FinSpy VM Unpacking Tutorial Part 3: Devirtualization. Phase #4: Second Attempt at Devirtualization — Möbius Strip Reverse Engineering

By Rolf Rolles

Published: 2018-02-21 · Archived: 2026-04-05 18:58:50 UTC

[Note: if you've been linked here without context, the introduction to Part #3 describing its four phases can be found here.]

1. Introduction

In Part #3, Phase #1, we deobfuscated the FinSpy VM bytecode program by removing the Group #2 instructions. In Part #3, Phase #2, we made a first attempt to devirtualize the FinSpy VM bytecode program back into x86 code. This was mostly successful, except for a few issues pertaining to functions and function calls, which we examined in Part #3, Phase #3.

Now we are ready to take a second stab at devirtualizing our FinSpy VM sample. We need to incorporate the information from Part #3, Phase #3 into our devirtualization of X86CALLOUT instructions. After having done so, we will take a second look at the devirtualized program to see whether any issues remain. After addressing one more major observation and a small one, our devirtualization will be complete.

2. Devirtualization, Take Two

We are finally ready to choose an address in the original FinSpy sample at which to insert the devirtualized code, devirtualize the FinSpy VM program, and copy the devirtualized machine code into the original binary. I chose the address 0x500000, for no particular reason other than that it was after any of the existing sections in the binary.

If everything we've done so far has worked correctly, now we have all of the information we need to generate proper functions in our devirtualized program. We have a set containing the non-virtualized functions called by the FinSpy VM program. For virtualized function targets, we have a list containing tuples of the function addresses, the VM instruction key corresponding to the first virtualized instruction in the function, and a list of prologue bytes to prepend before the devirtualization of the first virtualized instruction.

We derive two dictionaries from the virtualized function information.

1. The dictionary named X86_VMENTRY_TO_KEY_DICT maps an X86CALLOUT target to the VM instruction key corresponding to the beginning of the virtualized function body.
2. The dictionary named KEY_TO_PROLOGUE_BYTES_DICT maps the VM instruction key to the copied x86 prologue machine code bytes for the function beginning at the VM instruction with that key.

Now we make two changes to our instruction-by-instruction devirtualization process:

- In the loop that iterates over all VM bytecode instructions and produces the devirtualized output, consult KEY_TO_PROLOGUE_BYTES_DICT to see if the instruction corresponds to the beginning of a virtualized function. If so, insert the prologue bytes before devirtualizing the instruction.
- When devirtualizing X86CALLOUT instructions, look up the address of the target in the NOT_VIRTUALIZED set.
    - If the target is in the set, then nothing special needs to be done to devirtualize the X86CALLOUT instruction; emit an x86 CALL instruction with a dummy displacement DWORD of 0x0, and a fixup to later replace the 0x0 value with the proper distance from the source to the target (similarly to how we devirtualized VM jump instructions).
    - If the target is not in the set, then we need to generate an x86 CALL instruction to the devirtualized address of the target's VM instruction key. Emit a dummy x86 CALL instruction as before. Also generate a fixup specifying the offset of the dummy 0x0 displacement DWORD in the x86 CALL instruction, and the target of the X86CALLOUT instruction.

After the instruction-by-instruction devirtualization process, we need to process the fixups generated for the two varieties of X86CALLOUT instructions mentioned above (i.e., based on whether the destination is virtualized or not).

Here is partial code from the second approach at devirtualization.

```
# This is the same devirtualization function from before, but
# modified to devirtualize X86CALLOUT instructions and insert
# function prologues where applicable.
# It has a new argument: "newImageBase", the location in the
# FinSpy-virtualized binary at which we emit our devirtualized
# code.
def RebuildX86(insns, newImageBase):

    # These are the same as before:
    mcArr  = [] # Machine code array into which we generate code
    locsDict = dict() # VM location -> x86 position dictionary
    keysDict = dict() # VM key -> x86 position dictionary
    locFixups = []    # List of fixup locations for jumps

    # New: fixup locations for calls to virtualized functions
    keyFixups = []

    # New: fixup locations for calls to non-virtualized functions
    binaryRelativeFixups = []

    # Same as before: iterate over all instructions
    for i in insns:

        # Same as before: memorize VM position/key to x86 mapping
        currLen = len(mcArr)
```

```
        locsDict[i.Pos] = currLen
        keysDict[i.Key] = currLen

        # New: length of prologue instructions inserted before
        # devirtualized FinSpy VM instruction. Only obtains a
        # non-zero value if this instruction corresponds to the
        # beginning of a virtualized function.
        prologueLen = 0

        # New: is this VM instruction the beginning of a
        # virtualized function?
        if i.Key in KEY_TO_PROLOGUE_BYTES_DICT:

            # Get the prologue bytes that should be inserted
            # before this VM instruction.
            prologueBytes = KEY_TO_PROLOGUE_BYTES_DICT[i.Key]

            # Increase the length of the instruction.
            prologueLen += len(prologueBytes)

            # Copy the raw x86 machine code for the prologue
            # into the mcArr array before devirtualizing the
            # instruction.
            mcArr.extend(prologueBytes)

        # Now devirtualize the instruction. Handling of
        # "Raw X86", "Indirect Call", and jumps are identical
        # to before, so the code is not duplicated here.

        # Is this an "X86CALLOUT" ("Direct Call")?
        if isinstance(i,RawX86Callout):

            # New: emit 0xE8 (x86 CALL disp32)
            mcArr.append(0xE8)

            # Was the target a non-virtualized function?
            if i.X86Target in NOT_VIRTUALIZED:

                # Emit a fixup from to the raw target
                binaryRelativeFixups.append((i.Pos,prologueLen+1,i.X86Target))

            # Otherwise, the target was virtualized
            else:
                # Emit a fixup to the devirtualized function body
                # specified by the key of the destination
                keyFixups.append((i.Pos,prologueLen+1,i.X86Target))
```

```
            # Write the dummy destination DWORD in the x86 CALL
            # instruction that we just generated. This will be
            # fixed-up later.
            mcArr.extend([0x00, 0x00, 0x00, 0x00])
```

The Python code above generates additional fixup information for devirtualized X86CALLOUT instructions. The two cases of the destination being virtualized or not are handled similarly, though they are placed in two different lists ("keyFixups" for virtualized targets, and "binaryRelativeFixups" for non-virtualized targets). After the main devirtualization loop shown above, we must process the fixups just generated, the same way we did for the jump instructions. The process of applying the fixups is nearly identical to what we did for jump instructions, except that for virtualized targets, we need to determine the VM instruction key corresponding to the x86 address of the X86CALLOUT target. Here is the code for fixing up calls to virtualized functions:

```
# Fixups contain:
# * srcBegin: beginning of devirtualized CALL instruction
# * srcFixup: distance into devirtualized CALL instruction
#             where displacement DWORD is located
# * dst:      the X86CALLOUT target address
for srcBegin, srcFixup, dst in keyFixups:

    # Find the machine code address for the source
    mcSrc = locsDict[srcBegin]

    # Lookup the x86 address of the target in the information
    # we extracted for virtualized functions. Extract the key
    # given the function's starting address.
    klDst = X86_VMENTRY_TO_KEY_DICT[dst]

    # Find the machine code address for the destination
    mcDst = keysDict[klDst]

    # Set the displacement DWORD within x86 CALL instruction
    StoreDword(mcArr, mcSrc+srcFixup, mcDst-(mcSrc+srcFixup+4))
```

Next, and more simply, here is the code for fixing up calls to non-virtualized functions:

```
# Same comments as above
for srcBegin, srcFixup, dst in binaryRelativeFixups:

    # Find the machine code address for the source
    mcSrc = locsDict[srcBegin]

    # Compute the distance between the end of the x86
    # CALL instruction (at the address at which it will
    # be stored when inserted back into the binary) and
```

```
    # the raw x86 address of the X86CALLOUT target
    fixup = dst-(newImageBase+mcSrc+srcFixup+4)

    # Set the displacement DWORD within x86 CALL instruction
    StoreDword(mcArr, mcSrc+srcFixup, fixup)
```

3. Inspecting the Devirtualization

Now we are in a similar place to where we were after our initial devirtualization attempt in Part #3, Phase #2; let's look at the devirtualized code in IDA and see if anything jumps out as being obviously incorrect.

IDA's navigation bar shows a few things.

- The first third of the binary -- in the transparently-colored regions -- contains data defined as arrays. IDA has not identified code in these regions.
- The red-colored areas have properly been indentified as code, but don't have any incoming references, therefore they have not been defined as functions.

These two issues are related: if the regions currently marked as data are actually code, and if they make function calls to the code in red, then perhaps IDA can tell us that the red regions do have incoming references and should be defined as functions. I selected the entire text section, undefined it by pressing 'U', and then selected it again and pressed 'C' to turn it into code. The result was much more pleasing:

Now the whole devirtualized blob is defined as code. There is still an obvious cluster of functions that don't have incoming references.

Next we list the remaining issues that appear when inspecting the new devirtualization.

3.1 Some Functions Don't Have Incoming References

As we just saw from the navigation bar, there is a cluster of functions with no incoming references. Furthermore, inspecting these functions shows that they all lack prologues, like we noticed originally for all functions in our first devirtualization. If we turn them into functions, IDA makes its objections well-known with its black-on-red text:

So apparently, our prologue extraction scripts have missed these functions. We'll have to figure out why.

3.2 Many Call Instructions Have Invalid Destinations

IDA's "Problems" window (View->Open Subviews->Problems) helpfully points us to another category of errors. Many function calls have unresolved addresses, which IDA highlights as black-on-red text.

These issues have an innocuous explanation. In this Phase #4, we made the decision to choose the address 0x500000 as the base address at which to install the devirtualized code within the original binary. The x86 CALL instructions targeting non-virtualized functions are thus computed relative to an address in that region of the binary. Since we are currently inspecting the .bin file on its own, its base address is 0x0, and not 0x500000 like it

will be when we insert it the devirtualized code into IDA. The x86 CALL displacements are indeed nonsensical at the moment, but we'll double check on them after we've inserted the devirtualized code back into the binary.

3.3 One Call in Particular is Weird

All of the x86 CALL instructions described in the previous issue have displacements that begin with the nibbles 0xFFF....., indicating that the destination of those CALL instructions lies at an address located physically before the CALL instruction. However, one x86 CALL instruction at the beginning of a devirtualized function has a positive displacement, also colored black-on-red:

```
seg000:00000E70 sub_E70 proc near
seg000:00000E70     push    0B6Ch
seg000:00000E75     push    40B770h
seg000:00000E7A     call    near ptr 6D85h ; <- bogus destination
```

I looked at the corresponding function in the original binary from which this prologue had been copied, and the situation became clear.

```
.text:00404F77         push    0B6Ch
.text:00404F7C         push    offset stru_40B770
.text:00404F81         call    __SEH_prolog
.text:00404F86         xor     esi, esi
.text:00404F88         mov     [ebp-20h], esi
.text:00404F8B         push    edi        ; Save obfuscation register #1
.text:00404F8C         push    ebp        ; Save obfuscation register #1
.text:00404F8D         mov     ebp, offset word_411A6E ; Junk obfuscation
.text:00404F92         shrd    edi, esi, 0Eh           ; Junk obfuscation
```

The prologue for the pre-virtualized function installed an exception handler by calling __SEH_prolog. Our prologue extraction script simply copied the raw bytes for the prologue. Since x86 CALL instructions are encoded relative to their source and destination addresses, we can't simply copy a CALL instruction somewhere else without updating the destination DWORD; if we don't, the destination will be incorrect.

Since this issue appeared only once, instead of re-architecting the prologue extraction functionality to deal with this special case, I decided to just manually byte-patch my devirtualized code once I've copied it into the original binary. If I wanted to write a more fully-automated FinSpy devirtualization tool, I would tackle this issue more judiciously.

3.4 What are These Function Pointers?

The second devirtualization contains many pointers that reference hard-coded addresses within the original FinSpy binary from which we extracted the VM bytecode. For example, the following example references a function pointer and an address in the .text section:

```
seg000:00004BB9     push    0
seg000:00004BBE     push    0
seg000:00004BC3     push    400h
seg000:00004BC8     push    0FFFFh
seg000:00004BCD     call    dword ptr ds:401088h
seg000:00004BD3     mov     eax, ds:41FF38h
seg000:00004BD8     push    0
seg000:00004BDD     call    dword ptr [eax+38h]
```

Since we are examining the devirtualized code in isolation from the original binary, IDA cannot currently provide us meaningful information about the addresses in question. We can check the addresses in the FinSpy sample IDB to see if they make any sense; for example, here's the address referenced by the CALL:

```
.idata:00401088     ; LRESULT __stdcall SendMessageW(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lPara
.idata:00401088     extrn SendMessageW:dword
```

Things look good; we see four arguments pushed in the code above, and the function pointer references a function with four arguments. Once we've inserted our devirtualization back into the original binary, IDA will resolve the references seamlessly, and allow us to make full use of its normal facilities for cross-referencing, naming, type inference, and parameter tracking.

I also noticed that some of the instructions made reference to items within the original binary's .text section.

```
seg000:0000333E     mov     dword ptr [esi+4], 4055D5h
seg000:00003345     mov     dword ptr [esi], 40581Eh

; ... later ...

seg000:000034C4     mov     dword ptr [esi+4], 40593Ch
seg000:000034CB     mov     dword ptr [esi], 4055FEh

; ... later ...

seg000:000034DC     mov     dword ptr [esi+4], 40593Ch
seg000:000034E3     mov     dword ptr [esi], 405972h

; ... more like the above ...
```

Looking at these addresses in the original binary, I found that they corresponded to virtualized functions in the .text section. For example, here are the contents of the first pointer from the snippet above -- 0x4055D5 -- within the original binary:

```
.text:004055D5     mov     edi, edi                ; Original prologue
.text:004055D7     push    ebp                     ; Original prologue
```

```
.text:004055D8     mov     ebp, esp                ; Original prologue
.text:004055DA     push    ebp                     ; Push obfuscation register #1
.text:004055DB     push    esi                     ; Push obfuscation register #2
.text:004055DC     mov     esi, offset word_41CCBA ; Junk obfuscation
.text:004055E1     mov     ebp, 7C9E085h           ; Junk obfuscation
.text:004055E6     bswap   ebp                     ; Junk obfuscation
.text:004055E8     pop     esi                     ; Pop obfuscation register #2
.text:004055E9     pop     ebp                     ; Pop obfuscation register #1
.text:004055EA     push    5A329Bh                 ; Push VM instruction entry key
.text:004055EF     push    ecx                     ; Obfuscated JMP
.text:004055F0     sub     ecx, ecx                ; Obfuscated JMP
.text:004055F2     pop     ecx                     ; Obfuscated JMP
.text:004055F3     jz      GLOBAL__Dispatcher      ; Enter FinSpy VM
```

And it turns out that the VM key pushed by this sequence, namely 0x5A329B, references one of the functions in the devirtualized binary which otherwise did not have a incoming reference. Great! We would like to extract the addresses of the pointed-to functions so that we can process them with the scripts we developed in Part #3, Phase #3 in order to extract their prologues. We'd also like to alter the raw x86 instructions that reference the function pointers to make them point to their devirtualized targets within the devirtualized blob instead.

4. Next Step: Function Pointers

At this point, only two issues remain. First, we noticed that some devirtualized functions still don't have prologues. The explanation for this behavior must be that the addresses of their virtualized function stubs must not have been passed to the scripts. If we had provided those virtualized functions' addresses, our scripts would have found something for their prologues, even if it was an incorrect prologue. Yet the scripts found nothing.

Secondly, we noticed that the devirtualized code contained function pointers referencing the addresses of the prologue-lacking functions from the previous paragraph. We would like to replace the raw function pointers within the x86 instructions with the addresses of the corresponding functions in the devirtualized code.

4.1 Extracting Function Pointers from the VM Bytecode Disassembly

It seems like the first step in resolving both issues is to locate the function pointers within the FinSpy VM. I took a look at the raw FinSpy VM instructions from the snippet above with the function pointer references.

Here's the first VM instruction:

```
0x030630: X86 mov dword ptr [esi+4h], 4055D5h
```

Here's the raw bytes that encode that VM instruction:

```
seg000:00030630 dd 5A5C54h ; <- VM instruction key
seg000:00030634 db  1Bh    ; <- Opcode: raw x86
seg000:00030635 db    7    ; <- Length of x86 instruction: 7
```

```
seg000:00030636 db    3    ; <- Fixup offset: #3
seg000:00030637 db    0    ; <- Unused
seg000:00030638 db 0C7h    ; <- x86: mov dword ptr [esi+4], 55D5h
seg000:00030639 db  46h
seg000:0003063A db    4
seg000:0003063B db 0D5h    ; <- 3: offset of 0x55D5 in x86 instruction
seg000:0003063C db  55h
seg000:0003063D db    0
seg000:0003063E db    0
```

The important thing to notice is that the x86 machine code contained within this instruction disassembles to:

```
mov dword ptr [esi+4], 55D5h
```

Whereas the x86 instruction shown in the VM bytecode disassembly listing is, instead:

```
mov dword ptr [esi+4h], 4055D5h
```

The difference is in the DWORD value (55D5h in the raw VM bytecode versus 4055D5h in the VM bytecode disassembly).

The reason for this difference lies in the line labeled "Fixup offset: #3". You may recall from part two that all FinSpy VM instructions have two byte fields at offsets +6 and +7 into the VM instruction structure that were named "RVAPosition1" and "RVAPosition2". To quote the description of those fields from part two:

"Some instructions specify locations within the x86 binary. Since the binary's base address may change when loaded as a module by the operating system, these locations may need to be recomputed to reflect the new base address. FinSpy VM side-steps this issue by specifying the locations within the binary as relative virtual addresses (RVAs), and then adding the base address to the RVA to obtain the actual virtual addresses within the executing module. If either of [RVAPosition1 or RVAPosition2] is not zero, the FinSpy VM treats it as an index into the instruction's data area at which 32-bit RVAs are stored, and fixes the RVA up by adding the base address to the RVA."

In a bit of unplanned, happy serendipity, when I was writing my FinSpy VM bytecode disassembler, I made a Python class called "GenericInsn" that served as the base class for all other Python representations of FinSpy VM instruction types. Its Init() method is called in the constructor for every VM instruction type. And in particular, Init() includes the following code:

```
if self.Op1Fixup:
    ApplyFixup(self.Remainder, self.Op1Fixup & 0x7F, self.Pos)
if self.Op2Fixup:
    ApplyFixup(self.Remainder, self.Op2Fixup & 0x7F, self.Pos)
```

Thus, we are in the fortunate position where FinSpy VM helpfully tags all pointers to items in the original binary by setting these RVAPosition1 and RVAPosition2 fields. And furthermore, our existing function "ApplyFixup" already receives all of these values when we disassemble a FinSpy VM bytecode program. Thus, all we need to do to extract the function pointers is to include some logic inside of ApplyFixup that detects when one of these embedded RVAs refers to a function pointer, and if it does, to store the virtual address of the function pointer into a global set. The logic I used to determine function pointers was simply checking whether the virtual address was between the beginning of the first function in the .text section, and the last address in the .text section.

To wit, I changed my implementation of ApplyFixup as follows:

```
# New: constants describing the first function in the
# .text section and the end of the .text section.
TEXT_FUNCTION_BEGIN = 0x401340
TEXT_FUNCTION_END = 0x41EFC6

# New: a global dictionary whose keys are fixed-up
# virtual addresses, and whose values are lists of
# VM instruction positions whose bodies reference
# those virtual addresses.
from collections import defaultdict
ALL_FIXED_UP_DWORDS = defaultdict(list)

# Existing ApplyFixup function
def ApplyFixup(arr, FixupPos, InsnPos):
    # New: Python scoping statement
    global ALL_FIXED_UP_DWORDS

    # Existing ApplyFixup logic
    OriginalDword = ExtractDword(arr, FixupPos)
    FixedDword = OriginalDword + IMAGEBASE_FIXUP
    StoreDword(arr, FixupPos, FixedDword)

    # New: if the fixed-up DWORD is in the .text
    # section, save it in ALL_FIXED_UP_DWORDS and
    # add the VM instruction position (InsnPos) to
    # the list of positions referencing that DWORD.
    if FixedDword >= TEXT_FUNCTION_BEGIN and FixedDword <= TEXT_FUNCTION_END:
        ALL_FIXED_UP_DWORDS[FixedDword].append(InsnPos)
```

4.2 Extracting Prologues from Virtualized Function Pointers

Next, I also wanted to treat these virtual addresses as though they were the beginnings of virtualized functions, so that my existing machinery for extracting function prologues would incorporate them. In Part #3, Phase #3, I had written a function called "ExtractCalloutTargets" that scanned the FinSpy VM instructions looking for X86CALLOUT instructions and extracted their target addresses. This was then passed to the function prologue

extraction scripts to collect the data that was used in this Phase #4 to devirtualize X86CALLOUT instructions and insert the function prologues from virtualized functions into the devirtualization.

It seemed natural to modify ExtractCalloutTargets to incorporate the virtual addresses we collected in the previous subsection. To wit, I modified that function as such:

```
# Existing ExtractCalloutTargets function
def ExtractCalloutTargets(insns, vmEntrypoint):
    # New: Python scoping statement
    global ALL_FIXED_UP_DWORDS

    # New: initialize the set to the function pointer
    # addresses collected in ApplyFixup
    calloutTargets = set(ALL_FIXED_UP_DWORDS.keys())

    # Existing: add vmEntrypoint to set
    calloutTargets.add(vmEntrypoint)

    # Existing: extract X86CALLOUT targets
    for i in insns:
        if isinstance(i,RawX86Callout):
            if i.X86Target not in NOT_VIRTUALIZED:
                calloutTargets.add(i.X86Target)

    # Existing: return list of targets
    return list(calloutTargets)
```

Now I ran the function prologue extraction scripts from Part #3, Phase #3 again, to re-generate the virtualized function prologue and VM instruction entry keys for the function pointers in addition to the existing data for the X86CALLOUT targets. I then pasted the output data back into the second devirtualization program we wrote in this Phase #4 (remembering to copy in the data I'd generated manually for those virtualized functions without junk obfuscation sequences), and ran the devirtualization again. This time, the unreferenced functions had proper prologues, though they were still unreferenced.

4.3 Fixing the Function Pointers in the Devirtualized x86 Code

The last remaining issue is that the devirtualized x86 instructions which reference the function pointers still use the addresses of the virtualized functions in the .text section, whereas we want to modify them to point to their devirtualized equivalents instead. This is implemented in the RebuildX86 devirtualization function after the machine code array for the devirtualized program has been fully generated.

Fortunately for us, we already know which VM instructions reference the function pointers -- we collected that information when we modified ApplyFixup() to locate and log virtualized function pointers. Not only did we log the virtual addresses of the purported function pointers, but we also logged a list of VM instruction positions referencing each such function pointer in the ALL_FIXED_UP_DWORDS dictionary.

4.3.1 A Slight Complication

A slight complication lead me to a solution that perhaps could have been more elegant. Namely, we collect the positions of the VM instructions referencing function pointers within ApplyFixup() at the time that we disassemble the VM bytecode program. However, the simplifications in Part #3, Phase #1 can potentially merge instructions together when collapsing patterns of VM instructions into smaller sequences. Therefore, it might be the case that the VM instruction positions that we have collected no longer refer to valid locations in the VM program after the simplifications have been applied. However, we'd still expect the function pointers to appear in the machine code for the VM instructions into which the removed instruction was merged.

To work around this issue, I made use of the locsDict dictionary that we generated through devirtualization. Namely, that dictionary recorded the offset within the devirtualized x86 blob of each VM instruction processed in the main devirtualization loop. We find the offset within the devirtualized x86 machine code array of the prior VM instruction with an entry within locsDict, and we find the devirtualized offset of the next VM instruction with an entry within locsDict. This gives us a range of bytes to search in the devirtualized machine code looking for the byte pattern corresponding to the function pointer for the virtualized function. Once found, we can replace the raw bytes with the address of the devirtualized function body for that virtualized function.

4.3.2 Locating the Function Pointers in the Devirtualized Blob

Here is the code for locating function pointers as just described; if it is still unclear, read the prose remaining in this subsection.

```
# dword: the virtual address of a virtualized function
# posList: the list of VM instruction positions
# referencing the value of dword
for dword, posList in ALL_FIXED_UP_DWORDS.items():

    # For each position referencing dword:
    for pos in posList:

        # Set the low and high offset within the
        # devirtualized blob to None
        lowPos,highPos = None,None

        # posSearchLow is the backwards iterator
        posSearchLow = pos

        # Continue while we haven't located a prior
        # instruction with a devirtualization offset
        while not lowPos:

            # Does posSearchLow correspond to a
            # devirtualized instruction? I.e., not
            # something eliminated by a pattern
            # substitution.
```

```
            if posSearchLow in locsDict:

                # Yes: get the position and quit
                lowPos = locsDict[posSearchLow]

            else:
                # No: move to the previous instruction
                posSearchLow -= INSN_DESC_SIZE

        # Now search for the next higher VM position
        # with a devirtualization offset
        posSearchHigh = pos+INSN_DESC_SIZE

        # Continue while we haven't located a later
        # instruction with a devirtualization offset
        while not highPos:

            # Does posSearchLow correspond to a
            # devirtualized instruction? I.e., not
            # something eliminated by a pattern
            # substitution.
            if posSearchHigh in locsDict:

                # Yes: get the position and quit
                highPos = locsDict[posSearchHigh]
            else:
                # No: move to the next instruction
                posSearchHigh += INSN_DESC_SIZE
```

For each instruction position X that references one of the pointers to virtualized functions, I locate the last VM instruction at or before X in the locsDict array. This is implemented as a loop that tries to find X in locsDict. If locsDict[X] exists, we save that value -- the offset of the corresponding devirtualized instruction within the devirtualized blob. If locsDict[X] does not exist, then the VM instruction must have been removed by one of the pattern simplifications, so we move on to the prior VM instruction by subtracting the size of an instruction -- 0x18 -- from X. We repeat until we find an X that has been devirtualized; if X becomes 0x0, then we reach the beginning of the VM instructions, i.e., devirtualized offset 0x0.

We do much the same thing to find the next VM instruction with a corresponding devirtualized offset: add the size of a VM instruction -- 0x18 -- to X and look it up in locsDict. If it's not a member of locsDict, add 0x18 and try again. Once we find it, If X ever exceeds the last legal VM location, set the offset to the end of the machine code array. Once we've found the next VM instruction's devirtualized position, we record it and stop searching.

4.3.3 Rewriting the Function Pointers

Immediately after the code just shown, we have now found a suitable range within the x86 machine code array that ought to contain the raw bytes corresponding to the virtual address of a virtualized function referenced via

pointer. Next we simply byte-search this portion of the array looking for that address, and once found, replace the address with that of the corresponding devirtualized function body. There is nothing especially complicated about this; we simply consult the book-keeping metadata that we gathered through devirtualization to locate the devirtualized offset of the virtualized function pointer, add the offset of the new image base at which we are inserting the devirtualized blob within the binary, and store the DWORD value at the found position within the machine code array.

4.3.4 Done

After writing the code just described, now the pointers to virtualized functions have been modified to point to their devirtualized counterparts. Here again are two references to virtualized functions before the modifications just described:

```
seg000:00003555     mov     dword ptr [esi+4], 4055D5h
seg000:0000355C     mov     dword ptr [esi], 40581Eh
```

And, the same code after modification:

```
seg000:00003555     mov     dword ptr [esi+4], 50154Ah
seg000:0000355C     mov     dword ptr [esi], 5017B3h
```

5. Inserting the Devirtualized Code back Into the Binary

The last step before we can analyze the FinSpy dropper is to re-insert our devirtualized blob back into the binary. We have already chosen an address for it: 0x500000, which was important in generating the devirtualized code.

At this point I struggled to load the devirtualized code with with IDA's File->Load File->Additional binary file... and Edit->Segments->Create Segment menu selections. Although both of these methods allowed me to load the raw devirtualized machine code into the database, I experienced weird issues with both methods. Namely, the cross-section data references and/or function cross-references were broken. IDA might display the correct addresses for data items, and allow you to follow cross-references by pressing "enter" over an address, but it would not show symbolic names or add cross-references. For example we might see something like this:

```
.data:00504C3C call    dword ptr ds:401088h
```

Rather than what we see when things are working properly:

```
.data:00504C3C call    ds:SendMessageW
```

I tried screwing with every option in the two dialogs mentioned, especially the segment attributes ("CODE" instead of "DATA"). For some attempts, the code references worked properly but the data references didn't; and for other attempts, the opposite was true. More often neither would work. Igor Skochinsky from Hex-Rays has

always been very helpful to me, but this time he was away from his keyboard and did not hear my cries of anguish until it was too late. (Thanks anyway, Igor.)

That being the case, although it wasn't my first choice, I ended up enlarging the .data section via Edit->Segments->Edit Segment and then loading the binary contents with a one-liner in IDC:

```
loadfile(fopen("mc-take2.bin", "rb"), 0, 0x500000, 26840);
```

And this time, everything worked. You can see the IDB with the devirtualized code here.

From there I reverse engineered the devirtualized FinSpy dropper program. Whereas I was not impressed with the FinSpy VM, the dropper was more sophisticated than I was expecting. You can see a mostly-complete analysis in the linked IDB; start reading from address 0x50207E, the address of WinMain() within the devirtualized code. I've tried to comment most of the assembly language, but a lot of the action is inside of Hex-Rays (i.e., look at those functions inside of Hex-Rays to see a lot of comments that aren't in the assembly language view).

6. Conclusion

FinSpy VM is weak. It's closer in difficulty to a crackme than to a commercial-grade VM. (I suppose it is slightly more polished than your average VM crackme.) Having a "Raw x86" instruction in the FinSpy VM instruction set, where those instructions make up about 50% of the VM bytecode program, makes devirtualization trivial.

I almost didn't publish this series because I personally didn't find anything interesting about the FinSpy VM. But, hopefully, through all the tedium I've managed to capture the trials and tribulations of writing a deobfuscator from scratch. If you're still reading at this point, hopefully you found something interesting about it, and hopefully this slog wasn't all for nothing. Hopefully next time you come across a FinSpy-protected sample, this series will help you make short work of it.
