A Walk-Through Tutorial, with Code, on Statically Unpacking the FinSpy VM: Part One, x86 Deobfuscation — Möbius Strip Reverse Engineering

By Rolf Rolles

Published: 2018-01-23 · Archived: 2026-04-05 17:29:52 UTC

1. Introduction

Normally when I publish about breaking virtual machine software protections, I do so to present new techniques. Past examples have included:

- Writing an IDA processor module to unpack a VM
- Logging VM execution with DLL injection
- Compiler-based techniques to unpack commercial-grade VMs
- Abstract interpretation-based techniques to deobfuscate control flow
- Automated generation of peephole superdeobfuscators
- Program synthesis-based deobfuscation of metamorphic behavior in VM handlers

Today's document has a different focus. I am not going to be showcasing any particularly new techniques. I will, instead, be providing a step-by-step walk-through of the process I used to analyze the FinSpy VM, including my thoughts along the way, the procedures and source code I used, and summaries of the notes I took. The interested reader is encouraged to obtain the sample and walk through the analysis process for themselves.

I have three motives in publishing this document:

1. I think it's in the best interest of the security defense community if every malware analyst is able to unpack the FinSpy malware VM whenever they encounter it (for obvious reasons).
2. Reverse engineering is suffering from a drought of hands-on tutorial material in modern times. I was fortunate to begin reverse engineering when such tutorials were common, and they were invaluable in helping me learn the craft. Slides are fine for large analyses, but for smaller ones, let's bring back tutorials for the sake of those that have followed us.
3. Publications on obfuscation, especially virtualization obfuscation, have become extremely abstruse particularly in the past five years. Many of these publications are largely inaccessible to those not well-versed in master's degree-level program analysis (or above). I want to demonstrate that easier techniques can still produce surprisingly fast and useful results for some contemporary obfuscation techniques. (If you want to learn more about program analysis-based approaches to deobfuscation, there is currently a public offering of my SMT-based program analysis training class, which has over 200 slides on modern deobfuscation with working, well-documented code.)

Update: the rest of this document, the second and third parts, are now available online at the links just given.

2. Initial Steps

The first thing I did upon learning that a new FinSpy sample with VM was publicly available was, of course, to obtain the sample. VirusTotal gave the SHA256 hash; and I obtained the corresponding sample from Hybrid-Analysis.

The next step was to load the sample into IDA. The navigation bar immediately tipped me off that the binary was obfuscated:

- The first half of the .text section is mostly colored grey and red, indicating data and non-function code respectively.
- The second half of the .text section is grey in the navigation bar, indicating data turned into arrays.

A normal binary would have a .text section that was mostly blue, indicating code within functions.

3. Analysis of WinMain: Suspicions of VM-Based Obfuscation

IDA's auto-analysis feature identified that the binary was compiled by the Microsoft Visual C compiler. I began by identifying the WinMain function. Normally IDA would do this on my behalf, but the code at that location is obfuscated, so IDA did not name it or turn it into a function. I located WinMain by examining the ___tmainCRTStartup function from the Visual C Run-Time and finding where it called into user-written code. The first few instructions resembled a normal function prologue; from there, the obfuscation immediately began.

```
.text:00406154     mov     edi, edi                ; Normal prologue
.text:00406156     push    ebp                     ; Normal prologue
.text:00406157     mov     ebp, esp                ; Normal prologue
.text:00406159     sub     esp, 0C94h              ; Normal prologue
.text:0040615F     push    ebx                     ; Save registers #1
.text:00406160     push    esi                     ; Save registers #1
.text:00406161     push    edi                     ; Save registers #1
.text:00406162     push    edi                     ; Save registers #2
.text:00406163     push    edx                     ; Save registers #2
.text:00406164     mov     edx, offset byte_415E41 ; Obfuscation - #1
.text:00406169     and     edi, 0C946B9C3h         ; Obfuscation - #2
.text:0040616F     sub     edi, [edx+184h]         ; Obfuscation - #3
.text:00406175     imul    edi, esp, 721D31h       ; Obfuscation - #4
.text:0040617B     stc                             ; Obfuscation
.text:0040617C     sub     edi, [edx+0EEh]         ; Obfuscation - #5
.text:00406182     shl     edi, cl                 ; Obfuscation
.text:00406184     sub     edi, [edx+39h]          ; Obfuscation - #6
.text:0040618A     shl     edi, cl                 ; Obfuscation
.text:0040618C     imul    edi, ebp                ; Obfuscation
.text:0040618F     mov     edi, edi                ; Obfuscation
.text:00406191     stc                             ; Obfuscation
.text:00406192     sub     edi, 0A14686D0h         ; Obfuscation
```

```
; ... obfuscation continues ...

.text:004065A2     pop     edx                     ; Restore registers
.text:004065A3     pop     edi                     ; Restore registers
```

The obfuscation in the sequence above continues for several hundred instructions, nearly all of them consisting of random-looking modifications to the EDI register. I wanted to know A) whether the computations upon EDI were entirely immaterial junk instructions, or whether a real value was being produced by this sequence, and B) whether the memory references in the lines labeled #1, #3, #5, and #6 were meaningful.

As for the first question, note that the values of the registers upon entering this sequence are unknown. We are, after all, in WinMain(), which uses the __cdecl calling convention, meaning that the caller did not pass arguments in registers. Therefore, the value computed on line #2 is unpredictable and can potentially change across different executions. Also, the value computed on line #4 is pure gibberish -- the value of the stack pointer will change across runs (and the modification to EDI overwrites the values computed on lines #1-#3).

As for the second question, I skimmed the obfuscated listing and noticed that there were no writes to memory, only reads, all intertwined with gibberish instructions like the ones just described. Finally, the original value of edi is popped off the stack at the location near the end labeled "restore registers". So I was fairly confident that I was looking at a sequence of instructions meant to do nothing, producing no meaningful change to the state of the program.

Following that was a short sequence:

```
.text:004065A4     push    5A403Dh                 ; Obfuscation
.text:004065A9     push    ecx                     ; Obfuscation
.text:004065AA     sub     ecx, ecx                ; Obfuscation
.text:004065AC     pop     ecx                     ; Obfuscation
.text:004065AD     jz      loc_401950              ; Transfer control elsewhere
.text:004065AD ; ---------------------------------------------------------------------------
.text:004065B3     db 5 dup(0CCh)
.text:004065B8 ; ---------------------------------------------------------------------------
.text:004065B8     mov     edi, edi
.text:004065BA     push    ebp
.text:004065BB     mov     ebp, esp
.text:004065BD     sub     esp, 18h

; ... followed by similar obfuscation to what we saw above ...
```

By inspection, this sequence just pushes the value 5A403Dh onto the stack, and transfers control to loc_401950. (The "sub ecx, ecx" instruction above sets the zero flag to 1, therefore the JZ instruction will always branch.)

Next we see the directive "db 5 dup(0CCh)" followed by "mov edi, edi". Reverse engineers will recognize these sequences as the Microsoft Visual C compiler's implementation of hot-patching support. The details of hot-patching are less important than the observation that I expected that the original pre-obfuscated binary contained a

function that began at the address of the first sequence, and ended before the "db 5 dup(0CCh)" sequence. I.e. I expect that the obfuscator disassembled all of the code within this function, replaced it with gibberish instructions, placed a branch at the end to some other location, and then did the same thing with the next function.

This is a good sign that we're dealing with a virtualization-based obfuscator: namely, it looks like the binary was compiled with an ordinary compiler, then passed to a component that overwrote the original instructions (rather than merely encrypting them in-place, as would normal packers).

4. Learning More About the VM Entrypoint and VM Pre-Entry

Recall again the second sequence of assembly code from the previous sequence:

```
.text:004065A4     push    5A403Dh                 ; Obfuscation - #1
.text:004065A9     push    ecx                     ; Obfuscation
.text:004065AA     sub     ecx, ecx                ; Obfuscation
.text:004065AC     pop     ecx                     ; Obfuscation
.text:004065AD     jz      loc_401950              ; Transfer control elsewhere
```

Since -- by supposition -- all of the code from this function was replaced with gibberish, there wasn't much to meaningfully analyze. My only real option was to examine the code at the location loc_401950, the target of the JZ instruction on the last line. The first thing I noticed at this location, loc_401950, was that there were 125 incoming references, nearly all of them of the form "jz loc_401950", with some of the form "jmp loc_401950". Having analyzed a number of VM-based obfuscators in my day, this location fits the pattern of being the part of the VM known as the "entrypoint" -- the part where the virtual CPU begins to execute. Usually this location will save the registers and flags onto the stack, before performing any necessary setup, and finally beginning to execute VM instructions. VM entrypoints usually require a pointer or other identifier to the bytecode that will be executed by the VM; maybe that's the value from the instruction labeled #1 in the sequence above? Let's check another incoming reference to that location to verify:

```
.text:00408AB8     push    5A7440h ; #2
.text:00408ABD     push    eax
.text:00408ABE     sub     eax, eax
.text:00408AC0     pop     eax
.text:00408AC1     jz      loc_401950
```

The other location leading to the entrypoint is functionally identical, apart from pushing a different value onto the stack. This value is not a pointer; it does not correspond to an address within the executable's memory image. Nevertheless, we expect that this value is somehow responsible for telling the VM entrypoint where the bytecode is located.

5. Analyzing the VM Entrypoint Code

So far we have determined that loc_401950 is the VM entrypoint, targeted by 125 branching locations within the binary, which each push a different non-pointer DWORD before branching. Let's start analyzing that code:

```
.text:00401950                  loc_401950:
.text:00401950 0F 82 D1 02 00 00    jb      loc_401C27
.text:00401956 0F 83 CB 02 00 00    jnb     loc_401C27
```

Immediately we see an obvious and well-known form of obfuscation. The first line jumps to loc_401C27 if the "below" conditional is true, and the second line jumps to loc_401C27 if the "not below" conditional is true. I.e., execution will reach loc_401C27 if either "below" or "not below" is true in the current EFLAGS context. I.e., these two instructions will transfer control to loc_401C27 no matter what is in EFLAGS -- and in particular, we might as well replace these two instructions with "jmp loc_401C27", as the effect would be identical.

Continuing to analyze at loc_401C27, we see another instance of the same basic idea:

```
.text:00401C27                  loc_401C27:
.text:00401C27 77 CD                ja      short loc_401BF6
.text:00401C29 76 CB                jbe     short loc_401BF6
```

Here we have an unconditional branch to loc_401BF6, split across two instructions -- a "jump if above", and "jump if below or equals", where "above" and "below or equals" are logically opposite and mutually exclusive conditions.

After this, at location loc_401BF6, there is a legitimate-looking instruction (push eax), followed by another conditional jump pair to loc_401D5C. At that location, there is another legitimate-looking instruction (push ecx), followed by a conditional jump pair to loc_4019D2. At that location, there is another legitimate-looking instruction (push edx), followed by another conditional jump pair. It quickly became obvious that every legitimate instruction was interspersed between one or two conditional jump pairs -- there are hundreds or thousands of these pairs throughout the binary.

Though an extremely old and not particularly sophisticated form of obfuscation, it is nevertheless annoying and degrades the utility of one's disassembler. As I discussed in a previous entry on IDA processor module extensions, IDA does not automatically recognize that two opposite conditional branches to the same location are an unconditional branch to that location. As a result, IDA thinks that the address following the second conditional branch must necessarily contain code. Obfuscation authors exploit this by putting junk bytes after the second conditional branch, which then causes the disassembler to generate garbage instructions, which may overlap and occlude legitimate instructions following the branch due to the variable-length encoding scheme for X86. (Note that IDA is not to blame for this conundrum -- ultimately these problems are undecidable under ordinary Von Neumann-based models of program execution.) The result is that many of the legitimate instructions get lost in the dreck generated by this process, and that, in order to follow the code as usual in manual static analysis, one would spend a lot of time manually undefining the gibberish instructions and re-defining the legitimate ones.

6. Deobfuscating the Conditional Branch Obfuscation: Theory and Practice

Manually undefining and redefining instructions as just described, however, would be a waste of time, so let's not do that. Speaking of IDA processor modules, once it became clear that this pattern repeated between every

legitimate non-control-flow instruction, I got the idea to write an IDA processor module extension to remove the obfuscation automatically. IDA processor module extensions give us the ability to have a function of ours called every time the disassembler encounters an instruction. If we could recognize that the instruction we were disassembling was a conditional branch, and determine that the following instruction contains its opposite conditional branch to the same target as the first, we could replace the first one with an unconditional branch and NOP out the second branch instruction.

Thus, the first task is to come up with a way to recognize instances of this obfuscation. It seemed like the easiest way would be to do this with byte pattern-recognition. In my callback function that executes before an instruction is disassembled, I can inspect the raw bytes to determine whether I'm dealing with a conditional branch, and if so, what the condition is and the branch target. Then I can apply the same logic to determine whether the following instruction is a conditional branch and determine its condition and target. If the conditions are opposite and the branch targets are the same, we've found an instance of the obfuscation and can neutralize it.

In practice, this is even easier than it sounds! Recall the first example from above, reproduced here for ease of reading:

```
.text:00401950 0F 82 D1 02 00 00     jb      loc_401C27
.text:00401956 0F 83 CB 02 00 00     jnb     loc_401C27
```

Each of these two instructions is six bytes long. They both begin with the byte 0F (the x86 two-byte escape opcode stem), are then followed by a byte in the range of 80 to 8F, and are then followed by a DWORD encoding the displacement from the end of the instructions to the branch targets. As a fortuitous quirk of x86 instruction encodings, opposite conditional branches are encoded with adjacent bytes. I.e. 82 represents the long form of JB, and 83 represents the long form of JNB. Two long branches have opposite condition codes if and only if their second opcode byte differs from one another in the lowest bit (i.e. 0x82 ^ 0x83 == 0x01). And note also that the DWORDs following the second opcode byte differ by exactly 6 -- the length of a long conditional branch instruction.

That's all we need to know for the long conditional branches. There is also a short form for conditionals, shown in the second example above and reproduced here for ease of reading:

```
.text:00401C27 77 CD                 ja      short loc_401BF6
.text:00401C29 76 CB                 jbe     short loc_401BF6
```

Virtually identical comments apply to these sequences. The first bytes of both instructions are in the range of 0x70 to 0x7F, opposite conditions have differing lowest bits, and the second bytes differ from one another by exactly 2 -- the length of a short conditional branch instruction.

7. Deobfuscating the Conditional Branch Obfuscation: Implementation

I started by copying and pasting my code from the last time I did something like this. I first deleted all the code that was specific to the last protection I broke with an IDA processor module extension. Since I've switched to

IDA 7.0 in the meantime, and since IDA 7.0 made breaking changes vis-a-vis prior APIs, I had to make a few modifications -- namely, renaming the custom analysis function from deobX86Hook::custom_ana(self) to deobX86Hook::ev_ana_insn(self, insn), and replacing every reference to idaapi.cmd.ea with insn.ea. Also, my previous example would only run if the binary's MD5 matched a particular sum, so I copied and pasted the sum of my sample out of IDA's database preamble over the previous MD5.

From there I had to change the logic in custom_ana. The result was even simpler than my last processor module extension. Here is the logic for recognizing and deobfuscating the short form of the conditional branch obfuscation:

```
b1 = idaapi.get_byte(insn.ea)
if b1 >= 0x70 and b1 <= 0x7F:
    d1 = idaapi.get_byte(insn.ea+1)
    b2 = idaapi.get_byte(insn.ea+2)
    d2 = idaapi.get_byte(insn.ea+3)
    if b2 == b1 ^ 0x01 and d1-2 == d2:
        # Replace first byte of first conditional with 0xEB, the opcode for "JMP rel8"
        idaapi.put_byte(insn.ea, 0xEB)
        # Replace the following instruction with two 0x90 NOP instructions
        idaapi.put_word(insn.ea+2, 0x9090)
```

Deobfuscating the long form is nearly identical; see the code for details.

8. Admiring My Handiwork, Cleaning up the Database a Bit

Now I copied the processor module extension to %IDA%\plugins and re-loaded the sample. It had worked! The VM entrypoint had been replaced with:

```
.text:00401950 loc_401950:
.text:00401950     jmp     loc_401C27
```

Though the navigation bar was still largely red and ugly, I immediately noticed a large function in the middle of the text section:

Looking at it in graph mode, we can see that it's kind of ugly and not entirely as nice as analyzing unobfuscated X86, but considering how trivial it was to get here, I'll take it over the obfuscated version any day. The red nodes denote errant instructions physically located above the valid ones in the white nodes. IDA's graphing algorithm includes any code within the physically contiguous region of a function's chunks in the graph display, regardless of whether they have incoming code cross-references, likely to make displays of exception handlers nicer. It would be easy enough to remove these and strip the JMP instructions if you wanted to write a plugin to do so.

Next I was curious about the grey areas in the .text section navigation bar held. (Those areas denote defined data items, mixed in with the obfuscated code in the .text section.) I figured that the data held there was most likely

related to the obfuscator. I spent a minute looking at the grey regions and found this immediately after the defined function:

```
.text:00402AE0     dd offset loc_402CF2
.text:00402AE4     dd offset loc_402FBE

; ... 30 similar lines deleted ...

.text:00402B60     dd offset loc_4042DC
.text:00402B64     dd offset loc_40434D
```

34 offsets, each of which contains code. Those are probably the VM instruction handlers. For good measure, let's turn those into functions with an IDAPython one-liner:

```
for pFuncEa in xrange(0x00402AE0, 0x00402B68, 4):
    idaapi.add_func(idaapi.get_long(pFuncEa))
```

Now a large, contiguous chunk of the navigation bar for the .text section is blue. And at this point I realized I had forgotten to create a function at the original dispatcher location, so I did that manually and here was the resulting navigation bar:

Hex-Rays doesn't do a very good job with any of the functions we just defined, since they were originally written in assembly language and use instructions and constructs not ordinarily produced by compilers. I don't blame Hex-Rays for that and I hope they continue to optimize for standard compiler-based use cases and not weird ones like this.

Lastly, I held PageDown scrolling through the text section to see what was left. The majority of it was VM entrypoints like those we saw in section 3. There were a few functions that appeared like they had been produced by a compiler.

So now we have assessed what's in the text section -- a VM with 34 handlers, 125+ virtualized functions, and a handful of unvirtualized ones. Next time we'll take a look at the VM.

9. Preview of Parts 2 and 3, and Beyond

After this I spent a few hours analyzing the VM entrypoint and VM instruction handlers. Next, through static analysis I obtained the bytecode for the VM program contained within this sample. I then wrote a disassembler for the VM. That's part two.

From there, by staring at the disassembled VM bytecode I was able to write a simple pattern-based deobfuscator. After that I re-generated the X86 machine code, which was not extremely difficult, but it was more laborious than I had originally anticipated. That's part three.

After that, I re-inserted the X86 machine code into the original binary and analyzed it. It turned out to be a fairly sophisticated dropper for one of two second-stage binaries. It was fairly heavy on system internals and had a few

tricks that aren't widely documented, so I may publish one or more of those as separate entries, and/or I may publish an analysis of the entire dropper.

Finally, I analyzed -- or rather, still am analyzing -- the second-stage binaries. They may or may not prove worthy of publication.
