vb 2023 LONDON

4 - 6 October, 2023 / London, United Kingdom

# DEOBFUSCATING VIRTUALIZED MALWARE USING HEX-RAYS DECOMPILER

Georgy Kucherin

*Kaspersky, Russia*

georgy.kucherin@gmail.com

## ABSTRACT

Code virtualization is one of the most challenging obfuscation techniques. It involves translating code into a custom instruction set that is unknown to reverse engineers. As the process of removing this obfuscation is tedious, advanced threat actors like Lazarus or FinFisher favour protecting their malware with virtualization.

Existing papers on deobfuscating virtualized code rely on creating standalone instruments. The typical workflow of such tools consists of disassembling virtualized code, optimizing it and then converting it to a known architecture such as x86. The deobfuscated code is then loaded into a reverse engineering framework such as *IDA* for further analysis.

In our paper, we present a novel and less arduous approach to defeating code virtualization. Rather than using standalone tools, we rely entirely on *IDA Pro* and *Hex-Rays Decompiler*, two popular reverse engineering instruments. As *Hex-Rays* already implements various code optimization routines, it allows deobfuscation to be performed with much less effort.

We describe our approach step by step, demonstrating how to apply it to FinSpy VM, a malware obfuscator commonly discussed in papers on code devirtualization. First, we introduce features of the *IDA* SDK that we use for automating deobfuscation. Then, we explain how to translate virtualized code into the x86 architecture using the disassembler API. Finally, we detail how to harness the *Hex-Rays* microcode to decompile the translated assembly into C and thus obtain clean devirtualized code. While describing the deobfuscation process, we will provide multiple recommendations on how to efficiently use the scripting capabilities of *IDA* and *Hex-Rays Decompiler*.

The commented code of the deobfuscator is released along with the paper. It can be used as a template for working with other virtualized malware.

## INTRODUCTION

While analysing highly sophisticated malware, it is common to encounter obfuscation techniques that significantly delay the process of reverse engineering. An example of such a technique is code virtualization. The goal of virtualization is to translate code from a commonly known architecture (such as x86) to a completely exotic one. Specifically, in order to virtualize a binary (e.g. a *Windows* executable file), malware developers usually perform the following procedure:

1. Design a virtual processor architecture. This architecture may have any number of instructions or registers. The only requirement is that the custom architecture should be able to compute any x86 function.
2. Create a translator program that converts functions located in the executable file from x86 to the virtual architecture.
3. Develop a program that emulates the virtual architecture, i.e. executes virtual architecture instructions on an x86 processor.
4. Embed the emulator program along with the virtualized code into an executable file and make the emulator launch on startup of this executable.

As a result of these actions, the executable produced at step 4 will perform the same functionalities as the original one. However, the presence of the virtual architecture will make the obfuscated executable much more difficult to analyse.

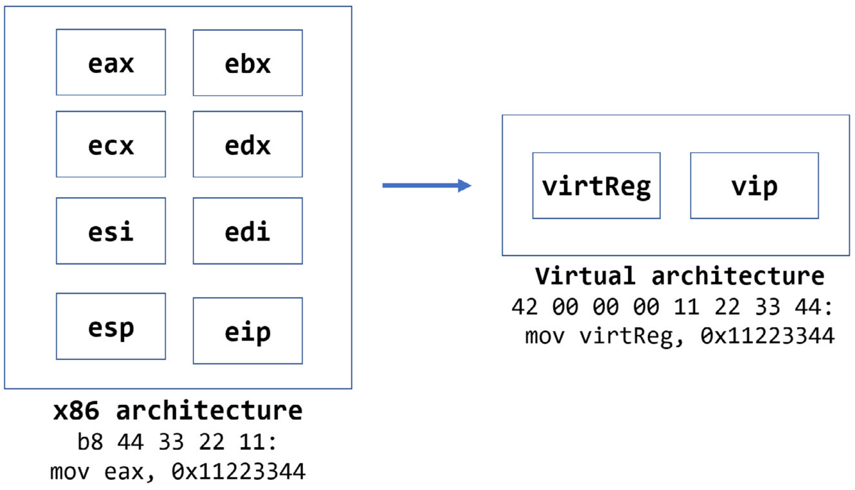


*Figure 1: Illustration of the obfuscation process. An x86 instruction is translated to an architecture with a virtual instruction pointer (vip) and a general-purpose register (virtReg).*

In order to analyse a virtualized executable, a malware analyst usually needs to perform the following actions:

1. Reverse engineer the emulator program in order to understand the inner workings of the virtual architecture. For example, in the case demonstrated in Figure 1, an analyst would need to deduce that the instruction with bytes `42 00 00 00 11 22 33 44` assigns the value `0x11223344` to a virtual register).

2. Convert the code of virtualized functions back to a known architecture (e.g. the instruction `mov virtReg, 0x11223344` from Figure 1 can be converted to the x86 instruction `mov eax, 0x11223344`).
3. Optimize the converted x86 code to make it easier to analyse with commonly used tools such as *IDA*.

Existing papers on malware devirtualization ([1, 2]) propose performing steps 2 and 3 by creating standalone scripts or even fully fledged binary analysis frameworks. As an alternative, it is possible to perform all the deobfuscation steps with just *IDA Pro* and the *Hex-Rays Decompiler*, two instruments commonly used for reverse engineering. Such an approach has the following advantages:

- The automation capabilities of *IDA Pro* provide an API that can be used for efficiently performing operations with assembly instructions.
- The *Hex-Rays Decompiler* implements various optimization algorithms, thus using it will simplify the optimization step.

In the following sections of this paper, we will demonstrate this approach and its efficiency by applying it to FinSpy VM, an obfuscator that is used in the FinFisher spyware and commonly discussed in papers on devirtualization.

## OVERVIEW OF FINSPY VM

The sample of FinSpy that we will be analysing has the SHA256 hash `94ABF6DF38F26530DA2864D80E1A0B7CDFCE63FD27B142993B89C52B3CEE0389`. It is available for free download from vx-underground [4] – we recommend that the reader downloads the sample and follows along with the instructions in this paper.

Before describing the devirtualization method itself, we will need first to study the internals of the FinSpy VM obfuscator. As information about this obfuscator is already published in existing papers, we will provide only the most essential information that is crucial for understanding the rest of this paper. Interested readers can refer to [1, 2, 3] for more detailed descriptions of the obfuscator's inner workings.

One of the obfuscated functions in this binary is located at the address `0x405362`. It starts with the following code:

```
.text:00405362 loc_405362:    mov    edi, edi
.text:00405364                push   ebp
.text:00405365                mov    ebp, esp
.text:00405367                sub    esp, 0C7Ch
.text:0040536D                push   ebx
.text:0040536E                push   esi
.text:0040536F                push   edi
.text:00405370                push   735FBBCh
.text:00405375                push   edx
.text:00405376                xor    edx, edx
.text:00405378                pop    edx
.text:00405379                jz     loc_401930
```

As can be observed from the code above, it performs the following:

- Allocates space on the function stack.
- Pushes registers and a 32-bit value (`0x735FBBC`) on the stack. This value identifies the first virtual instruction that is to be executed.
- Performs a jump to the virtual machine emulator (at address `0x4005379`).

As for the code of the emulator to which the jump is taken, it is as follows:

```
.text:00401930 loc_401930:
.text:00401930                jz     loc_401EB8
.text:00401936                jnz    loc_401EB8

.text:00401EB8 loc_401EB8:
.text:00401EB8                pusha
.text:00401EB9                jp     loc_40193E
.text:00401EBF                jnp    loc_40193E
```

…

As can be observed from this code, pairs of instructions at addresses `0x00401930 – 0x00401936` and `0x00401EB9 – 0x00401EBF` contain conditional jumps. They lead to the same destination, and their conditions are opposite. Thus, these jumps implement an obfuscation technique called opaque predicates [5]. To combat it, we can replace each pair of conditional jumps with an unconditional jump (e.g. replace the `jz` and `jnz` instructions at `0x401930` with the `jmp loc_401EB8` instruction). This can be done with a Python script authored by Rolf Rolles [6]. A version of this script that works with latest versions of *IDA* can be found at [7].

In order for the script to work, it should be placed in the `plugins` directory of the *IDA* installation. Once *IDA* is restarted, the opaque predicate obfuscation will be eliminated. This in turn will allow a function to be defined at address `0x401930` (which is called at address `0x405379` in the disassembly listing).

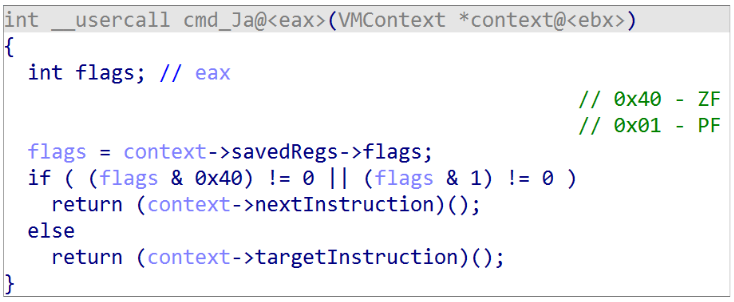
An analysis of this function reveals the following details about FinSpy VM:

- The obfuscated binary embeds compressed (with APLib) and encrypted (with 4-byte XOR) bytecode of the virtual machine (in the analysed sample this bytecode is contained at address `0x40A0A8`).
- The decrypted and decompressed bytecode contains 24-byte virtual instructions. The bytes of each instruction contain the following information:
    - A unique identifier of the instruction (4 bytes)
    - Instruction type (1 byte)
    - Size of instruction operands (1 byte)
    - Relocation data (2 bytes)
    - Instruction operands (16 bytes).

```
[CC B5 33 A2] [12] [01] [00 00] [60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
[CA B5 33 A2] [12] [02] [00 00] [33 C9 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
[C0 B5 33 A2] [12] [06] [00 00] [81 E9 00 87 85 87 00 00 00 00 00 00 00 00 00 00]
[C2 B5 33 A2] [18] [00] [00 00] [01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
[61 2E 30 A2] [1D] [00] [00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
```

*Figure 2: Snippet of FinSpy VM decrypted bytecode, with the components listed above separated by square brackets.*

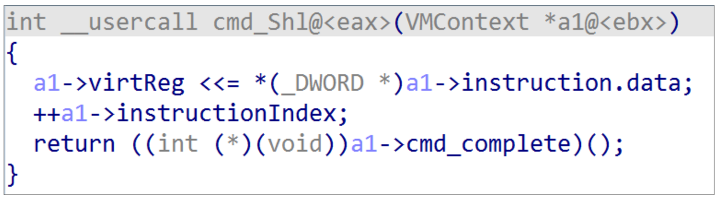
- The obfuscator implements 34 handler functions, one for each instruction type. These handlers are stored in a table (at address `0x4028B3`). Each handler is responsible for interpreting the operands of the instruction and then executing it.
- Implemented instructions can be divided into three types:
    - Instructions performing various jumps, either conditional or unconditional. Conditional jump instructions use bits of the EFLAGS register to check where a jump should be taken.



```
int __usercall cmd_Ja@<eax>(VMContext *context@<ebx>)
{
  int flags; // eax
                                                  // 0x40 - ZF
                                                  // 0x01 - PF
  flags = context->savedRegs->flags;
  if ( (flags & 0x40) != 0 || (flags & 1) != 0 )
    return (context->nextInstruction)();
  else
    return (context->targetInstruction)();
}
```

*Figure 3: A handler implementing the ja (jump if above) instruction, which checks the ZF and PF flags.*

- Instructions performing various operations with a virtual register, that we will refer to as `virtReg`. Such instructions are used for moving data from x86 registers to the virtual register and vice versa, as well as performing arithmetic operations (such as addition or bit shifting).



```
int __usercall cmd_Shl@<eax>(VMContext *a1@<ebx>)
{
  a1->virtReg <<= *(_DWORD *)a1->instruction.data;
  ++a1->instructionIndex;
  return ((int (*)(void))a1->cmd_complete)();
}
```

*Figure 4: A handler implementing the shl virtReg, imm8 instruction that performs a left shift of bits in the virtual register.*

- Instructions executing native x86 code (examples of executed x86 instructions include `test` and `ret`).

Now that we have an outline of how the FinSpy VM obfuscator works and what virtual instructions it implements, we can describe the deobfuscation process itself.

## TRANSLATING VIRTUAL INSTRUCTIONS TO X86

As we have discussed in the introduction, the next step after analysing the virtual machine internals is to translate virtual instructions into x86 ones. We have three types of instructions, and translation of two of these types is straightforward:

- Instructions executing native x86 code do not need to be translated at all.
- Instructions performing jumps (such as `ja`) can be replaced with their x86 equivalents.

The greatest difficulty arises with translating instructions that use the `virtReg` register. While translating such instructions (e.g. the `shl virtReg, imm8` instruction), we need to find a place where we can store the value of the virtual register. There are two options for that storage:

1. An x86 register (such as `eax`);
2. An in-memory global variable.

If we choose the first option, the instruction `shl virtReg, imm8` would be translated as `shl eax, imm8`. In the case of the second option, the translation will be `shl dword ptr [virtReg], imm8`.

To decide which option to choose, we can take a look at the following set of virtual instructions:

```
exec {mov eax, 0x3}     ; this instruction executes native x86 code
mov virtReg, 0x400300   ; this instruction moves a constant into the virtual register
mov [virtReg], eax      ; this instruction writes data to an address stored in
                        ; the virtual register
```

If we execute these three instructions, the value `0x3` will be written to the address `0x400300`. If we choose to use option 1 and replace the virtual register with `eax`, the translation will be as follows:

```
mov eax, 0x3
mov eax, 0x400300
mov dword ptr [eax], eax.
```

However, this translation is not equivalent to the virtualized code as it moves the value `0x400300` (instead of `0x3`) to the address `0x400300`. Thus, it is not enough to simply replace the virtual register with an x86 one, and option 1 is not suitable for us. Thus, we are left with the second option.

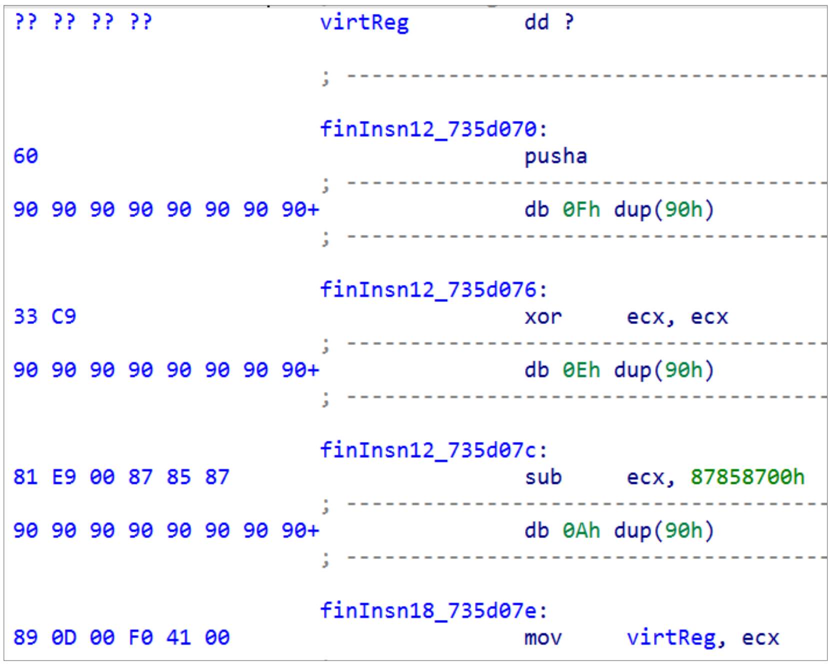
Now that we have decided to store the `virtReg` register in memory, we can provide equivalent x86 code for virtual instructions. We provide examples of such translations below, while all translations can be examined at [7].

| **Virtual instruction** | **Instruction description** | **x86 translation** |
|---|---|---|
| `shl virtReg, imm8` | Performs a left shift of the bits in the virtual register | `shl dword ptr [virtReg], imm8` |
| `add virtReg, imm32` | Adds a constant to the virtual register | `add dword ptr [virtReg], imm32` |
| `mov virtReg, [virtReg]` | Retrieves a DWORD located at an address in the virtual register and stores it in the virtual register. | `push eax`<br>`mov eax, [virtReg]`<br>`mov eax, [eax]`<br>`mov [virtReg], eax`<br>`pop eax` |

Once we compose an x86 translation for every virtual instruction, we are able to:

- Iterate over every virtual instruction in the bytecode, translating it to x86.
- Write the bytes of translated instructions into the IDB (*IDA* database) of our sample.

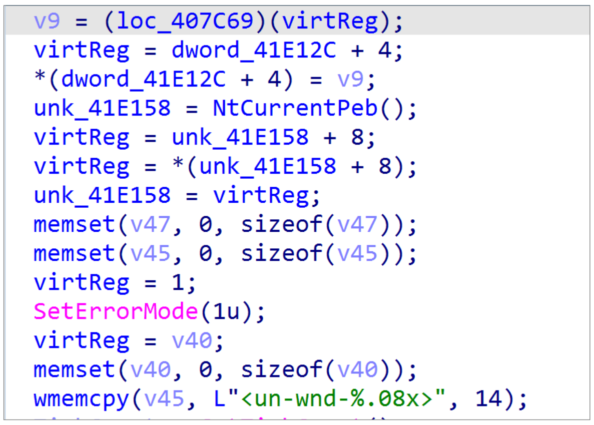
It is best to store these bytes in a separate segment that can be created using the `add_segm_ex` API function [8]. We can then write data to this segment using the `put_bytes` [9] function. We will additionally use this segment to store the value of the `virtReg` resister. As an effect, we get the following results:



```
?? ?? ?? ??              virtReg         dd ?

                         ; ---------------------------------------

                         finInsn12_735d070:
60                                       pusha
                         ; ---------------------------------------
90 90 90 90 90 90 90 90+                 db 0Fh dup(90h)
                         ; ---------------------------------------

                         finInsn12_735d076:
33 C9                                    xor     ecx, ecx
                         ; ---------------------------------------
90 90 90 90 90 90 90 90+                 db 0Eh dup(90h)
                         ; ---------------------------------------

                         finInsn12_735d07c:
81 E9 00 87 85 87                        sub     ecx, 87858700h
                         ; ---------------------------------------
90 90 90 90 90 90 90 90+                 db 0Ah dup(90h)
                         ; ---------------------------------------

                         finInsn18_735d07e:
89 0D 00 F0 41 00                        mov     virtReg, ecx
```

*Figure 5: Snippet of the translated x86 code. The dup(90h) bytes represent nop instructions that have been placed for purposes of padding.*

Once the segment is filled with assembled code, we can patch the obfuscated function at address `0x405379`, replacing the jump to the virtual machine emulator with a jump to the created segment (if the reader launches the script from [7], the target address will be `0x432744`). Once this jump is patched, we get the following decompilation results:



```
v9 = (loc_407C69)(virtReg);
virtReg = dword_41E12C + 4;
*(dword_41E12C + 4) = v9;
unk_41E158 = NtCurrentPeb();
virtReg = unk_41E158 + 8;
virtReg = *(unk_41E158 + 8);
unk_41E158 = virtReg;
memset(v47, 0, sizeof(v47));
memset(v45, 0, sizeof(v45));
virtReg = 1;
SetErrorMode(1u);
virtReg = v40;
memset(v40, 0, sizeof(v40));
wmemcpy(v45, L"<un-wnd-%.08x>", 14);
```

*Figure 6: Snippet of the decompilation results.*

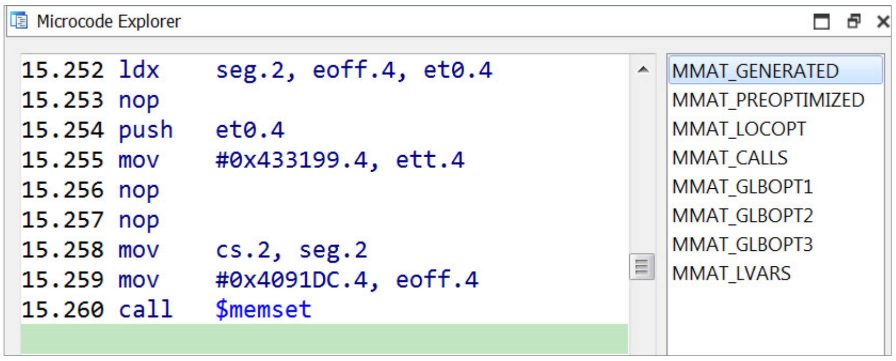
Although this code is readable, the decompilation results are far from satisfactory. That is because they contain dead store assignments to the `virtReg` variable, such as the assignment of the constant `1` in this snippet:

```
virtReg = 1;
SetErrorMode(1u);
virtReg = v40;
```

To make sure that the decompilation results are correct in every case, the *Hex-Rays Decompiler* does not perform dead store elimination with global variables. However, in our case, we need to force the decompiler to perform such optimizations. To do that, we will need to study the internals of the *Hex-Rays Decompiler* and then use a novel optimization technique.

## ANALYSING MICROCODE OF THE DECOMPILED FUNCTION

In order to modify decompilation results, we can use an API that is referred to as *microcode API*. It allows us to interact with microcode – an intermediate language between assembly code and C pseudocode. In order to look at a function's microcode, we can use the Lucid plug-in [10]. When this plug-in is installed, we can press Ctrl-Shift-M in the decompiler window to view the microcode:



```
15.252 ldx    seg.2, eoff.4, et0.4
15.253 nop
15.254 push   et0.4
15.255 mov    #0x433199.4, ett.4
15.256 nop
15.257 nop
15.258 mov    cs.2, seg.2
15.259 mov    #0x4091DC.4, eoff.4
15.260 call   $memset
```

*Figure 7: Microcode Explorer window.*

The microcode itself is displayed on the left, while the panel on the right allows the selection of one of multiple so-called *maturity levels*. The microcode at the lowest maturity level (`MMAT_GENERATED`) looks like assembly code, while the microcode at its highest maturity level (`MMAT_LVARS`) is most close to pseudocode.

We will be looking at the microcode at the `MMAT_PREOPTIMIZED` level. At this level, redundant assignments to the `virtReg` variable look as follows:

```
1.43 mov      #1.4, $virtReg.4
1.44 call     $SetErrorMode <std:"UINT uMode" #1.4>.0
1.45 mov      &(_C).4, $virtReg.4
```

As can be seen, the `virtReg` variable is contained in the operands of the `mov` instruction. We can further examine microcode instructions by representing them in the form of a graph:

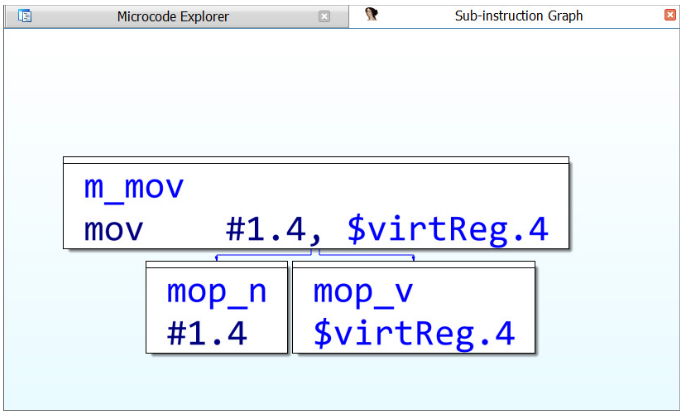


*Figure 8: Graph representing the mov #1.4, $virtReg.4 instruction.*

As can be seen from the graph above, this instruction has an operand with type `mop_v`, which is the `virtReg` variable, as well as an operand with type `mop_n`, which is the constant `1`. In order to optimize code and make redundant assignments to the `virtReg` variable disappear, we need to iterate through all operands of all microcode instructions. For every operand, we need to check if it is the `virtReg` variable. If so, we need to replace it with another object that allows optimizations to be performed on it.

One such 'optimizable' object that the decompiler implements is called a *kernel register* [11], which is an object similar to a processor register. While there is a limited number of x86 processor registers (`eax`, `ebx`, `ecx`, etc.), it is possible to use any number of kernel registers in a function's microcode. Thus, in order to perform the optimizations that we need, we can create a kernel register and replace every occurrence of the `virtReg` variable with this created register.

## REPLACING THE VIRTUAL REGISTER VARIABLE WITH A KERNEL REGISTER

As we have discussed above, we need to iterate over all instructions in the microcode to perform replacements of their operands. With *Hex-Rays* API, operand iteration can be performed by using *decompiler hooks*. A decompiler hook is a class that is derived from `ida_hexrays.Hexrays_Hooks`. To be able to interact with the microcode at the `MMAT_PREOPTIMIZED` level, we need to override the `preoptimized` method in the hook class. In this method, we need to:

- Allocate a kernel register;
- Invoke a replacer that will iterate over all operands and search for the `virtReg` variable.

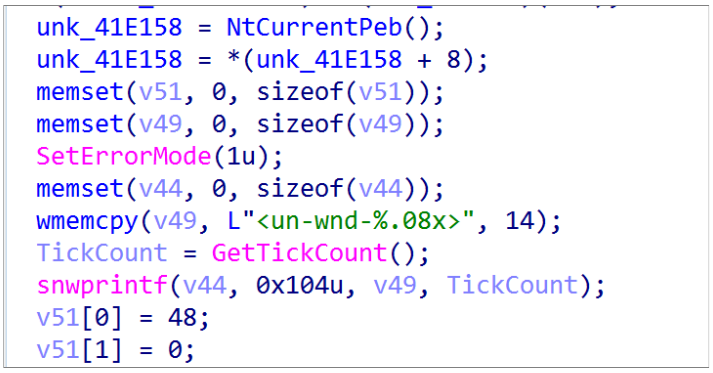
This can be performed with the following code:

```
class DecompilerHook(ida_hexrays.Hexrays_Hooks):
    def preoptimized(self, *args):
        global kernel_register
        mba = args[0] # MBA means "micro block array". This object contains the
                      # microcode of the function being decompiled
        kernel_register = mba.alloc_kreg(4) # We allocate a kernel register here
        if kernel_register != ida_hexrays.mr_none:
            repl = OperandReplacer()
            mba.for_all_ops(repl) # Invoke a class that iterates over operands
        return 0
event_hook = decompiler_hook()
event_hook.hook() # Activate the hook
```

As for the replacer, it is a class derived from `ida_hexrays.mop_visitor_t`. In this class, we should define a method called `visit_mop` that will check the operand type and replace it if needed:

```
class OperandReplacer(ida_hexrays.mop_visitor_t):
    def visit_mop(self, op, op_type, is_target):
        global kernel_register
        # Check if we are dealing with the virtual register variable
        if op.t == ida_hexrays.mop_v and op.g == register_addr
            # Replace the global variable with a kernel register
            op.make_reg(kernel_register, op.size)
            if self.blk:
                      self.blk.mark_lists_dirty() # inform IDA that the microcode changed
        return 0
```

When the replacement is complete, the decompiled code will not contain the `virtReg` variable. As can be observed in Figure 9, the code is clean of any obfuscations, which means that we have successfully performed devirtualization of FinSpy VM.



```
unk_41E158 = NtCurrentPeb();
unk_41E158 = *(unk_41E158 + 8);
memset(v51, 0, sizeof(v51));
memset(v49, 0, sizeof(v49));
SetErrorMode(1u);
memset(v44, 0, sizeof(v44));
wmemcpy(v49, L"<un-wnd-%.08x>", 14);
TickCount = GetTickCount();
snwprintf(v44, 0x104u, v49, TickCount);
v51[0] = 48;
v51[1] = 0;
```

*Figure 9: Snippet of the deobfuscated code.*

## CONCLUSION

In this paper, we described how the automation capabilities of *IDA Pro* and *Hex-Rays Decompiler* can be used to devirtualize functions protected by the FinSpy VM obfuscator. To perform devirtualization, we composed equivalent x86 code for every virtual instruction. We then created a segment in *IDA* where we placed the generated code. Finally, we improved the quality of decompilation by replacing occurrences of the virtual register global variable with kernel registers. Thanks to the vast automation capabilities of *IDA Pro*, we were able to efficiently develop the deobfuscator code. The devirtualizer implementation can be considered relatively compact as it contains fewer than 500 lines of code.

## ACKNOWLEDGEMENTS

We would like to thank Sergey Belov, Principal Security Researcher at *Kaspersky*, who first discovered the optimization technique involving kernel registers and provided invaluable assistance with the analysis of FinFisher spyware.

## REFERENCES

[1] Kafka, F. ESET's guide to deobfuscating and devirtualizing FinFisher. ESET. January 2018. https://web-assets.esetstatic.com/wls/2018/01/WP-FinFisher.pdf.

[2] Rolles, R. A walk-through tutorial, with code, on statically unpacking the FinSpy VM: part one, x86 deobfuscation. Möbius Strip Reverse Engineering. January 2018. https://www.msreverseengineering.com/blog/2018/1/23/a-walk-through-tutorial-with-code-on-statically-unpacking-the-finspy-vm-part-one-x86-deobfuscation.

[3] Allievi, A.; Florio, E. FinFisher exposed: A researcher's tale of defeating traps, tricks, and complex virtual machines. Microsoft. March 2018. https://www.microsoft.com/en-us/security/blog/2018/03/01/finfisher-exposed-a-researchers-tale-of-defeating-traps-tricks-and-complex-virtual-machines/.

[4] Link to the analysed FinSpy sample. https://samples.vx-underground.org/root/APTs/2015/2015.10.15/Samples/94abf6df38f26530da2864d80e1a0b7cdfce63fd27b142993b89c52b3cee0389.7z.

[5] Wikipedia. Opaque predicate. https://en.wikipedia.org/wiki/Opaque_predicate.

[6] Rolles, R. Opaque predicate removal script. https://github.com/RolfRolles/FinSpyVM/blob/master/FinSpyDeob.py.

[7] Devirtualizer repository. https://github.com/gkucherin/finspy_devirtualizer.

[8] Documentation of the add_segm_ex function. https://hex-rays.com/products/ida/support/idadoc/299.shtml.

[9] Documentation of the put_bytes function. https://www.hex-rays.com/products/ida/support/idapython_docs/ida_bytes.html.

[10] Lucid plugin for Hex-Rays decompiler. https://github.com/gaasedelen/lucid.

[11] Hex-Rays API documentation. https://www.hex-rays.com/products/ida/support/idapython_docs/ida_hexrays.html.