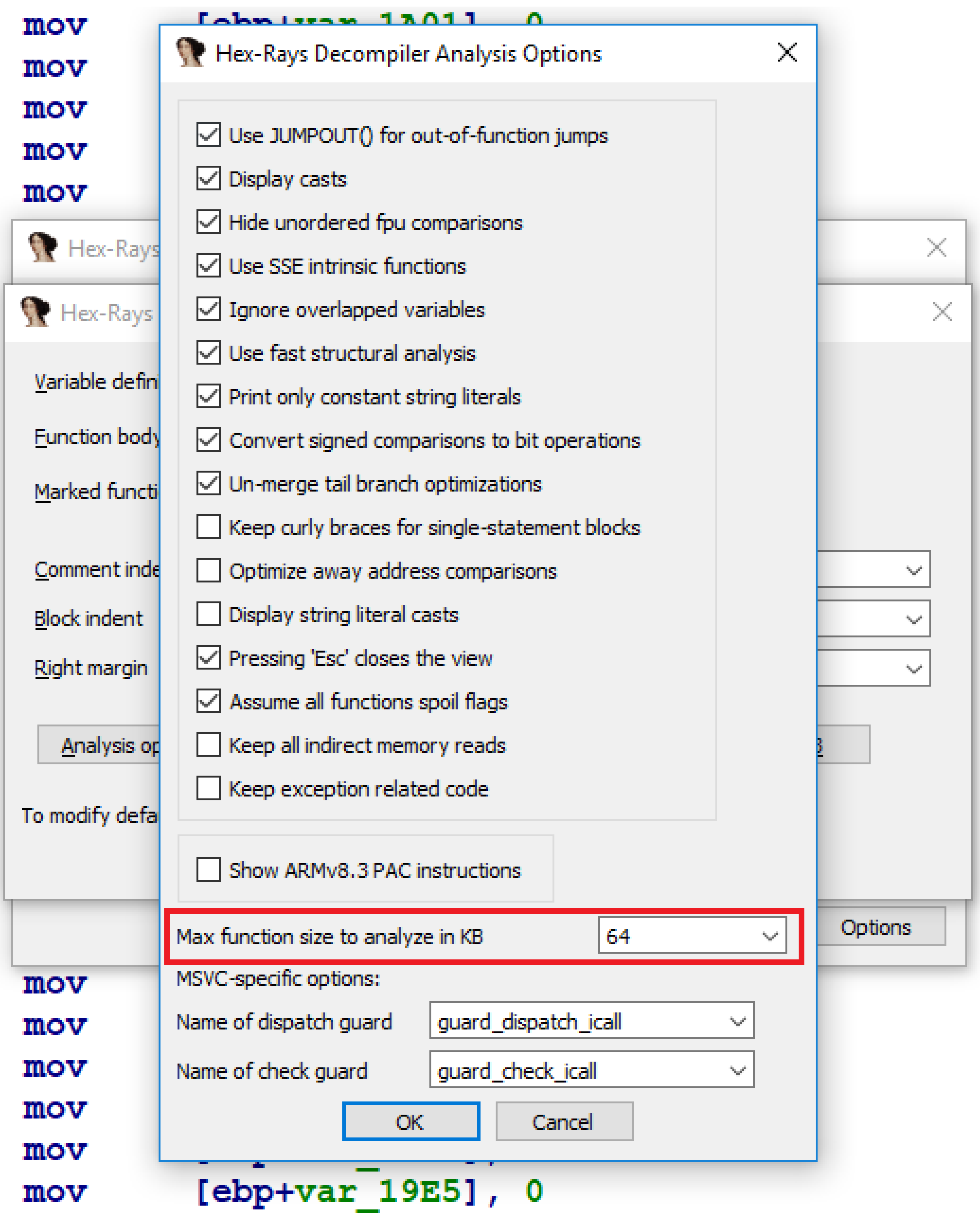
The Hex-Rays decompilation for this function is about 32,500 lines. The bulk of this comes from two sources: first, the declaration of one stack local variable per written stack byte:

```
HANDLE v103; // [esp+220h] [ebp-3EF8h]
HANDLE v104; // [esp+224h] [ebp-3EF4h]
char v105; // [esp+228h] [ebp-3EF0h]
char String1[64]; // [esp+268h] [ebp-3EB0h]
int v107; // [esp+2A8h] [ebp-3E70h]
char v108; // [esp+2CCh] [ebp-3E4Ch]
char v109; // [esp+2F0h] [ebp-3E28h]
char v110; // [esp+314h] [ebp-3E04h]
char v111; // [esp+315h] [ebp-3E03h]
char v112; // [esp+316h] [ebp-3E02h]
char v113; // [esp+317h] [ebp-3E01h]
char v114; // [esp+318h] [ebp-3E00h]
char v115; // [esp+319h] [ebp-3DFFh]
char v116; // [esp+31Ah] [ebp-3DFEh]
char v117; // [esp+31Bh] [ebp-3DFDh]
char v118; // [esp+31Ch] [ebp-3DFCh]
char v119; // [esp+31Dh] [ebp-3DFBh]
```

Second, one assignment statement per write to a stack variable:

```
v9326 = 77;
v9327 = 90;
v9328 = -112;
v9329 = 0;
v9330 = 3;
v9331 = 0;
v9332 = 0;
v9333 = 0;
v9334 = 4;
v9335 = 0;
v9336 = 0;
v9337 = 0;
v9338 = -1;
v9339 = -1;
v9340 = 0;
v9341 = 0;
v9342 = -72;
v9343 = 0;
v9344 = 0;
v9345 = 0;
```

To IDA's credit, it handles this function just fine; there is no noticeable slowdown in using IDA to analyze this function. Hex-Rays, however, has a harder time with it. (I don't necessarily blame Hex-Rays for this; the function is 118KB, after all, and Hex-Rays has much more work to do than IDA does in dealing with it.) First, I had to alter the Hex-Rays decompiler options in order to even decompile the function at all:



After making this change, Hex-Rays was very slow in processing the function, maxing out one of my CPU cores for about five minutes every time I wound up decompiling it. This is suboptimal for several reasons:

- I often use the File->Produce file->Create .c file... menu command more than once while reverse engineering a particular binary. This function turns every such command into a cigarette break.
- Some plugins, such as Referee, are best used in conjunction with the command just mentioned.
- When using the decompiler on this function in an interactive fashion (such as by renaming variables or adding comments), the UI becomes slow and unresponsive.

- Randomly looking at the cross-references to or from a given function becomes a game of Russian Roulette instead of a normally snappy and breezy part of my reverse engineering processes. Decompile the wrong function and you end up having to wait for the decompiler to finish.

Thus, it was clear that it was worth 15 minutes of my time to solve this problem. Clearly, the slowdowns all resulted from the presence of these 16,000 write instructions. I decided to simply get rid of them, with the following high-level plan:

- Extract the two .bin files written onto the stack by the corresponding 112KB of compiled code
- Patch those .bin files into the database
- Replace the 112KB worth of instructions with one patched call to memcpy()
- Patch the function's code to branch over the 112KB worth of stack writes

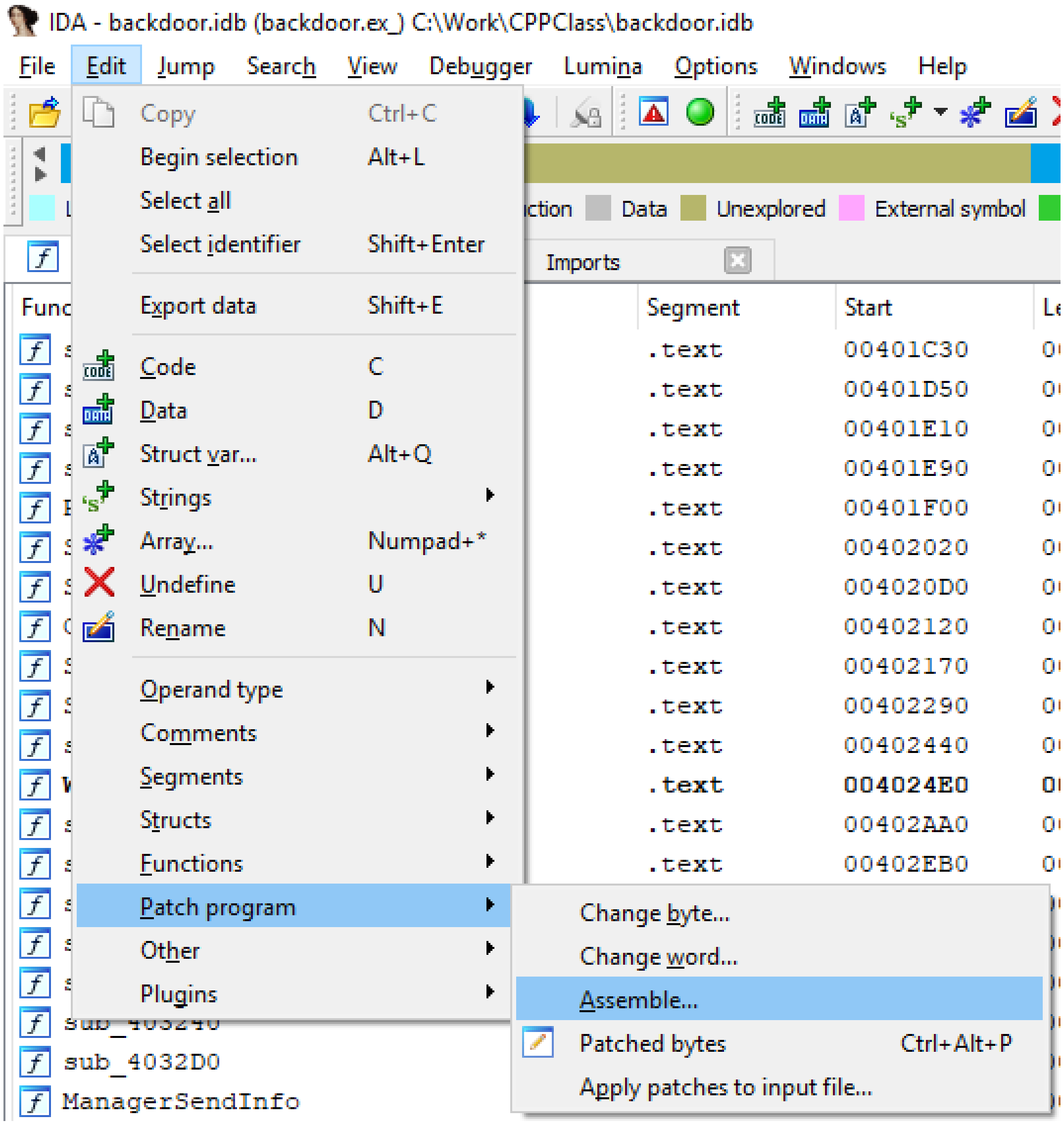
The first thing I did was copy and paste the Hex-Rays decompilation of the stack writes into its own text file. After a few quick sanity checks to make sure all the writes took place in order, I used a few regular expression search-and-replace operations and a tiny bit of manual editing to clean the data up into a format that I could use in Python.

```
data1 = [
77,
90,
-112,
0,
3,
0,
0,
0,
4,
0,
```

Next, a few more lines of Python to save the data as a binary file:

```
data2 = map(lambda x: x & 0xFF, data1)
newFile = open("data6655.bin", 'wb')
newFile.write(bytearray(data2))
```

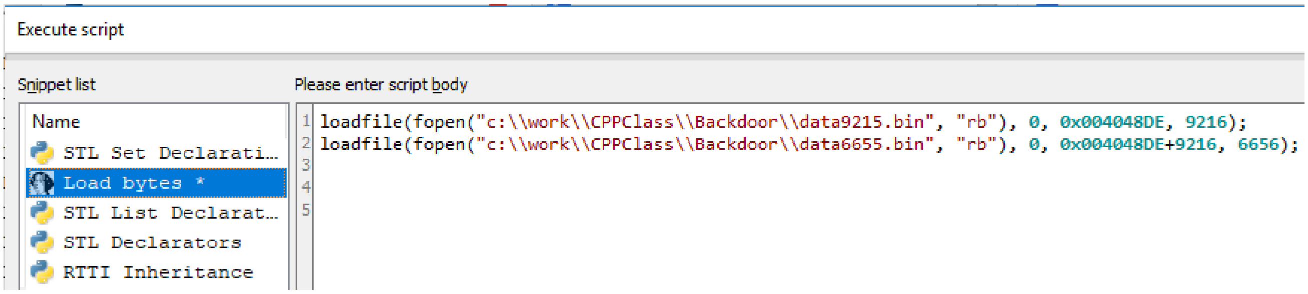
From there, I used IDA's Edit->Patch program->Assemble... command to write a small patch into the corresponding function:



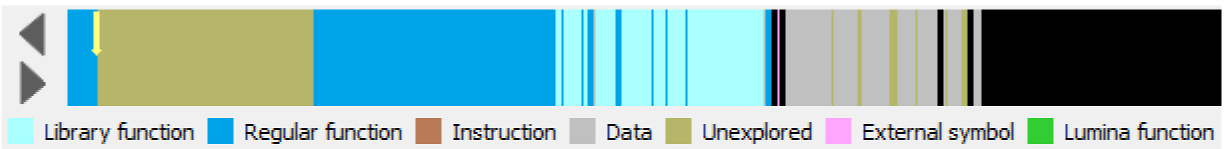
After a bit of fiddling and manual hex-editing the results, my patch was installed:

```
.text:004048BA call    sub_432E90
.text:004048BF test    eax, eax
.text:004048C1 jz      loc_41FBAE
.text:004048C7 lea     edi, [ebp+var_3E04]         ; PATCH
.text:004048CD mov     ecx, 3E00h                  ; PATCH
.text:004048D2 mov     esi, offset recovered_data  ; PATCH
.text:004048D7 rep movsb                           ; PATCH
.text:004048D9 jmp     after_writes                ; PATCH
.text:004048D9 NetworkCallbackThread endp
.text:004048D9
.text:004048D9 ; ---------------------------------------------------------------------------
.text:004048DE recovered_data db  4Dh ; M          ; DATA XREF: NetworkCallbackThread+E52↑o
.text:004048DF db  5Ah ; Z
.text:004048E0 db  90h
.text:004048E1 db    0
.text:004048E2 db    3
```

And then I used a two-line IDC script to load the binary files as data in the proper location:



Afterwards, the navigation bar showed that about 31% of the text section had been converted into data:



And now the problem is fixed. The function takes approximately two seconds to decompile, more in line with what we'd expect for a 7kb function. Hooray; no more endless waiting, all for the time cost of about three accidental decompilations of this function.

This example shows that, if you know your tools well enough to know what causes them problems, that sometimes you can work your way around them. Always stay curious, experiment, and don't simply settle for a suboptimal reverse engineering experience without exploring whether there might be an easier solution.