FinSpy VM Unpacking Tutorial Part 3: Devirtualization. Phase #3: Fixing the Function-Related Issues — Möbius Strip Reverse Engineering

By Rolf Rolles

Published: 2018-02-21 · Archived: 2026-04-05 13:58:54 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

At the end of Part #3, Phase #2, we noted that our first attempt at devirtualization is still subject to two remaining major issues.

- We had deferred generating proper displacements when devirtualizing the FinSpy VM X86CALLOUT instructions into x86 CALL instructions.
- We discovered that the functions in the devirtualized code were missing their prologues, since the prologues of virtualized function execute in x86 before entering the VM.

Resolving these issues was the most difficult part of devirtualizing FinSpy, and took about half of the total time I spent on devirtualization. I dreamed up one "nice" solution, and implemented instead one "hacky" solution, but both of them fall short of full automation.

Phase #3, this entry, discusses how FinSpy virtualizes functions and function calls. We illustrate the problems that FinSpy's design decisions present us in properly devirtualizing functions and function calls. We state precisely what problems need to be solved. Then, we discuss several ideas for solving them, and eventually write an IDAPython script to collect the information we need. We finish by attending to some issues that arise in the course of writing and using this script.

Part #3, Phase #4, the next entry, will incorporate this information into the devirtualization process. After fixing a few more issues, our journey in devirtualizing FinSpy will be complete.

2. How FinSpy Virtualizes Functions and Calls

Ultimately, our major remaining task in devirtualizing FinSpy VM bytecode programs is to devirtualize the X86CALLOUT instructions. To do that, we need to attend to the several issues described in Part #3, Phase #2 and in the introduction above. It turns out that these issues arise due to how the FinSpy VM virtualizes functions, and due to the mechanics of the X86CALLOUT VM instruction.

In looking at the FinSpy VM bytecode, we can see that function calls have been virtualized in a way that is very natural, identical to what we'd expect to see in an x86 binary compiled with ordinary compilation techniques. For

example:

```
0x006ff0: X86 push ebx
0x007020: X86 push dword ptr [ebp+0FFFFFFFCh]
0x007098: X86 push eax
0x0070c8: X86CALLOUT 0x40ae74
```

This sequence almost looks like an x86 function call sequence already. And for this particular example, if we inspect the code at the destination of the X86CALLOUT instruction -- 0x40ae74 -- we see a normal x86 function body:

```
.text:0040AE74     ; void *__cdecl memcpy(void *Dst, const void *Src, size_t Size)
.text:0040AE74     memcpy proc near
.text:0040AE74
.text:0040AE74     Dst= dword ptr  4
.text:0040AE74     Src= dword ptr  8
.text:0040AE74     Size= dword ptr  0Ch
.text:0040AE74
.text:0040AE74 000 jmp     ds:__imp_memcpy
.text:0040AE74
.text:0040AE74     memcpy endp
```

If we were to devirtualize the "X86CALLOUT" VM instruction from above into an x86 CALL instruction to the same location, this is indeed exactly what we would see in a normal binary. The obvious approach to devirtualize the X86CALLOUT instruction in the snippet above would be to replace it with an ordinary x86 CALL instruction targeting the function above, and there seem to be no drawbacks or serious complications with doing so in this case.

However, other X86CALLOUT instructions tell a different tale. Here's another, similar sequence of FinSpy VM instructions:

```
0x00f570: X86 push 40h
0x00f5a0: X86 push 1000h
0x00f5d0: X86 push eax
0x00f600: X86 push esi
0x00f630: X86CALLOUT 0x408810
```

In this example, at the location specified in the X86CALLOUT instruction -- 0x408810 -- we do not see an ordinary x86 function. Rather, like we discussed in part one, we see that the original, pre-virtualized function has been overwritten by a sequence of code that ultimately enters the FinSpy VM. To re-state: virtualized functions still reside at their original addresses in the binary, and can be executed by calling that location as usual; however, virtualized functions have been modified to transfer control to the suitable location inside of the VM bytecode program. (I suspect the FinSpy authors chose to retain virtualized function addresses in order to preserve certain

metadata in the PE64 header surrounding exception records and function start addresses.) The x86 code at the location from above shows:

```
.text:00408810     mov     edi, edi                ; Ordinary prologue #0
.text:00408812     push    ebp                     ; Ordinary prologue #0
.text:00408813     mov     ebp, esp                ; Ordinary prologue #0
.text:00408815     push    ecx                     ; Ordinary prologue #0
.text:00408816     push    ecx                     ; Ordinary prologue #0
.text:00408817     push    eax                     ; Save obfuscation register #1.1
.text:00408818     push    ebx                     ; Save obfuscation register #2.1
.text:00408819     mov     ebx, offset unk_41B164  ; Junk obfuscation
.text:0040881E     mov     eax, 0B3BF98Dh          ; Junk obfuscation

; ... more junk obfuscation ...

.text:0040883E     bswap   eax                     ; Junk obfuscation
.text:00408840     stc                             ; Junk obfuscation
.text:00408841     pop     ebx                     ; Restore obfuscation register #2.2
.text:00408842     pop     eax                     ; Restore obfuscation register #1.2
.text:00408843     push    5A7314h                 ; Push VM instruction entry key #3
.text:00408848     push    eax                     ; Obfuscated JMP
.text:00408849     xor     eax, eax                ; Obfuscated JMP
.text:0040884B     pop     eax                     ; Obfuscated JMP
.text:0040884C     jz      GLOBAL__Dispatcher      ; Enter FinSpy VM #4
```

As in the code above, virtualized functions have been overwritten by sequences of x86 instructions that do up to four things.

1. Execute the original prologue of the pre-virtualized function (on the lines labeled #0 above)
2. After saving two registers on the lines labeled #1.1 and #2.1, perform junk obfuscation (meaningless computations), and then restore the values of those registers on the lines labeled #2.2 and #1.2.
3. Push the VM instruction key for the virtualized function body onto the stack (on the line labeled #3 above)
4. Jump to the FinSpy VM interpreter (on the line labeled #4 above)

The most important observations from the code above are:

1. Whenever an X86CALLOUT instruction targets the x86 location 0x408810, the result is that the VM begins executing VM instructions starting with the one whose key is 0x5A7314.
2. For the virtualized function at x86 location 0x408810 (at the VM instruction keyed 0x5A7314), the virtualized function's prologue is the first five x86 instructions from the sequence above (all of which are labeled #0).

2.1 Comments on Devirtualization Strategy

In discussing X86CALLOUT VM instructions to non-virtualized targets, we suggested that we could devirtualize them into an x86 CALL instruction directly targeting the specified location. However, that strategy will not

produce good results when the targeted address is a virtualized function. If we were to devirtualize the X86CALLOUT instruction in the second example above with an x86 CALL instruction to the second targeted function, our resulting "devirtualization" would still contain obfuscation and references to the VM. I.e., upon encountering an x86 CALL instruction in the "devirtualized" code, we as analysts would have to analyze the x86 code shown above to determine which VM instruction ended up executing as a result, and to inspect the prologue for the targeted function, which would not be present at the site of the devirtualized function body. This is hardly a "devirtualization", given that we would still need to analyze obfuscated code involving the FinSpy VM, and that IDA and Hex-Rays will balk at functions with no prologues.

Therefore, to truly "devirtualize" FinSpy VM programs, we need to replicate what happens when a virtualized function is called: namely, we need to replicate the original function's prologue from before the VM entry sequence, and also determine the VM location at which the virtualized function's body resides, in order to emit an X86 CALL instruction from the site of the X86CALLOUT VM instruction to the devirtualized address of the destination, thereby entirely eliminating reliance on the FinSpy VM.

Hence, it is clear that we need two different strategies for devirtualizing X86CALLOUT instructions. For non-virtualized targets, we could simply emit x86 CALL instructions to the destination. For virtualized targets, we would need to determine the VM address of the VM bytecode translation for the function, locate the devirtualized versions of those VM instructions within our devirtualized array, and replace the X86CALLOUT VM instruction with an x86 CALL instruction pointing to the proper location within the devirtualized code.

Additionally, for each virtualized function, we need to extract its function prologue from the .text section, and insert those instructions before the devirtualized body of the corresponding function.

3. Stating Our Task Precisely

To devirtualize X86CALLOUT instructions, we need the following information:

- A list of all x86 locations used as call targets, and whether or not the functions at those locations are virtualized.
- For each virtualized function:
    - The VM instruction key pushed before entering the VM
    - The raw machine code comprising the function's prologue

With this information, we could solve our remaining issues as follows:

- When devirtualizing any FinSpy VM instruction, look up its VM instruction key to see if it's the first VM instruction implementing a virtualized function body. If so, insert the prologue bytes from the corresponding x86 function in the .text section before devirtualizing the instruction as normal.
- When devirtualizing X86CALLOUT instructions, check to see whether the destination is a virtualized function or not.
    - If the destination is virtualized, emit an x86 CALL instruction whose displacement points to the x86 address of the devirtualized function body within the blob of devirtualized code.
    - If the destination is not virtualized, emit an X86 CALL instruction pointing to the referenced location in the .text section. This requires knowledge of the base address at which the devirtualized

code shall be stored in the binary containing the FinSpy VM.

3.1 Challenges in Obtaining the Requisite Information

Unfortunately for us, we've got a bit of work ahead of us, since the FinSpy VM does not readily provide the information we're interested in.

- The FinSpy VM metadata does not contain a list of called function addresses (virtualized or not). We have to perform our own analysis to determine which x86 functions may be called through VM execution, and whether or not the called functions are virtualized.
- The FinSpy VM makes no distinction between the cases where the destination address is virtualized, and the case in which it is not virtualized. In both cases, the target of an X86CALLOUT location is simply an address in the .text section. If the target is virtualized, then there will be a FinSpy VM entry sequence at the destination. If the target is not virtualized, there will not be a FinSpy VM entry sequence at the destination. Thus, the onus is upon us to determine whether a particular call target is virtualized or not, and if it is, to extract its function prologue and determine the VM key of the VM instruction that implements the virtualized function's body.

4. Initial Approach to Discovering Virtualized Function Addresses

Now that our tasks are laid out, let's set about solving them. For the first task -- obtaining the list of virtualized function addresses -- my first thought was to extract the targets from all of the X86CALLOUT instructions. That idea was easy enough to implement. I wrote a small Python script to dump all of the X86CALLOUT targets from the FinSpy VM bytecode program:

```
# Iterate through the VM instructions and extract direct call targets
# Inputs:
# * insns: a list of VM instruction objects
# * vmEntrypoint: the address of WinMain() in the original binary
# Output:
# * A list of call targets
def ExtractCalloutTargets(insns, vmEntrypoint):

    # Create a set containing just the address of the VM entrypoint
    calloutTargets = set([vmEntrypoint])

    # Iterate through all VM instructions
    for i in insns:
        # Was the instruction an X86CALLOUT?
        if isinstance(i,RawX86Callout):
            # Add it to the set
            calloutTargets.add(i.X86Target)

    # Return a list instead of a set
    return list(calloutTargets)
```

We add the address of the VM entrypoint because it isn't explicitly called from anywhere within the VM program -- WinMain() contains a VM entry sequence that causes it to execute.

The code above was a good start toward obtaining the list of x86 functions that may be called by virtualized code, and as far as I knew at this point, this list contained all such function addresses. I later discovered that some virtualized functions are not referenced via X86CALLOUT instructions (analogously to how WinMain()'s virtualized entrypoint is not the target of an X86CALLOUT instruction), and hence are not included in the list of virtualized functions generated above. Later it became clear that not including those virtualized function addresses not referenced by X86CALLOUT instructions causes serious problems while collecting the function information in preparation for devirtualization. Still, all of those problems were yet to materialize; this was my initial, blissfully ignorant approach.

5. A Nice Approach to Extracting Information from Virtualized Functions

For extracting the VM entry keys and function prologues, the first solution that came to mind was the best one I came up with. It is not, however, what I ended up actually doing.

Some modern reverse engineering tools (such as the Unicorn Engine) support custom emulation of assembly language snippets (as opposed to full-system emulation). The user supplies a starting address and an initial state for the registers and memory. From there, the emulation is controllable programmatically: you can emulate one instruction at a time and inspect the state after each to determine when to stop. (My own program analysis framework, Pandemic, part of my SMT training class, also supports this functionality.)

With access to a customizable x86 emulator, and assuming that we knew the address of the VM interpreter, we could extract the VM entry instruction keys for a virtualized function as follows:

- Put the emulator's registers and memory into some initial state.
- Set the initial x86 EIP to the beginning of the virtualized function's x86-to-VM entrypoint.
- Emulate instructions until EIP reaches the address of the VM interpreter. Save each emulated instruction into a list.
- Extract the DWORD from the bottom of the stack. This is the VM key corresponding to the beginning of the function.
- Scan the list of emulated instructions in reverse order to determine which two registers are being used for junk obfuscation.
- Scan the list of emulated instructions in forward order until the point at which those two registers are pushed. The instructions before these two x86 PUSH instructions comprise the prologue for the function beginning at the starting address.

This is the best solution I have come up with so far, which mirrors the approaches I've taken for similar problems arising from different obfuscators. The emulation-based solution is very generic. For the task of recovering the VM entry key for a given entrypoint, it uses a semantic pattern -- no matter how the code at the VM entrypoint is implemented, the end result is that the VM entry key is pushed on the stack before the VM begins executing. Semantic pattern-matching is preferable in the extreme to syntactic pattern-matching. It would not matter if the FinSpy authors change the obfuscation at the virtualized functions, so long as the fundamental architecture of VM

entry remains the same. I expect that this method would continue to work until and unless there was a major overhaul of the FinSpy VM entry schema.

The logic for extracting the function prologues is based on syntactic pattern-matching. Therefore it is more brittle, and would require tweaking if the FinSpy VM authors changed the VM entrypoint obfuscation.

The reason I did not pursue the emulation-based solution is not especially interesting: I had just gotten a new laptop and had not installed my development environment at that time, and I wanted a quick solution. Anyway, even if I had, I'm not sure I would have released the code for it. While staring at build environment errors, I started to wonder if there was another way to obtain the x86 call target/VM key information. What I came up to replace the VM entry key extraction did work, but in all fairness, it wasn't a very good solution and is not suitable for a fully-automated solution, since it does involve some manual work.

6. Hacky Solution, Overview

Remember, our overall goal is to discover, for each virtualized function:

1. The key for the VM instruction implementing the virtualized function's body
2. The non-virtualized prologue instructions for the virtualized function

In lieu of the nicer, emulation-based solution, the hacky solution I came up with uses the IDA API to extract information in several separate parts before eventually combining them. The entire script can be found here.

1. First, we find all x86 locations that jump to the FinSpy VM interpreter.
2. Second, for each such referencing address, we extract the key pushed shortly beforehand, and also extract the names of the registers used by the junk obfuscation following the non-virtualized function prologue.
3. For each virtualized function start address, match it up with the subsequent address that enters the VM (i.e., match it with one of the addresses described in #1). From step #2, we know which VM key is pushed before entering the VM interpreter; thus, we now know which VM key corresponds to the function start address. We also know which registers are used for the junk obfuscation for the virtualized function beginning at that address.
4. Now that we know the identities of the registers used in junk obfuscation for a given virtualized function, scan forwards from its starting address looking for the beginning of the junk obfuscation -- i.e., two consecutive x86 PUSH instructions pushing those register values. Copy the raw bytes before the beginning of the junk obfuscation; these are the prologue bytes.
5. Correlate the addresses of the virtualized functions with the prologue bytes and VM instruction key extracted in previous steps.

6.1 Steps #1 and #2: Extract VM Entry Keys and Obfuscation Registers

The implementation for the first two steps just described is not very sophisticated or interesting. The first function, ExtractKey, takes as input the address of an instruction that jumps to the VM entrypoint. I.e., its input is the address of an instruction such as the one labeled #4 below:

```
; ... prologue, junk obfuscation above this ...
.text:004083FC     pop     eax                     ; Pop obfuscation register #2.2
.text:004083FD     pop     ebx                     ; Pop obfuscation register #1.2
.text:004083FE     push    5A6C26h                 ; Push VM instruction key -- #3
.text:00408403     push    eax                     ; Obfuscated JMP
.text:00408404     xor     eax, eax                ; Obfuscated JMP
.text:00408406     pop     eax                     ; Obfuscated JMP
.text:00408407     jz      GLOBAL__Dispatcher      ; Enter FinSpy VM #4
```

From there, the script disassembles instructions backwards, up to some user-specifiable amount, and inspects them one-by-one looking for the first x86 PUSH instruction (i.e., the one on the line labeled #3 above) that pushes the VM key of the first instruction in the virtualized function's body.

Once it finds the x86 PUSH instruction on line #3, it inspects the previous two x86 instructions -- those on lines #1.2 and #2.2 -- to see if they are x86 POP instructions, corresponding to the restoration of the registers the VM entry sequences use for obfuscation.

If all checks pass, the script returns a 4-tuple containing:

- The address of the branch to the VM (i.e., that of line #4 above)
- The VM key DWORD (from line #3, e.g. 0x5A6C26 in the above)
- The two x86 register names from line #1.2 and #2.2.

That code is shown below (excerpted from the complete script):

```
# Given an address that references the VM dispatcher, extract
# the VM instruction key pushed beforehand, and the names of
# the two registers used for junk obfuscation. Return a tuple
# with all information just described.
def ExtractKey(vmXref):
    currEa = vmXref
    Key = None

    # Walk backwards from the VM cross-reference up to a
    # specified number of instructions.
    for _ in xrange(MAX_NUM_OBFUSCATED_JMP_INSTRUCTIONS):

        # Is the instruction "PUSH DWORD"?
        if idc.GetMnem(currEa) == "push" and idc.GetOpType(currEa, 0) == o_imm:

            # Extract the key, exit loop
            Key = idc.GetOperandValue(currEa, 0)
            break

        # Otherwise, move backwards by one instruction
        else:
```

```
            currEa = idc.PrevHead(currEa, 0)

    # After looping, did we find a key?
    if Key is not None:
        # Extract the first operands of the two subsequent instructions
        prevEa1 = idc.PrevHead(currEa,  0)
        prevEa2 = idc.PrevHead(prevEa1, 0)

        # Were they pop instructions?
        if idc.GetMnem(prevEa1) == "pop" and idc.GetMnem(prevEa2) == "pop":
            # Return the information we just collected
            return (vmXref,Key,idc.GetOpnd(prevEa1,0),idc.GetOpnd(prevEa2,0))

        # If not, be noisy about the error
        else:
            print "%#lx: found key %#lx, but not POP reg32, inspect manually" % (xref, Key)
            return (vmXref,Key,"","")

    # Key not found, be noisy
    else:
        print "%#lx: couldn't locate key within %d instructions, inspect manually" % (xref,MAX_NUM_O
        return None
```

There is a second function, ExtractKeys, which iterates over all cross-references to the VM entrypoint, and calls ExtractKey for each. It is not interesting, and so the code is not reproduced in this document, though it is in the included source code.

The results of ExtractKeys -- the tuples from above -- are then saved in a Python variable called locKey.

```
locKey = ExtractKeys(0x00401950)
```

This script can go wrong in several ways. Some of the references to the VM entrypoint might not have been properly disassembled, and so those references won't be in the set returned by IDA's cross-reference functionality. Secondly, if the FinSpy VM authors modified their virtualized entrypoint obfuscation strategy, it might necessitate changes to the strategy of simply walking backwards. Also, I found out later that not every x86-to-VM entry sequence used junk obfuscation, so the pattern-matching looking for the two x86 POP instructions failed. But, we're getting ahead of ourselves; I found and fixed those issues later on.

6.2 Step #3: Correlating Virtualized Function Beginnings with VM Entrypoints

Now that we have information about each location that enters the VM, we need to correlate this information with the list of X86CALLOUT targets corresponding to virtualized functions. I.e., given an address that is targeted by an X86CALLOUT instruction, we want to determine which address will subsequently transfer control into the VM (i.e., one of the addresses inspected in the previous steps). To assist in explanation, an example of a VM entry sequence is shown.

```
.text:00408360     mov     edi, edi                ; Original function prologue #0
.text:00408362     push    ebp                     ; Original function prologue #0
.text:00408363     mov     ebp, esp                ; Original function prologue #0
.text:00408365     sub     esp, 320h               ; Original function prologue #0
.text:0040836B     push    ebx                     ; Original function prologue #0
.text:0040836C     push    edi                     ; Original function prologue #0
.text:0040836D     push    ebx                     ; Push obfuscation register #1.1
.text:0040836E     push    eax                     ; Push obfuscation register #2.1

; ... junk obfuscation not shown ...

.text:004083FC     pop     eax                     ; Pop obfuscation register #2.2
.text:004083FD     pop     ebx                     ; Pop obfuscation register #1.2
.text:004083FE     push    5A6C26h                 ; Push VM instruction key -- #3
.text:00408403     push    eax                     ; Obfuscated JMP
.text:00408404     xor     eax, eax                ; Obfuscated JMP
.text:00408406     pop     eax                     ; Obfuscated JMP
.text:00408407     jz      GLOBAL__Dispatcher      ; Enter FinSpy VM #4
```

In the previous section, we extracted information about each location -- such as the address labeled #4 above -- that enters the VM. Namely, we extract the VM instruction key for the virtualized function body from the line labeled #3, and the names of the two registers used for obfuscation on the lines labeled #2.2 and #1.2.

Now, given the beginning of a virtualized function's VM entrypoint -- such as the first address above labeled #0 -- we want to know the address of the instruction that ultimately transfers control into the VM, the one labeled #4 in the example above. Once we know that, we can look up the address of that instruction within the information we've just collected, which will then tell us which two registers are used for the junk obfuscation (the ones popped on the lines labeled #2.2 and #1.2 -- namely, EAX and EBX). From there, we can scan forward in the prologue (the instructions labeled #0 above) until we find x86 PUSH instructions that save those two registers before the junk obfuscation begins (namely the x86 PUSH instructions on the lines labeled #1.1 and #2.1 above). Once we find those two registers being pushed in that order, we know that we've reached the end of the prologue. Therefore, every instruction leading up to that point is the virtualized function's prologue. We also know the VM instruction key for the first instruction in the virtualized function's body, the one labeled #3 in the sequence above.

Correlating virtualized function addresses with VM entry addresses is simple. Since there is no control flow in the junk obfuscation, given the address of a virtualized function's entrypoint, the address of the sequentially-next JZ instruction that enters the VM is the corresponding VM entry address.

Therefore, I chose to sort the VM entry information from the previous step -- called locKey -- by the address of the JZ instruction, and stored the sorted list in a Python variable called sLocKey:

```
sLocKey = sorted(locKey, key=lambda x: x[0])
```

And then I wrote a small function that, given the start address for a virtualized function, iterates through sLocKey and finds the next greater address that enters the VM, and returns the corresponding tuple of information about the VM entry sequence.

```
# Given:
# * ea, the address of a virtualized function
# Find the entry in sLocKey whose VM entry branch
# location is the closest one greater than ea
# Ouput:
# The tuple of information for the next VM entry
# instruction following ea
def LookupKey(ea):
    global sLocKey
    for i in xrange(len(sLocKey)):
        if sLocKey[i][0] < ea:
            continue
        return sLocKey[i]
    return sLocKey[i-1]
```

Now by passing any virtualized function start address into the LookupKey function above, we will obtain the VM key and junk obfuscation register information from the next-subsequent address after the start address that enters the VM.

Two functions collate the virtualized function addresses with the register data:

```
# Given the address of a virtualized function, look up the VM key and
# register information. Return a new tuple with the virtualized function
# start address in place of the address that jumps to the VM interpreter.
def CollateCalloutTarget(tgt):
    assocData = LookupKey(tgt)
    return (tgt, assocData[1], assocData[2], assocData[3])

# Apply the function above to all virtualized function addresses.
def CollateCalloutTargets(targets):
    return map(CollateCalloutTarget, targets)
```

The only thing that can go wrong here is if we haven't located all of the JZ instructions that enter the VM. If we don't, then the information returned will correspond to some other virtualized function -- which is a real problem that did happen in practice, and required some manual effort to fix. Those issues will be discussed later when they arise.

6.3 Step #4: Extract Prologues for Virtualized Functions

At this point, for every virtualized function's starting address, we have information about the registers used in the junk obfuscation, as well as the VM instruction key for the virtualized function body. All that's left to do is extract

the x86 function's prologue.

This is a very simple affair using standard pattern-matching tricks, very similar to the techniques we used to extract the VM instruction key and junk obfuscation register names. We simply iterate through the instructions at the beginning of a function, looking for two x86 PUSH instructions in a row that push the two registers used in junk obfuscation. (I noticed that the first junk instruction after the two x86 PUSH instructions was "mov reg2, address", where "address" was within the binary. I added this as a sanity check.) The very simple code is shown below (excerpted from the complete script).

```
# Given:
# * CallOut, the target of a function call
# * Key, the VM Instruction key DWORD pushed
# * Reg1, the name of the first obfuscation register
# * Reg2, the name of the second obfuscation register
# Extract the prologue bytes and return them.
def ExtractPrologue(CallOut, Key, Reg1, Reg2):

    # Ensure we have two register names.
    if Reg1 == "" or Reg2 == "":
        return (Key, CallOut, [])

    currEa = CallOut

    # Walk forwards from the call target.
    for i in xrange(MAX_NUM_PROLOG_INSTRUCTIONS):

        # Look for PUSH of first obfuscation register.
        if idc.GetMnem(currEa) == "push" and idc.GetOpnd(currEa, 0) == Reg1:
            nextEa = idc.NextHead(currEa, currEa+16)

            # Look for PUSH of second obfuscation register.
            if idc.GetMnem(nextEa) == "push" and idc.GetOpnd(nextEa, 0) == Reg2:
                thirdEa = idc.NextHead(nextEa, nextEa+16)

                # Sanity check: first junk instruction is "mov reg2, address"
                if idc.GetMnem(thirdEa) == "mov" and idc.GetOpnd(thirdEa, 0) == Reg2:
                    destAddr = idc.GetOperandValue(thirdEa, 1)

                    # Was "address" legal?
                    if destAddr <= PROG_END and destAddr >= PROG_BEGIN:
                        return (Key, CallOut, [ idc.Byte(CallOut + j) for j in xrange(currEa-CallOut

                    # If not, be noisy
                    else:
                        print "# 0x%08lx/0x%08lx: found push %s / push %s, not followed by mov %s, a
                        pass
```

```
        # Move forward by one instruction
        currEa = idc.NextHead(currEa, currEa+16)

    # If we didn't find the two PUSH instructions within some
    # specified number, be noisy.
    print "# 0x%08lx: did not find push %s / push %s sequence within %d instructions" % (CallOut, Re
    return (Key, CallOut, [])
```

Now we have all of the information we need to devirtualize X86CALLOUT instructions properly. One final function collates the information for virtualized function entrypoints with the information collected for jump instructions into the VM:

```
# Extract the prologues from all virtualized functions.
def GetPrologues(calloutTargets):
    return map(lambda data: ExtractPrologue(*data),CollateCalloutTargets(calloutTargets))
```

6.4 All Together

First, we run our script to extract the targets of the FinSpy VM X86CALLOUT instructions. This script uses the VM bytecode disassembler, which runs outside of IDA. Thus, we take the output of this script, and copy and paste it into the IDAPython script we developed above to extract function information.

Next, inside of the IDA database for the FinSpy sample, we run our scripts above. They first extract the VM entry instruction information, and then use the X86CALLOUT targets generated by the previous scripts to extract the function prologue and VM entry key for each virtualized function. We print that information out, and copy and paste it into the devirtualization program.

A portion of the output is shown below; the full data set can be seen in the complete Python file:

```
(0x5a4b3a, 0x406a02, [139, 255, 85, 139, 236, 81, 81, 83, 86, 87]),
(0x5a19e6, 0x40171c, [85, 139, 236]),
(0x5a7a1d, 0x408e07, [139, 255, 85, 139, 236, 81, 131, 101, 252, 0]),
(0x5a7314, 0x408810, [139, 255, 85, 139, 236, 81, 81]),
(0x5a8bf9, 0x409a11, [139, 255, 85, 139, 236, 83, 86, 87, 179, 254]),
```

Each tuple contains:

1. The VM instruction key for the first instruction of the virtualized function's body
2. The x86 address of the virtualized function
3. A list of x86 machine code bytes for the virtualized function's prologue

6.5 Issues in Collecting Function Information

The goal of the script detailed in pieces above is to collect the VM instruction keys, junk obfuscation register names, and function prologues for each virtualized function. Ultimately we will use this information to devirtualize X86CALLOUT instructions and prepend the prologues to the devirtualized versions of the virtualized functions. The scripts above assist in substantially automating this process, but it still requires manual intervention if any of its assumptions are violated. We can detect these erroneous situations and manually edit the data. Below we discuss the failure cases for the scripts above.

(The scripts just created are the primary reason my approach is "semi-automated" and not "fully-automated".)

6.5.1 Issue #1: Not All Referenced Callout Targets Were Defined as Code

First, not every address extracted as the target of an X86CALLOUT instruction was defined as code in the IDA database for the FinSpy VM sample. This lead to two issues. First, the script for extracting the prologue for that location would fail. Second, since the subsequent reference to the VM entrypoint was consequently also not defined as code, that reference would not be included in the set of addresses generated by the script for extracting the key and register sequence.

This situation was easy to identify. The script for extracting the prologue would give an error about not being able to find the PUSH instructions for the two obfuscation registers. I inspected the locations for which the prologue script failed, and if that location was not defined as code, I manually defined the locations as code and ran the scripts again. Thereafter, the scripts would properly process these locations automatically.

6.5.2 Issue #2: Not Every Virtualized Function Used the Same VM Entry Schema

Second, the FinSpy VM did not always generate identical schema for x86-to-VM entrypoints at the sites of virtualized functions. Since the FinSpy virtualization obfuscator tool overwrites the original function with a VM entry sequence, small functions might not provide enough space for all parts of the obfuscation sequences. Either there would be no junk obfuscation sequence after the prologue, or the JMP to the VM interpreter would not be obfuscated as an XOR REG, REG / JZ sequence. Also, small functions might not have prologues.

The script to extract virtualized function VM key / obfuscation register information would fail for these functions. Two errors are shown, as well as the corresponding code at the erroneous locations.

```
0x408412: found key 0x5a6d38, but not POP reg32, inspect manually
0x408d17: found key 0x5a7952, but not POP reg32, inspect manually

.text:0040840D         push    5A6D38h
.text:00408412         jmp     GLOBAL__Dispatcher

.text:00408D0C         mov     edi, edi
.text:00408D0E         push    5A7952h
.text:00408D13         push    ecx
.text:00408D14         xor     ecx, ecx
.text:00408D16         pop     ecx
.text:00408D17         jz      GLOBAL__Dispatcher
```

Our ultimate goal with this script is to collect key and prologue information for all virtualized functions. Automation has failed us, because our assumption that each x86-to-VM entrypoint had the same format was wrong. Thus, for these functions, I manually collected the information for these functions. In particular, I added eight manually-created entries to the list generated by the scripts; the first two shown below correspond to the examples above. (For the second entry, the prologue bytes [0x8B, 0xFF] correspond to the "mov edi, edi" instruction on line 0x408D0C above.)

```
(0x5A6D38,0x40840D,[]),
(0x5A7952,0x408D0C,[0x8B,0xFF]),
(0x5A2A19,0x405022,[0x8B,0x65,0xE8,0xC7,0x45,0xC0,0x01,0x00,0x00,0x00,0x83,0x4D,0xFC,0xFF,0x33,0xF6]
(0x5A3FA0,0x4060E6,[]),
(0x5A6841,0x408053,[]),
(0x5A6958,0x408107,[]),
(0x5A6AE9,0x408290,[]),
(0x5A6ED6,0x40851C,[0x8B,0xFF,0x55,0x8B,0xEC]),
```

6.5.3 Issue #3: Not All Callout Targets Were Virtualized Functions

Third, not all of the targets of X86CALLOUT instructions were virtualized -- ten addresses used as X86CALLOUT targets were not virtualized. This included a handful of C runtime functions for string manipulation and exception handling support, but also a few important functions used by the virtualized program. These functions were easy to identify since the prologue extraction script would fail for them. The script issued ten errors, the first three of which are shown:

```
# 0x0040a07c: did not find push ebp / push edx sequence within 20 instructions
# 0x0040ae80: did not find push ebp / push edx sequence within 20 instructions
# 0x0040709e: did not find push ecx / push edx sequence within 20 instructions
```

I inspected these addresses manually, and upon determining that the functions weren't virtualized, I created a Python set called NOT_VIRTUALIZED containing the addresses of these functions.

```
NOT_VIRTUALIZED = set([
0x0040a07c,
0x0040ae80,
0x0040709e,
0x0040ae74,
0x0040aec7,
0x004070d8,
0x00407119,
0x00401935,
0x00408150,
0x00407155,
])
```

Then, in my second devirtualization attempt in Part #3, Phase #4, I used this set to determine whether or not to emit an x86 CALL instruction directly to the referenced target.

7. Conclusion

In Phase #3, we examined FinSpy's mechanism for virtualizing functions and function calls. We determined that we would need several pieces of information to devirtualize these elements correctly, and then wrote scripts to collect the information. We attended to issues that arose in running the scripts, and ended up with the data we needed for devirtualization.

In the next and final phase, Part #3, Phase #4, we will incorporate this information into devirtualization. After fixing a few remaining issues, we will have successfully devirtualized our FinSpy VM sample.
