FinSpy VM Unpacking Tutorial Part 3: Devirtualization. Phase #2: First Attempt at Devirtualization

msreverseengineering.com/blog/2018/2/21/devirtualizing-finspy-phase-2-first-attempt-at-devirtualization

February 21, 2018

```
seg000:00000000      sub_0 proc near
seg000:00000000 000      mov     eax, [ebp+8]
seg000:00000003 000      mov     dword ptr [eax], 1D7h
seg000:00000009 000      mov     eax, 41F000h
seg000:0000000E 000      leave
seg000:0000000F -04      retn    4
seg000:0000000F      sub_0 endp ; sp-analysis failed
```

February 21, 2018 Rolf Rolles

[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 the previous Part #3, Phase #1, we inspected our FinSpy VM bytecode disassembly listing and discovered that the Group #2 instructions were used for obfuscation. After discovering obfuscation patterns and replacing them with simpler sequences, we obtained a simpler FinSpy VM bytecode program without any Group #2 instructions. After some further small improvements to the FinSpy VM bytecode disassembly process, we resume with a FinSpy VM bytecode listing that is suitable for devirtualization.

This Phase #2 discusses our first attempt at devirtualization. In fact, given the work done heretofore, the devirtualization code comes to about 100 lines of thoroughly-commented Python code. We then inspect our devirtualization, and discover several remaining issues that require correction before our task is complete. This phase fixes one of them, and the subsequent Part #3, Phase #3 examines the genesis of the remaining issues before fixing them. Part #3, Phase #4 makes a second (and eventually successful) approach toward devirtualizing FinSpy VM bytecode programs.

2. Whole-Program Devirtualization, First Attempt

After the previous Part #3, Phase #1 substitutions, our remaining FinSpy VM bytecode contains only Group #3 VM instructions (containing raw x86 machine code) and Group #1 VM instructions (all 16 varieties of x86 conditional branch instructions, and the now-correctly-identified unconditional JMP VM instructions). The following snippet is fairly

representative of our FinSpy VM bytecode program at present. It contains only "Raw X86" instructions, "X86CALLOUT", "X86JUMPOUT", and conditional/unconditional branch instructions -- the only categories of remaining VM instructions after the previous phase. It basically looks like x86 already, except the control flow instructions are FinSpy VM instructions instead of x86 instructions.

```
0x008e50: X86 mov ebx, edi
0x008e80: JMP VM[0x008aa8]
0x008e98: X86 push 2E50340Bh
0x008ec8: X86 push dword ptr [420358h]
0x008f28: X86CALLOUT 0x408360
0x008f40: X86 test eax, eax
0x008f58: JZ VM[0x009108] (fallthrough VM[0x008f70])
0x008f70: X86 lea ecx, dword ptr [ebp+0FFFFFFFCh]
0x008fd0: X86 push ecx
0x009000: X86 push 0FFFFFFFFh
0x009030: X86JUMPOUT jmp eax
0x009048: X86 test eax, eax
0x009060: JZ VM[0x009108] (fallthrough VM[0x009078])
0x009078: X86 cmp dword ptr [ebp+0FFFFFFFCh], 0h
0x009090: JZ VM[0x009108] (fallthrough VM[0x0090a8])
0x0090a8: X86 xor eax, eax
0x0090c0: X86 inc eax
0x0090d8: X86 leave
0x0090f0: X86 ret
0x009108: X86 xor eax, eax
0x009120: X86 leave
0x009138: X86 ret
```

2.1 Instruction-By-Instruction Devirtualization

At this point -- very quickly and easily into the analysis of the FinSpy VM bytecode program -- I felt comfortable writing a tool to reconstruct an x86 machine code program from the VM bytecode program. (Perhaps I was too comfortable, considering that a lot of work remained after my first attempt.) All of the remaining instructions seemed amenable to one-by-one translation back into x86. Namely, I wrote a simple loop to iterate over each VM instruction and create an x86 machine code array for the devirtualized program. At this point, there are four cases for the remaining VM instructions: the three varieties of Group #3 instructions, and the Group #1 branch instructions. We show simplified code for the devirtualizer, and then discuss the cases for the instruction types in more detail.

```
# Given: insns, a list of FinSpy VM instructions
# Generate and return an array of x86 machine code bytes for the VM program
def RebuildX86(insns):
    # Array of x86 machine code, built instruction-by-instruction
    mcArr  = []

    # Bookkeeping: which VM position/key corresponds to which
    # position within the x86 machine code array mcArr above
    locsDict = dict()
    keysDict = dict()

    # List of fixups for branch instructions
    locFixups = []

    # Iterate through the FinSpy VM instructions
    for i in insns:

        # currLen is the current position within mcArr
        currLen = len(mcArr)

        # Bookkeeping: memorize the x86 position for the
        # VM instruction's VM position and VM key
        locsDict[i.Pos] = currLen
        keysDict[i.Key] = currLen

        # Is it "Raw X86" or "X86JUMPOUT"? Just emit the
        # raw x86 machine code if so
        if isinstance(i,RawX86StraightLine):
            mcArr.extend(i.Remainder[0:i.DataLen])
        elif isinstance(i,RawX86Jumpout):
            mcArr.extend(i.Instruction[0:i.DataLen])

        # Is it a branch instruction?
        elif isinstance(i, ConditionalBranch):

            # Get the name of the branch
            jccName = INSN_NAME_DICT[i.Opcode]

            # Is this an unconditional jump?
            if jccName == "JMP":

                # Emit 0xE9 (x86 JMP disp32)
                mcArr.append(0xE9)

                # Opcode is 1 byte
                dispPos = 1

            # Otherwise, it's a conditional jump
            else:
                # Conditional jumps begin with 0x0F
                mcArr.append(0x0F)

                # Second byte is specific to the condition code
                mcArr.append(JCC_TO_OPCODE_DICT[jccName])
```

```
            # Opcode is 2 bytes
            dispPos = 2

        # Emit the displacement DWORD (0 for now)
        mcArr.extend([0x00, 0x00, 0x00, 0x00])

        # Emit a fixup: the JMP displacement targets
        # the VM location specified by i.VMTarget
        locFixups.append((i.Pos,dispPos,i.VMTarget))

    # Is it X86CALLOUT?
    elif isinstance(i,RawX86Callout):

        # We aren't handling this just yet.
        # Emit E8 00 00 00 00 (CALL $+5)
        # Revisit later
        mcArr.append(0xE8)
        mcArr.extend([0x00, 0x00, 0x00, 0x00])
```

Now we discuss the cases above:

- The instruction is Group #3 "Raw X86", which already contains raw x86 machine code in an array inside of the VM instruction object. In this case, we simply append the raw x86 machine code to the array.
- The instruction is Group #3 "Call Indirect". As this instruction also contains raw x86 machine code, the initial idea and approach was to simply append it to the array identically to the previous case. (I later discovered that this case required more care, though it ended up not being very difficult.)
- The instruction is a Group #1 conditional or unconditional jump. We discuss these in a subsequent subsection.
- The instruction is "Call Direct" (symbolized as X86CALLOUT in the FinSpy VM disassembly listing). We discuss these later in this subsection.

2.2 Bookkeeping Cross-References

While translating the instructions as described above, we update two dictionaries to keep track of the position of each VM instruction within the x86 machine code array. Recall from the introduction in Part #3, Phase #1 that each VM instruction has two uniquely identifying characteristics: 1) its offset within the VM bytecode array, a multiple of 0x18 (the length of a VM instruction), and 2) a "key" DWORD used to locate instructions. The two dictionaries, called locsDict and keysDict respectively, map each instruction's position, and each instruction's key, to the offset within the x86 machine code array where the devirtualized instruction resides. Reproducing the pertinent snippets from the simplified code above:

```
# Given: insns, a list of FinSpy VM instructions
# Generate and return an array of x86 machine code bytes for the VM program
def RebuildX86(insns):
    # Array of x86 machine code, built instruction-by-instruction
    mcArr  = []

    # Bookkeeping: which VM location/key corresponds to which
    # position within the x86 machine code array mcArr above
    locsDict = dict()
    keysDict = dict()

    # Iterate through the instructions
    for i in insns:

        # currLen is the current position in mcArr
        currLen = len(mcArr)

        # Bookkeeping: memorize the x86 position for the
        # VM instruction's VM position and VM key
        locsDict[i.Pos] = currLen
        keysDict[i.Key] = currLen

        # ... code to devirtualize was shown above ...
```

For example, the first VM instruction (the one at position 0x0 within the VM instruction array) is emitted at position 0 within the x86 machine code blob. Thus, we update the dictionary locsDict with the binding locsDict[0x0] = 0x0 (i.e., VM bytecode position 0x0 -> x86 machine code position 0x0). The first instruction has key 0x5A145D, so we update the dictionary keysDict with the binding: keysDict[0x5A145D] = 0x0.

The second VM instruction shall be devirtualized after the first instruction. The devirtualized first instruction was three bytes in length, thus the second VM instruction begins at position 3 within the x86 machine code array. Since the second VM bytecode instruction corresponds to position 0x78 within the VM bytecode array (after the simplifications from the previous Part #3, Phase #1), we update our dictionary with the information: locsDict[0x78] = 3 (i.e., VM bytecode position 0x78 -> x86 machine code instruction position 0x3). The second instruction has key 0x5A1461, so we update the dictionary keysDict with the binding: keysDict[0x5A1461] = 0x3.

This process continues sequentially for every instruction that we translate. We start a counter at 0x0, add the length of the devirtualized x86 machine code after each step, and at each iteration we associate the current value of the counter with the instruction's key and position before generating x86 machine code.

2.3 De-Virtualizing Branch Instructions

We have deferred discussing conditional branches; now we shall take the subject up again.

x86 branch instructions are encoded based on the distance between the address of the branch instruction and the address of the target. The addresses themselves of the source and destination locations aren't important, only the difference between them. The opcode of each of the 16 x86 conditional branch types is 0F 8x for some value of x; the opcode for an unconditional branch is E9. To encode a jump targeting the address 0x1234 bytes after the JMP instruction, this would be encoded as E9 34 12 00 00. If we needed to encode an x86 "JZ" instruction, whose destination was 0x1234 bytes after the JZ instruction, the encoding would be 0F 84 34 12 00 00.

Since we are devirtualizing VM instructions one-by-one in forward order, if the branch target is in the reverse direction, that means we've already devirtualized the destination, and hence we could immediately write the proper displacement. However, if the branch target is in the forwards direction, that means the destination lies at an address of a VM instruction that we haven't translated yet, and so we don't know the address of the destination, and so we can't immediately generate the x86 branch instruction. This conundrum -- common in the world of linkers -- necessitates a two-phase approach.

1. Phase #1: During devirtualization, the first thing to do is to emit the opcode for the jump (recalling the previous examples, e.g., E9 for an unconditional JMP, or 0F 84 for a conditional JZ). This is easy; we just look up the proper opcode for a given jump type within a dictionary that maps VM branch instruction types to x86 opcodes. After the x86 opcode for the branch type, we write a displacement DWORD of 0x0. Crucially, we also generate a "fixup": we add an entry to a list that contains both the position of the displacement DWORD within the x86 machine code array, as well as the VM bytecode instruction EIP to which the jump must eventually point. These fixups are processed by phase two. (Note that the code for phase #1 was shown in the previous section.)
2. Phase #2: After devirtualizing the entire FinSpy VM bytecode program, the "locsDict" dictionary described in the previous section on bookkeeping now has information about where each VM instruction lies within the devirtualized x86 machine code array. We also have a list of fixups, telling us which locations in the x86 machine code program correspond to the dummy 0x0 DWORDs within the branch instructions that we generated in the previous phase, which now need to be fixed up to point to the correct locations. Each such fixup also tells us the position of the VM bytecode instruction (within the VM bytecode array) to which the branch should point. Thus, we simply look up the destination's devirtualized x86 machine code position within locsDict, and compute the difference between the position after the jump displacement DWORD and the position of the destination. We replace the dummy 0x0 DWORD with the correct value. Voila, the branches now have the correct destinations.

Here is the code for phase #2, which executes after the main loop in "RebuildX86" has emitted the devirtualized x86 machine code for all VM instructions:

```
# Fixups contain:
# * srcBegin: beginning of devirtualized branch instruction
# * srcFixup: distance into devirtualized branch instruction
#             where displacement DWORD is located
# * dst:      the position within the VM program where the
#             branch destination is located
for srcBegin, srcFixup, dst in locFixups:
    # Find the machine code address for the source
    mcSrc = locsDict[srcBegin]
    # Find the machine code address for te destination
    mcDst = locsDict[dst]
    # Set the displacement DWORD within x86 branch instruction
    StoreDword(mcArr, mcSrc+srcFixup, mcDst-(mcSrc+srcFixup+4))
```

(Pedantic optional note: x86 also offers short forms of the conditional and unconditional branch instructions: if the displacement of the destination fits within a single signed byte, there are shorter instructions that can be used to encode the branch. I.e., EB 50 is the encoding for unconditionally jumping to 50 bytes after the source instruction, and 75 F0 performs a JNZ to 0xFFFFFFF0 bytes after the source instruction. Because using the short forms requires a more sophisticated analysis, I chose to ignore the short forms of the branch instructions (and use only the long forms) when performing devirtualization. The only side effect of this is that some branch instructions in the devirtualized program are longer than necessary -- but in terms of their semantic effects, the long and short branches function identically. If you wanted to be less lazy about this, and emit small jumps when applicable, you should read this paper or the source code to an x86 assembler.)

2.4 Devirtualizing X86CALLOUT Instructions, Take One

The last variety of VM instruction that we need to handle is the X86CALLOUT instruction. These instructions specify an RVA within the .text section at which the targeted x86 function begins. x86 CALL instructions are encoded identically to x86 branch instructions -- following the x86 CALL opcode E8, there is a DWORD specifying the distance from the end of the x86 CALL instruction to the target address. Thus, like we just discussed for devirtualizing branch instructions, we need to know the source and destination addresses for the CALL to generate the proper displacement for the x86 CALL instruction.

The targets of the virtualized branch instructions are specified as locations within the VM program. I.e., none of the jumps exit the VM. Thus, our task in fixing the branch instructions was simply to locate the distance between the branch instruction in the devirtualization and its target. Since x86 branch instructions are relatively-addressed, we do not need to take into account where within the original binary we will eventually reinsert the devirtualized code -- just the distance between the source and target address is enough information.

On the other hand, the targets of all X86CALLOUT instructions are outside of the VM. Like with jumps, we'll need to know the distance between the source and target addresses. Unlike with jumps, for the devirtualized CALL instructions, we do need to know where within

the original binary our devirtualized code shall be located in order to compute the CALL displacement DWORD correctly.

Since I hadn't decided where I was going to store the devirtualized code, and yet I wanted to see if my devirtualization process was otherwise working, I decided to postpone resolving this issue. I decided for the time being to just generate dummy x86 instructions to devirtualize X86CALLOUT instructions. I.e., for each "Call Direct" VM instruction, at this stage, I chose to emit E8 00 00 00 00 (x86 CALL $+5) for its machine code. Clearly this is an incorrect translation -- to reiterate, this approach is just a placeholder for now -- and the devirtualized program will not work at this stage (and we won't see proper call destinations in the devirtualized program), but doing this allowed me to proceed without having to decide immediately where to put the devirtualized code.

(Note that when it came time to fix this issue, I actually discovered a second and much more severe set of issues with devirtualizing X86CALLOUT instructions, which ended up consuming roughly half of my total time spent in the devirtualization phase. We will return to those issues when they arise naturally.)

3. Moment of Truth, and Stock-Taking

At last, we have our first devirtualized version of the FinSpy VM bytecode program for this sample. If you would like to see the results for yourself, you can load into IDA the binary for the first devirtualization. Now for the real test: did it work? How close to being done are we?

I took the output .bin file and loaded it into IDA. It looked pretty good! Most of it looks like something that was generated by a compiler. After the thrill of initial success wore off, I began looking for mistakes, things that hadn't been properly translated, or opportunities to improve the output. After this investigation, I'll go back to the drawing board and see what I can do about fixing them. Then I'll do the same thing again: look at the second iteration of the output, find more issues, fix them, and repeat. Hopefully, this process will eventually terminate. (It did.)

3.1 Problem #1: Call Targets are Missing

This problem was anticipated. As discussed in the last section, since I wasn't sure just yet how to handle the X86CALLOUT targets, I deliberately emitted broken x86 CALL instructions (namely E8 00 00 00 00, CALL $+5), with the plan to fix them later. Not surprisingly, those instructions were broken in the devirtualized output:

```
seg000:000005D6 BE 08 02 00 00        mov     esi, 208h
seg000:000005DB 56                    push    esi
seg000:000005DC 8D 85 C0 FB FF FF     lea     eax, [ebp-440h]
seg000:000005E2 57                    push    edi
seg000:000005E3 50                    push    eax
seg000:000005E4 E8 00 00 00 00        call    $+5 ; <- call target missing
seg000:000005E9 56                    push    esi
seg000:000005EA 8D 85 C8 FD FF FF     lea     eax, [ebp-238h]
seg000:000005F0 57                    push    edi
seg000:000005F1 50                    push    eax
seg000:000005F2 E8 00 00 00 00        call    $+5 ; <- call target missing
seg000:000005F7 56                    push    esi
seg000:000005F8 8D 85 A8 F5 FF FF     lea     eax, [ebp-0A58h]
seg000:000005FE 57                    push    edi
seg000:000005FF 50                    push    eax
seg000:00000600 E8 00 00 00 00        call    $+5 ; <- call target missing
seg000:00000605 83 C4 24              add     esp, 24h
```

Therefore I don't know where any of the calls are going, and so there are no function call cross-references. Obviously we will need to stop kicking the can down the road at some point and properly attend to this issue.

3.2 Observation #2: What are These Indirect Jump Instructions?

Another thing I noticed was an odd juxtaposition of indirect jump instructions where the surrounding context indicated that an indirect call would have been more sensible. For example:

```
seg000:00000B51     push    dword ptr [ebp-0Ch]
seg000:00000B54     mov     eax, ds:41FF2Ch
seg000:00000B59     jmp     dword ptr [eax+14h]
seg000:00000B5C     mov     eax, ds:41FF38h
seg000:00000B61     push    esi
seg000:00000B62     jmp     dword ptr [eax+90h]
seg000:00000B68     mov     eax, ds:41FF38h
seg000:00000B6D     jmp     dword ptr [eax+74h]
```

We see arguments being pushed on the stack, as though in preparation for a call. Then, instead of a call, we see a jump. Then, the assembly language indicates that another call should be taking place, but instead there's another indirect jump. There are no intervening cross-references jumping to the locations after the indirect jumps. What's up with that?

3.3 Problem #3: None of the Functions Have Prologues

Although a lot of the functions looked exactly like x86 machine code emitted by Microsoft Visual Studio to me, something was off. IDA tipped me off to the problem, since every function had a red message indicating a problem with the function's stack frame. Here is a screenshot of IDA, showing the colored message (and with the stack pointer shown for illustration):

```
seg000:00000000      sub_0 proc near
seg000:00000000 000      mov     eax, [ebp+8]
seg000:00000003 000      mov     dword ptr [eax], 1D7h
seg000:00000009 000      mov     eax, 41F000h
seg000:0000000E 000      leave
seg000:0000000F -04      retn    4
seg000:0000000F      sub_0 endp ; sp-analysis failed
```

Most of the code looks good, but the "leave" instruction is supposed to pop the EBP register off the stack -- and yet the prologue does not push the EBP register onto the stack. Also, the first instruction makes reference to "[EBP+8]", which assumes that EBP has previously been established as the frame pointer for this function -- and yet, being the first instruction in the function, clearly the devirtualized code has not yet established EBP as the frame pointer. Which memory location is actually being accessed by this instruction?

In other functions, we see the epilogues popping registers off the stack -- but inspecting the beginnings of those functions, we see that those instructions were not previously pushed onto the stack. Here's an example from the beginning of a function:

```
seg000:000004FB     mov     ebx, [ebp+8]     ; Overwrite EBX
seg000:000004FE     mov     edi, [ebx+3Ch]   ; Overwrite EDI
seg000:00000501     add     edi, ebx
seg000:00000503     mov     eax, [edi+0A0h]
seg000:00000509     test    eax, eax         ; Early return check
seg000:0000050B     jz      return_ebx       ; JZ taken => fail, return

; ... more function code ...

seg000:000005A3 return_ebx:
seg000:000005A3     pop     edi              ; Restore EDI (NOT PREVIOUSLY SAVED)
seg000:000005A4     mov     eax, ebx
seg000:000005A6     pop     ebx              ; Restore EBX (NOT PREVIOUSLY SAVED)
seg000:000005A7     leave                    ; Restore EBP (NOT PREVIOUSLY SAVED)
seg000:000005A8     retn    8                ; Return
```

We're going to need to know why these functions seemingly are missing instructions at their beginnings.

I can see why IDA is complaining about these functions -- despite being mostly coherent x86 implementations of C functions, they are clearly slightly broken. But how are they broken, and how can I fix them?

At this point I remembered something about the original binary. Back in part one, we analyzed a few VM entrypoints, and noticed that they began with normal-looking function prologues, before a segue into gibberish instructions, before finally pushing a VM key on the stack and transferring control to the VM entrypoint. To wit, here is the x86 code corresponding to the virtualized function from the screenshot above:

```
.text:00401340     push    ebp                     ; Ordinary prologue
.text:00401341     mov     ebp, esp                ; Ordinary prologue
.text:00401343     push    esi                     ; Save obfuscation register #1
.text:00401344     push    edx                     ; Save obfuscation register #2
.text:00401345     mov     edx, offset word_403DFE ; Junk obfuscation
.text:0040134A     xor     esi, esp                ; Junk obfuscation
.text:0040134C     shl     esi, cl                 ; Junk obfuscation
.text:0040134E     shr     esi, 1                  ; Junk obfuscation
.text:00401350     shl     esi, cl                 ; Junk obfuscation
.text:00401352     pop     edx                     ; Restore obfuscation register #2
.text:00401353     pop     esi                     ; Restore obfuscation register #1
.text:00401354     push    5A145Dh                 ; Push VM instruction key
.text:00401359     push    edx                     ; Obfuscated JMP
.text:0040135A     xor     edx, edx                ; Obfuscated JMP
.text:0040135C     pop     edx                     ; Obfuscated JMP
.text:0040135D     jz      GLOBAL__Dispatcher      ; Enter VM
```

That's where the missing function prologues went -- they are still at the locations where at which original functions resided within the binary. The prologues have not been virtualized. Upon calling an x86 function that has been virtualized, the function prologue will execute in x86 as usual, before the virtualized body of the function is executed within the VM. So in order to properly devirtualize the functions within the VM bytecode, we will need to extract their prologues from the x86, and during devirtualization, prepend those prologue bytes before the devirtualization of the function bodies.

4. Fixing the Indirect Jumps

Now that we've identified a few issues with the devirtualized code, our next task is to fix them. We'll start with the lowest-hanging of the fruit, the weird indirect jumps, reproduced here for coherence:

```
seg000:00000B51     push    dword ptr [ebp-0Ch]
seg000:00000B54     mov     eax, ds:41FF2Ch
seg000:00000B59     jmp     dword ptr [eax+14h]
seg000:00000B5C     mov     eax, ds:41FF38h
seg000:00000B61     push    esi
seg000:00000B62     jmp     dword ptr [eax+90h]
seg000:00000B68     mov     eax, ds:41FF38h
seg000:00000B6D     jmp     dword ptr [eax+74h]
```

The x86 JMP instructions were copied verbatim from machine code stored within X86JUMPOUT instructions. So, in a sense, the JMP instructions are "correct". Nevertheless, this disassembly listing looks wrong. The indirect jump does not push a return address, so the code at the destination doesn't know where to resume executing. Consequently, the instruction following one of the indirect jumps will never execute unless its address is referenced somewhere else in the devirtualized code -- but that didn't seem to be the case, and it would have been weird if it was the case, since none of the situations in which a compiler would have emitted an address contained within the body of a function (a

switch case, or an exception handler) seemed to apply. Another possibility would have been tail calls to function pointers, but the context of the indirect jumps did not indicate that a tail call was about to take place.

It struck me that these indirect jumps probably ought to be indirect calls instead. To investigate, I looked again at the disassembly of the VM instruction handler for the "Indirect Call" instructions. The "Indirect Call" FinSpy VM instruction is one of the ones that generates code dynamically while executing. Reproduced from part two, here's the code that the FinSpy VM generates dynamically when interpreting an "Indirect Call" instruction:

```
popf                          ; Restore flags from VMContext
popa                          ; Restore registers from VMContext
push offset @RESUME           ; PUSH RETURN ADDRESS (@RESUME)
[X86 machine code for indirect jump, copied from VM instruction]
@RESUME:                      ; <- RETURN LOCATION
push offset NextVMInstruction ; Resume at next VM insn
pusha                         ; Save registers
pushf                         ; Save flags
push offset VMContext
pop ebx                       ; EBX := VMContext *
push offset fpVMReEntryReturn
ret                           ; Re-enter VM
```

So indeed, before executing the indirect jump instruction from the x86 machine code stored within the instruction, the FinSpy VM "Indirect Call" handler pushes a return address on the stack. The return address points to the dynamically-generated code at the @RESUME label in the snippet above, which then continues execution at the next VM instruction following the "Indirect Call" instruction. That's why the devirtualized code isn't pushing a return address -- the VM takes care of that.

I expect that the FinSpy VM authors have code to translate x86 indirect call instructions into x86 indirect jump instructions. Thus I will need to do the reverse process: I will need to take the raw machine code contained in the "Indirect Call" VM instructions, and convert the x86 indirect jump instructions into call instructions instead. I wrote the following function, and added a call to it in the constructor for the Python object representing an "Indirect Call" VM instruction:

```
# This function disassembles the raw machine code for indirect jump instructions,
# changes the instructions therein to indirect call instructions, re-assembles
# them, and returns the new machine code with its textual disassembly.
#
# Input: bytes, an array of machine code for an indirect jump.
# Output: a tuple (string: disassembly, bytes: machine code for indirect call)

def ChangeJumpToCall(bytes):

    # Decode the x86 machine code
    i2container = X86Decoder(StreamObj(bytes)).Decode(0)

    # Fetch the instruction from the decoded bundle
    insn = i2container.instr

    # Ensure it's an indirect jump!
    assert(insn.mnem == XM.Jmp)

    # Change the mnemonic to call
    insn.mnem = XM.Call

    # Return new textual disassembly and machine code
    return (str(insn),EncodeInstruction(insn))
```

In reality, using my disassembler library was overkill here. There are only a few ways to encode indirect calls and indirect jumps in x86 machine code; simple pattern-replacement could have identified and re-written them very easily. But despite being overkill, there are no technical drawbacks with the solution based on rewriting and reassembling the instructions.

The devirtualized binary after the above changes can be found here.

5. Conclusion

We began this Phase #2 with a deobfuscated FinSpy VM bytecode disassembly listing. From there, we formulated a strategy to devirtualize the program instruction-by-instruction. This was mostly straightforward, except we deliberately did not devirtualize the X86CALLOUT instructions from Group #3. After devirtualization, we discovered several remaining issues, all relating to functions and function calls. This phase ended by examining and fixing one of those issues.

In the next Part #3, Phase #3, we will examine the source of the function-related issues in our current devirtualization. In so doing, we will learn that we need to obtain some specific information about the X86CALLOUT VM instructions and virtualized functions that we shall need to obtain and incorporate into our devirtualization process. Part #3, Phase #3 will then write scripts to collect this information. The final Part #3, Phase #4 will then incorporate this information into the devirtualization process, and then discover and correct a few remaining issues before producing the final devirtualized listing for the FinSpy VM bytecode in our running example.