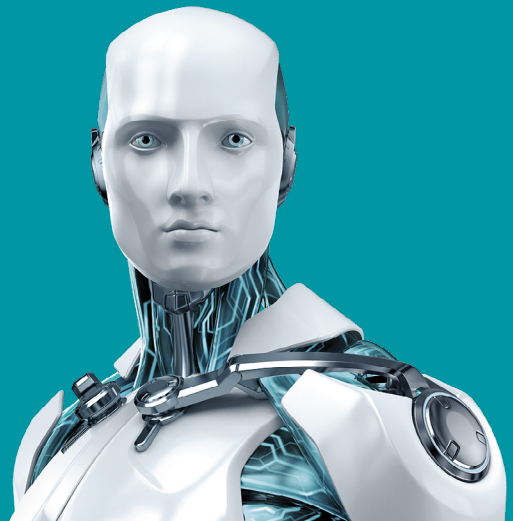
ESET Research Whitepapers // January 2018 // Author: Filip Kafka

ESET'S GUIDE TO DEOBFUSCATING AND DEVIRTUALIZING FINFISHER



ENJOY SAFER TECHNOLOGY™

CONTENTS

Introduction **3**

Anti-disassembly **4**

FinFisher's virtual machine **7**

Terms and definitions 8

Vm_start 8

FinFisher's interpreter **10**

1. Creating an IDA graph 10

2. Vm_dispatcher 11

3. Vm_context 12

4. Virtual instruction implementations – vm_handlers 14

5. Writing your own disassembler 17

6. Understanding the implementation of this virtual machine 19

7. Automating the disassembly process for more FinFisher samples 20

8. Compiling disassembled code without the VM 20

Conclusion **22**

Appendix A: IDA Python script for naming FinFisher vm_handlers **23**

INTRODUCTION

Thanks to its strong anti-analysis measures, the FinFisher spyware has gone largely unexplored. Despite being a prominent surveillance tool, only partial analyses have been published on its more recent samples.

Things were put in motion in the summer of 2017 with ESET's analysis of FinFisher surveillance campaigns that ESET had discovered in several countries. In the course of our research, we have *identified campaigns where internet service providers* most probably played the key role in compromising the victims with FinFisher.

When we started thoroughly analyzing this malware, the main part of our effort was overcoming FinFisher's anti-analysis measures in its Windows versions. The combination of advanced obfuscation techniques and proprietary virtualization makes FinFisher very hard to de-cloak.

To share what we learnt in de-cloaking this malware, we have created this guide to help others take a peek inside FinFisher and analyze it. Apart from offering practical insight into analyzing FinFisher's virtual machine, the guide can also help readers to understand virtual machine protection in general – that is, proprietary virtual machines found inside a binary and used for software protection. We will not be discussing virtual machines used in interpreted programming languages to provide compatibility across various platforms, such as the Java VM.

We have also analyzed Android versions of FinFisher, whose protection mechanism is based on an open source LLVM obfuscator. It is not as sophisticated or interesting as the protection mechanism used in the Windows versions, thus we will not be discussing it in this guide.

Hopefully, experts from security researchers to malware analysts will make use of this guide to better understand FinFisher's tools and tactics, and to protect their customers against this omnipotent security and privacy threat.

ANTI-DISASSEMBLY

When we open a FinFisher sample in IDA Pro, the first protection we notice in the main function is a simple, yet very effective, anti-disassembly trick.

FinFisher uses a common anti-disassembly technique – hiding the execution flow by replacing one unconditional jump with two complementary, conditional jumps. These conditional jumps both target the same location, so regardless of which jump is made, the same effective code execution flow results. The conditional jumps are then followed by garbage bytes. These are meant to misdirect the disassembler, which normally will not recognize that they are dead code, and will steam on, disassembling garbage code.

What makes this malware special is the way in which it uses this technique. In most other malware we've analyzed, it is only used a few times. FinFisher, however, uses this trick after every single instruction.

This protection is very effective at fooling the disassembler – many parts of the code aren't disassembled properly. And of course, it is impossible to use the graph mode in IDA Pro. Our first task will be to get rid of this anti-disassembly protection.

The code was clearly not obfuscated manually but with an automated tool and we can observe a pattern in all the jump pairs.

There are two different types of jump pairs – near jump with a 32-bit offset and short jump with an 8-bit offset.

The opcodes of both conditional near jumps (with a dword as a jump offset) start with a 0x0F byte; while the second bytes are equal to 0x8?, where ? in both jump instructions differs only by 1 bit. This is because x86 opcodes for complementary jumps are numerically consecutive. For example, this obfuscation scheme always pairs JE with JNE (0x0F 0x84 vs 0x0F 0x85 opcodes), JP with JNP (0x0F 0x8A vs 0x0F 0x8B opcodes), and so on.

These opcodes are then followed by a 32-bit argument specifying the offset to the destination of the jump. Since the size of both instructions is 6 bytes, the offsets in two consequent jumps differ exactly by 6. (Figure 1)

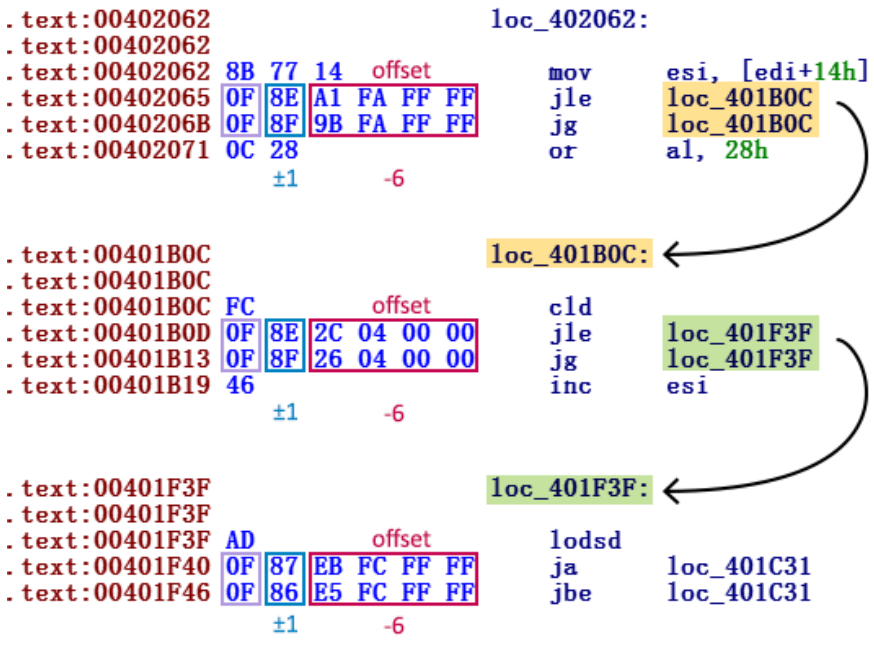


Figure 1 // Screenshot showing instructions followed by two conditional near jumps every time

For example, the code below can be used to detect two of these consecutive conditional jumps:

```
def is_jump_near_pair(addr):
    jcc1 = Byte(addr+1)
    jcc2 = Byte(addr+7)
    # do they start like near conditional jumps?
    if Byte(addr) != 0x0F || Byte(addr+6) != 0x0F:
       return False
    # are there really 2 consequent near conditional jumps?
    if (jcc1 & 0xF0 != 0x80) || (jcc2 & 0xF0 != 0x80):
       return False
    # are the conditional jumps complementary?
    if abs(jcc1-jcc2) != 1:
       return False
    # do those 2 conditional jumps point to the same destination?
    dst1 = Dword(addr+2)
    dst2 = Dword(addr+8)
    if dst1-dst2 != 6:
       return False
    return True
```

Deobfuscation of short jumps is based on the same idea, only the constants are different.

The opcode of a short conditional jump equals 0x7?, and is followed by one byte – the jump offset. So again, when we want to detect two consecutive, conditional near jumps, we have to look for opcodes: 0x7?; offset; 0x7? ± 1; offset -2. The first opcode is followed by one byte, which differs by 2 in two consequent jumps (which is, again, the size of both instructions). (Figure 2)

For example, this code can be used to detect two conditional short jumps:

```
def is_jcc8(b):
    return b&0xF0 == 0x70
def is_jump_short_pair(addr):
    jcc1 = Byte(addr)
    jcc2 = Byte(addr+2)
    if not is_jcc8(jcc1) || not
        is_jcc8(jcc2):
        return False
    if abs(jcc2-jcc1) != 1:
        return False
    dst1 = Byte(addr+1)
    dst2 = Byte(addr+3)
    if dst1 - dst2 != 2:
        return False
    return True
```

After detecting one of these conditional jump pairs, we deobfuscate this code by patching the first conditional jump to unconditional (using the 0xE9 opcode for the near jump pairs and 0xEB for the short jump pairs) and patch the rest of the bytes with NOP instructions (0x90)

```
def patch_jcc32(addr):
    PatchByte(addr, 0x90)
    PatchByte(addr+1, 0xE9)
    PatchWord(addr+6, 0x9090)
    PatchDword(addr+8,
        0x90909090)
def patch_jcc8(addr):
    PatchByte(addr, 0xEB)
    PatchWord(addr+2, 0x9090)
```

In addition to these two cases, there might be some places where a jump pair consists of a short and a near jump, rather than two jumps of the same category. However, this only occurs in a few cases in the FinFisher samples and can be fixed manually.

With these patches made, IDA Pro starts to "understand" the new code and is ready (or at least almost ready) to create a graph. It may be the case that we still need to make one more improvement: append tails, i.e. assign the node with the destination of the jump to the same

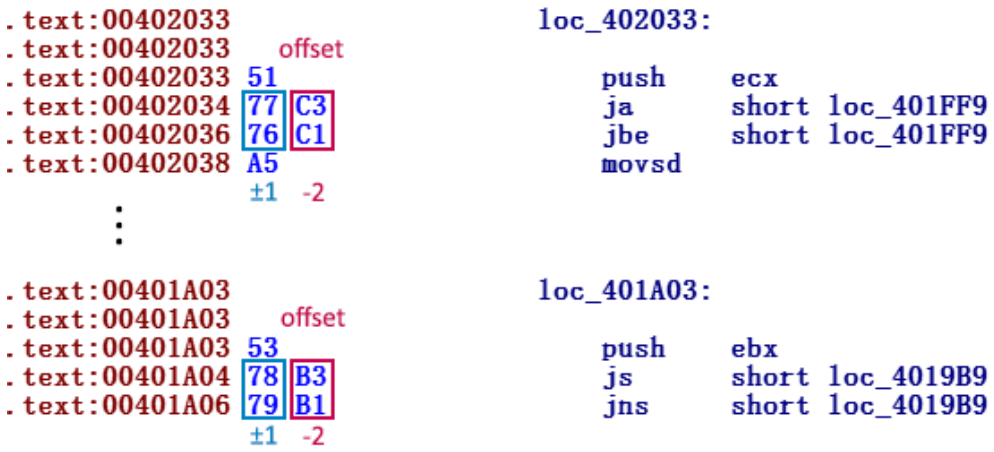


Figure 2 // Examples of instructions followed by two conditional short jumps every time

graph where the node with the jump instruction is located. For this, we can use the IDA Python function `append_func_tail`.

The last step of overcoming the anti-disassembly tricks consists of fixing function definitions. It may still occur that the instruction after the jumps is `push ebp`, in which case IDA Pro (incorrectly) treats this as the beginning of a function and creates a new function definition. In that case, we have to remove the function definition, create the correct one and append tails again.

This is how we can get rid of FinFisher's first layer of protection – anti-disassembly.

FINFISHER'S VIRTUAL MACHINE

After a successful deobfuscation of the first layer, we can see a clearer main function whose sole purpose is to launch a custom virtual machine and let it interpret the bytecode with the actual payload.

As opposed to a regular executable, an executable with a virtual machine inside uses a set of virtualized instructions, rather than directly using the instructions of the processor. Virtualized instructions are executed by a virtual processor, which has its own structure and does not translate the bytecode into a native machine code. This virtual processor as well as the bytecode (and virtual instructions) are defined by the programmer of the virtual machine. (Figure 3)

As mentioned in the introduction, a well-known example of a virtual machine is the Java Virtual Machine. But in this case, the virtual machine is inside the binary, so we are dealing with a virtual machine used for a protection against reverse engineering. There are well-known commercial virtual machine protectors, for example VMProtect or Code Virtualizer.

The FinFisher spyware was compiled from source code and the compiled binary was then protected with a virtual machine at the assembly level. The protection process includes translating instructions of the original binary into virtual instructions and then creating a new binary that contains the bytecode and the virtual CPU. Native instructions from the original binary are lost. The protected, virtualized sample must have the same behavior as a non-protected sample.

To analyze a binary protected with a virtual machine, one needs to:

1. Analyze the virtual CPU.
2. Write one's own disassembler for this custom virtual CPU
   and parse the bytecode.
3. Optional step: compile the disassembled code into a binary file to
   get rid of the virtual machine.

The first two tasks are very time-consuming, and the first one can also get quite difficult. It includes analyzing every *vm_handler* and understanding how registers, memory access, calls, etc. are translated.

Figure 3 // Bytecode interpreted by the virtual CPU

Terms and definitions

There is no standard for naming particular parts of a virtual machine. Hence, we will define some terms which will be referenced throughout the whole paper.

- Virtual machine (vm) – custom, virtual CPU; contains parts like the *vm_dispatcher*, *vm_start*, *vm_handlers*
- *vm_start* – the initialization part; memory allocation and decryption routines are executed here
- Bytecode (also known as pcode) – virtual opcodes of *vm_instructions* with their arguments are stored here
- *vm_dispatcher* – fetches and decodes virtual opcode; is basically a preparation for the execution of one of the vm_handlers
- *vm_handler* – an implementation of a *vm_instruction*; executing one *vm_handler* means executing one *vm_instruction*
- Interpreter (also known as *vm_loop*) – *vm_dispatcher* + *vm_handlers* – the virtual CPU
- Virtual opcode – an analog of the native opcode
- *vm_context (vm_structure)* – an internal structure used by the interpreter
- *vi_params* – a structure in the *vm_context* structure; the virtual instruction parameters, used by the *vm_handler*; it includes the *vm_opcode* and arguments

When interpreting the bytecode, the virtual machine uses a virtual stack and a single virtual register.

- *vm_stack* – an analog of a native stack, which is used by the virtual machine
- *vm_register* – an analog of a native register, used by this virtual machine; further referenced as tmp_REG
- *vm_instruction* – an instruction defined by developers of vm; the body (the implementation) of the instruction is called its *vm_handler*

In the following sections, we will describe the parts of the virtual machine in more technical detail and explain how to analyze them.

A deobfuscated graph of the main malware function consists of three parts – an initialization part and two other parts which we have named *vm_start* and interpreter (*vm_dispatcher* + *vm_handlers*).

The initialization part specifies a unique identifier of what could be interpreted as a bytecode entry point, and pushes it on the stack. Then, it jumps to the vm_start part that is an initialization routine for the virtual machine itself. It decrypts the bytecode and passes control to the *vm_dispatcher* that loops over the virtual instructions of the bytecode and interprets them using the *vm_handlers*.

The *vm_dispatcher* starts with a pusha instruction and ends with a `jmp dword ptr [eax+ecx*4]` instruction (or similar), which is a jump to the relevant *vm_handler*.

Vm_start

The graph created after the deobfuscation of the first layer is seen in Figure 4. The *vm_start* part is not so important for the analysis of the interpreter. However, it can help us understand the whole implementation of the vm; how it uses and handles virtual flags, virtual stack, etc. The second part – the *vm_dispatcher* with *vm_handlers* – is the crucial one.

The *vm_start* is called from almost every function, including the main function. The calling function always pushes a virtual instruction identifier and then it jumps to *vm_start*. Every virtual instruction has its own virtual identifier. In this example, the identifier of the virtual entry point, where the execution from the main function starts, is 0x21CD0554. (Figure 5)

In this part, there is a lot of code for preparing the *vm_dispatcher* – mainly for preparing the bytecode and allocating memory for the

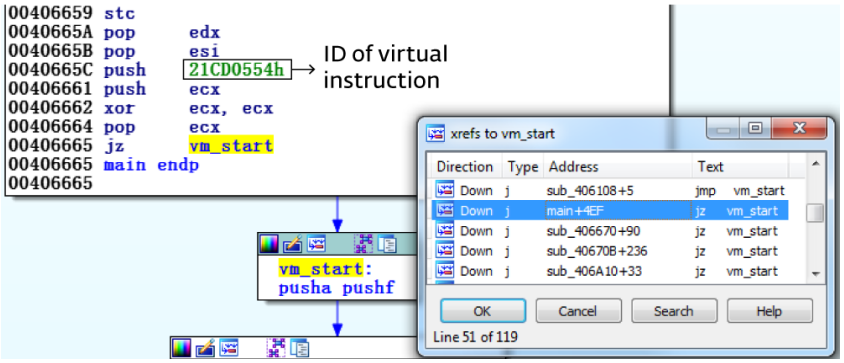


Figure 5 // *vm_start* is called from each of the 119 virtualized functions. The ID of the first virtual instruction of the respective function is given as an argument.

whole interpreter. The most important parts of the code do the following:

1. Allocate 1MB with RWX permission for bytecode and a few more variables.
2. Allocate 0x10000 bytes RWX for local variables in the virtual machine for the current thread – the *vm_stack*.
3. Decrypt a piece of code using an XOR decryption routine. The decrypted code is an aPLib unpacking routine.

   The XOR decryption routine used in the sample is a slightly modified version of XOR dword, key routine. Actually, it skips the first of the six dwords and then XORs only the remaining 5 dwords with the key. Following is the algorithm for the routine (further referred to as XOR decryption_code):

```
int array[6];
int key;
for (i = 1; i < 6; i++) {
       array[i] ^= key;
}
```

4. Call aPLib unpacking routine to unpack bytecode. After unpacking, virtual opcodes are still encrypted. (Figure 6)

Preparing virtual opcodes (step 1, 3 and 4) is done only once – at the beginning – and is skipped in subsequent executions of *vm_start*, when only instructions for proper handling of flags and registers are executed.

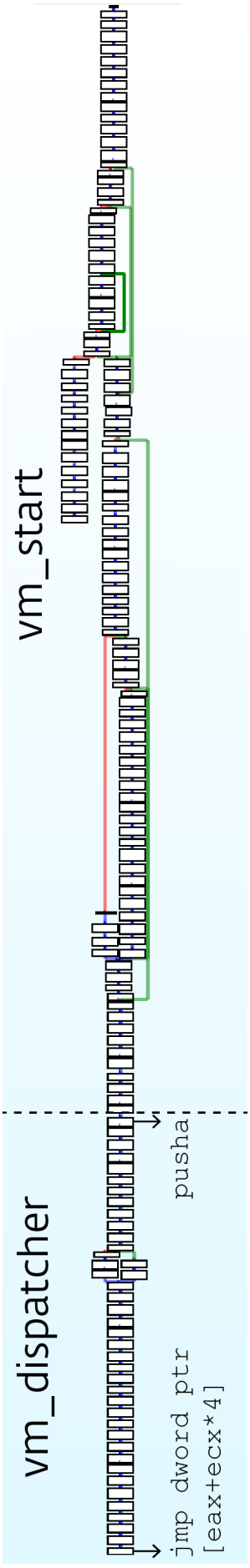


Figure 4 // Graph of the vm_start and vm_dispatcher

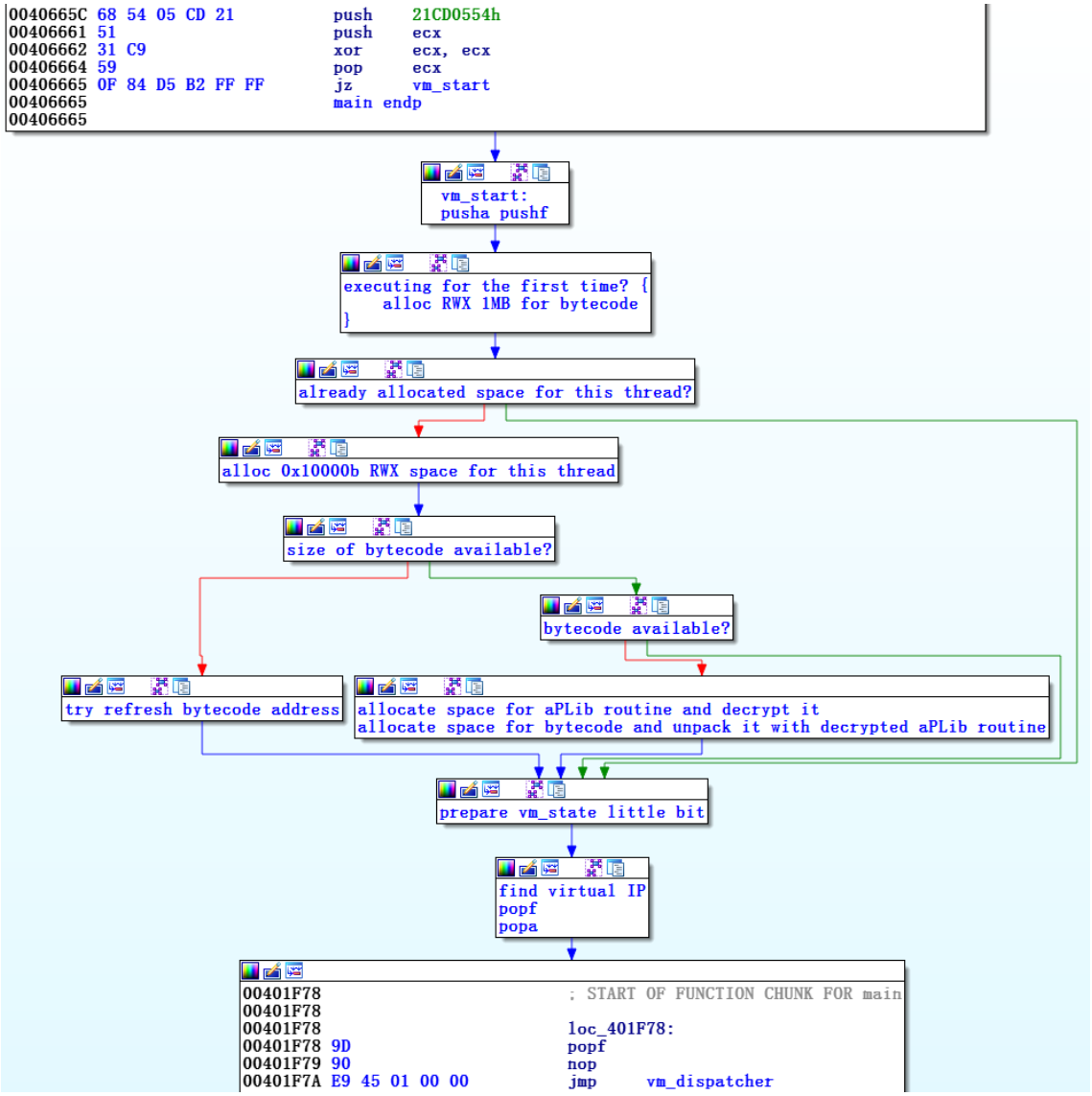


Figure 6 // All the code from the *vm_start* to the *vm_dispatcher* in grouped nodes named based on their purpose.

FINFISHER'S INTERPRETER

This part includes the *vm_dispatcher* with all the *vm_handlers* (34 in FinFisher samples) and is crucial for analyzing and/or devirtualizing the virtual machine. The interpreter executes the bytecode.

The instruction `jmp dword ptr [eax+ecx*4]` jumps to one of the 34 *vm_handlers*. Each *vm_handler* implements one virtual machine instruction. In order to know what every *vm_handler* does, we first need to understand the *vm_context* and *vm_dispatcher*.

1. Creating an IDA graph

Before diving into it, creating a well-structured graph can really help understanding the interpreter. We recommend splitting the graph into two parts – the *vm_start* and the *vm_dispatcher*, i.e. to define a beginning of a function at the vm_dispatcher's first instruction. What is still missing is the actual *vm_handlers* referenced by the *vm_dispatcher*. In order to connect these handlers with the graph of the

vm_dispatcher, the following functions can be used:

`AddCodeXref(addr_of_jmp_instr, vm_handler, XREF_USER|fl_JN)`
adding references from the last *vm_dispatcher* instruction to the beginnings of the *vm_handlers*

`AppendFchunk`
appending tails again

After appending every vm_handler to the dispatcher function, the resulting graph should look like (Figure 7)

2. Vm_dispatcher

This part is responsible for fetching and decoding the bytecode. It performs the following steps:

- Executes `pusha` and `pusf` instructions to prepare virtual registers and virtual flags for further execution of a virtual instruction.
- Retrieves the base address of the image and address of *vm_stack*
- Reads 24 bytes of bytecode specifying the next *vm_instruction* and its arguments

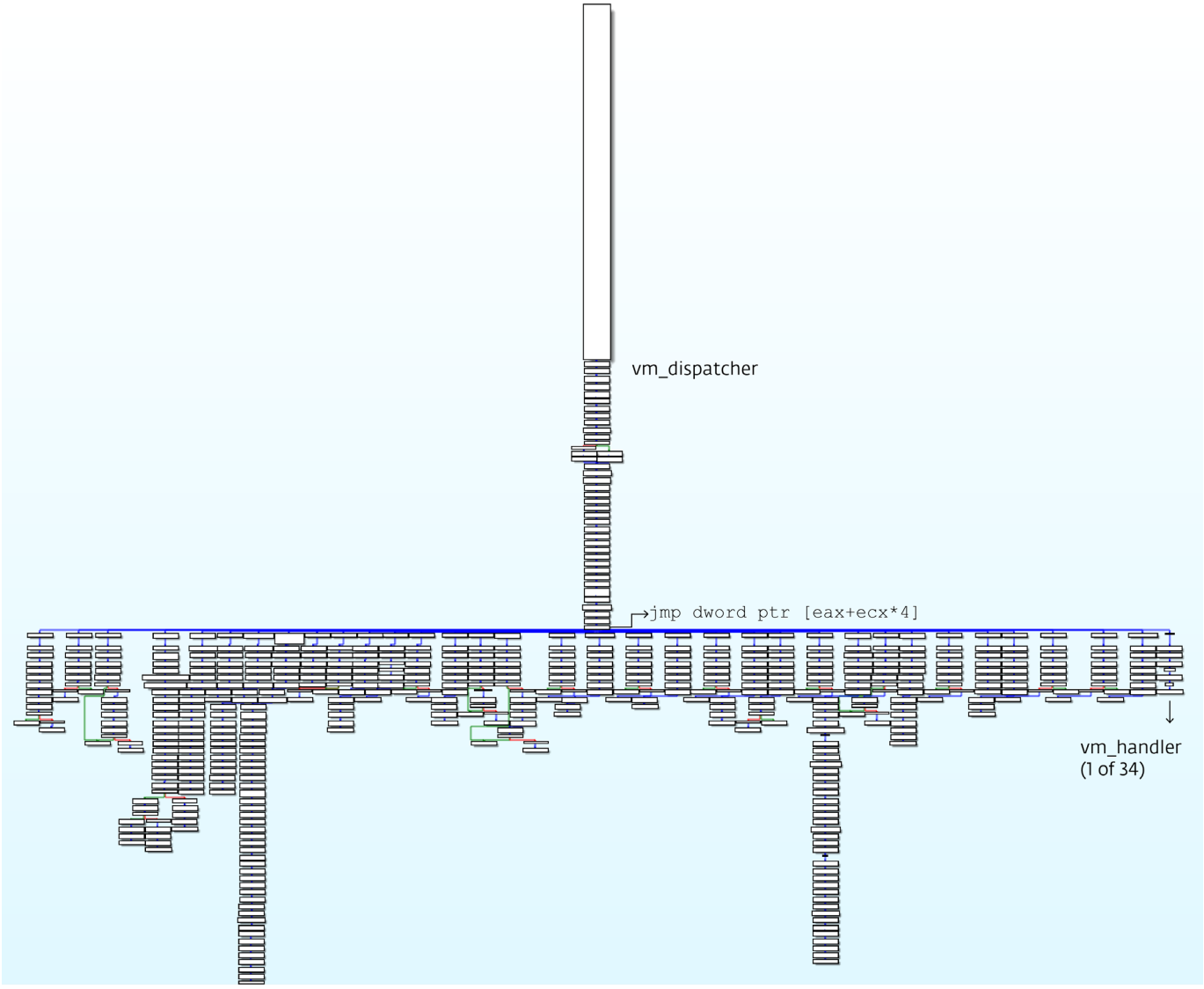


Figure 7 // Graph of the *vm_dispatcher* with all 34 *vm_handlers*.

- Decrypts the bytecode with the previously described XOR decryption routine
- Adds the image base to the bytecode argument in case the argument is a global variable
- Retrieves the virtual opcode (number 0-33) from the decrypted bytecode
- Jumps to the corresponding *vm_handler* which interprets the virtual opcode

After the *vm_handler* for an instruction has executed, the same sequence of steps is repeated for the next one, starting from the *vm_dispatcher's* first instruction.

In the case of the *vm_call* handler, the control is passed to the *vm_start* part instead (except for instances when a non-virtualized function follows).

3. Vm_context

In this part, we will describe the *vm_context* – a structure used by the virtual machine, containing all the information necessary for executing the vm_dispatcher and each *vm_handler*.

When looking at the code of both the *vm_dispatcher* and the *vm_handlers* in greater detail, we can notice there are a lot of data operation instructions, referring to `ebx+offset`, where offset is a number from 0x00 to 0x50. In Figure 8, we can see what the main part of *vm_handler* 0x05 in one FinFisher sample looks like. (Figure 8)

```
0040325D                ; START OF FUNCTION CHUNK FOR vm_dispatcher
0040325D
0040325D                loc_40325D:
0040325D 8D 4B 40       lea     ecx, [ebx+40h]
00403260 EB 18          jmp     short loc_40327A
00403260                ; END OF FUNCTION CHUNK FOR vm_dispatcher

0040327A                ; START OF FUNCTION CHUNK FOR vm_dispatcher
0040327A
0040327A                loc_40327A:
0040327A 8B 09          mov     ecx, [ecx]
0040327C EB E7          jmp     short loc_403265
0040327C                ; END OF FUNCTION CHUNK FOR vm_dispatcher

00403265                ; START OF FUNCTION CHUNK FOR vm_dispatcher
00403265
00403265                loc_403265:
00403265 8B 43 08       mov     eax, [ebx+8]
00403268 EB 27          jmp     short loc_403291
00403268                ; END OF FUNCTION CHUNK FOR vm_dispatcher

00403291                ; START OF FUNCTION CHUNK FOR vm_dispatcher
00403291
00403291                loc_403291:
00403291 89 08          mov     [eax], ecx
00403293 EB EE          jmp     short loc_403283
00403293                ; END OF FUNCTION CHUNK FOR vm_dispatcher
```

Figure 8 // Screenshot of one of the vm_handlers

The ebx register points to a structure we named *vm_context*. We must understand how this structure is used – what the members are, what they mean, and how they are used. When solving this puzzle for the first time, a bit of guessing is needed as to how the *vm_context* and its members are used.

For example, let's have a look at the sequence of instructions at the end of the vm_dispatcher:

```
movzx ecx, byte ptr [ebx+0x3C]
// opcode for vm_handler
jmp dword ptr [eax+ecx*4]
// jumping to one of the 34 vm_
handlers
```

Since we know that the last instruction is a jump to a *vm_handler*, we can conclude that ecx contains a virtual opcode and thus the 0x3C member of a *vm_struct* refers to a virtual opcode number.

Let's make one more educated guess. At the end of almost every vm_handler, there is the following instruction:

`add dword ptr [ebx], 0x18.`

This same member of the *vm_context* was also used earlier in the *vm_dispatcher's* code – just before jumping to a *vm_handler*. The *vm_dispatcher* copies 24 bytes from the structure member to a different location (`[ebx+38h]`) and decrypts it with the XOR decryption routine to obtain a part of the actual bytecode.

Hence, we can start thinking of the first member of the *vm_context* (`[ebx+0h]`) as a *vm_instruction_pointer*, and of the decrypted location (from `[ebx+38h]` to `[ebx+50h]`) as an ID of a virtual instruction, its virtual opcode and arguments. Together, we will call the structure *vi_params*.

Following the steps described above, and using a debugger to see what values are stored in the respective structure members, we can figure out all the members of the *vm_context*.

After the analysis, we can rebuild both FinFisher's *vm_context* and *vi_params* structure:

```
struct vm_context {

DWORD vm_instruct_ptr; // instruction pointer to the bytecode
DWORD vm_stack; // address of the vm_stack
DWORD tmp_REG; // used as a "register" in the virtual machine
DWORD vm_dispatcher_loop; // address of the vm_dispatcher
DWORD cleanAndVMDispatchFn; // address of the function which pops values and jumps
to the vm_dispatcher skipping the first few instructions from it
DWORD cleanUpDynamicCodeFn; // address of the function which cleans vm_instr_ptr and
calls cleanAndVMDispatchFn
DWORD jmpLoc1; // address of jump location
DWORD jmpLoc2; // address of next vm_opcode - just executing next vm_instruction
DWORD Bytecode_start; // address of the start of the bytecode in data section
DWORD DispatchEBP;
DWORD ImageBase; // Image base address
DWORD ESP0_flags; // top of the native stack (there are saved flags of the vm_code)
DWORD ESP1_flags; // same as previous
DWORD LoadVOpcodesSectionFn;
vi_params bytecode; // everything necessary for executing vm_handler, see below
DWORD limitForTopOfStack; // top limit for the stack
};
```

```
struct vi_params {
DWORD Virtual_instr_id;
DWORD OpCode; // values 0 - 33 -> tells which handler to execute
DWORD Arg0; // 4 dword arguments for vm_handler
DWORD Arg4; // sometimes unused
DWORD Arg8; // sometimes unused
DWORD ArgC; // sometimes unused
};
```

4. Virtual instruction implementations – vm_handlers

Each *vm_handler* handles one virtual opcode – since we have 34 *vm_handlers*, there are at most 34 virtual opcodes. Executing one *vm_handler* means executing one *vm_instruction*, so in order to reveal what a *vm_instruction* does, we need to analyze the corresponding vm_handler.

After reconstructing the *vm_context* and naming all the offsets from ebx, the previously shown *vm_handler* changes to a much more readable form, as seen in Figure 9.

```
0040325D                ; START OF FUNCTION CHUNK FOR vm_dispatcher
0040325D
0040325D                loc_40325D:
0040325D 8D 4B 40       lea     ecx, [ebx+vm_state.opcodes.Arg0]
00403260 EB 18          jmp     short loc_40327A
00403260                ; END OF FUNCTION CHUNK FOR vm_dispatcher

0040327A                ; START OF FUNCTION CHUNK FOR vm_dispatcher
0040327A
0040327A                loc_40327A:
0040327A 8B 09          mov     ecx, [ecx]
0040327C EB E7          jmp     short loc_403265
0040327C                ; END OF FUNCTION CHUNK FOR vm_dispatcher

00403265                ; START OF FUNCTION CHUNK FOR vm_dispatcher
00403265
00403265                loc_403265:
00403265 8B 43 08       mov     eax, [ebx+vm_state.tmp_REG]
00403268 EB 27          jmp     short loc_403291
00403268                ; END OF FUNCTION CHUNK FOR vm_dispatcher

00403291                ; START OF FUNCTION CHUNK FOR vm_dispatcher
00403291
00403291                loc_403291:
00403291 89 08          mov     [eax], ecx
00403293 EB EE          jmp     short loc_403283
00403293                ; END OF FUNCTION CHUNK FOR vm_dispatcher
```

Figure 9 // The previous *vm_handler* after inserting the *vm_context* structure

At the end of this function, we notice a sequence of instructions, starting with the vm_instruction_pointer, being incremented by 24 – the size of each vm_instruction's vi_params structure. Since this sequence is repeated at the end of almost every vm_handler, we conclude it is a standard function epilogue and the actual body of the vm_handler can be written as simply as:

```
mov    [tmp_REG], Arg0
```

So, there we go – we have just analyzed the first instruction of the virtual machine. :-)

To illustrate how the analyzed instruction works when executed, let's consider the vi_params structure filled as follows:

```
struct vi_params {
DWORD ID_of_virt_instr = doesn't
matter;
DWORD OpCode = 0x0C;
DWORD Arg0 = 0x42;
DWORD Arg4 = 0;
DWORD Arg8 = 0;
DWORD ArgC = 0;
};
```

From what was stated above, we can see that the following instruction will be executed:

```
mov    [tmp_REG], 0x42
```

At this point, we should understand what one of the *vm_instructions* does. The steps we followed should serve as a decent demonstration of how the entire interpreter works.

However, there are some *vm_handlers* that are harder to analyze. This vm's conditional jumps are tricky to understand because of the way they translate flags.

As mentioned before, the vm_dispatcher starts with pushing native EFLAGS (of *vm_code*) to the top of the native stack. Therefore, when the handler for a respective jump is deciding whether to jump or not, it looks at EFLAGS at the native stack and implements its own jump method. Figure 10 illustrates how the virtual JNP handler is implemented by checking the parity flag. (Figure 10)

Figure 10 // Screenshot of a *JNP_handler*

For other virtual conditional jumps, it may be necessary to check several flags – for example, the jump result of the virtualized JBE depends on the values of both CF and ZF – but the principle stays the same.

After analyzing all 34 *vm_handlers* in FinFisher's virtual machine, we can describe its virtual instructions as follows:

```
.text:00402ABA VM_table dd offset case_0_JL_loc1
.text:00402ABE dd offset case_1_JNP_loc1
.text:00402AC2 dd offset case_2_JLE_loc1
.text:00402AC6 dd offset case_3_vm_jcc
.text:00402ACA dd offset case_4_exec_native_code; same as case 6
.text:00402ACE dd offset case_5_mov_tmpREGref_Arg0; mov [tmpREG], Arg0
.text:00402AD2 dd offset case_6_exec_native_code
.text:00402AD6 dd offset case_7_JZ_loc1
.text:00402ADA dd offset case_8_JG_loc1
.text:00402ADE dd offset case_9_mov_tmpREG_Arg0; mov tmpREG, Arg0
.text:00402AE2 dd offset case_A_zero_tmpREG; mov tmpREG, 0
.text:00402AE6 dd offset case_B_JS_loc1
.text:00402AEA dd offset case_C_mov_tmpREGDeref_reg; mov [tmpREG], reg
.text:00402AEE dd offset case_D_mov_tmpREG_reg
.text:00402AF2 dd offset case_E_JB_loc1
.text:00402AF6 dd offset case_F_JBE_loc1
.text:00402AFA dd offset case_10_JNZ_loc1
.text:00402AFE dd offset case_11_JNO_loc1
.text:00402B02 dd offset case_12_vm_call
.text:00402B06 dd offset case_13_mov_tmpREG_reg; mov tmpREG, reg
.text:00402B0A dd offset case_14_JP_loc1
.text:00402B0E dd offset case_15_mov_reg_tmpREG; mov reg, tmpREG
.text:00402B12 dd offset case_16_JO_loc1
.text:00402B16 dd offset case_17_JGE_loc1
.text:00402B1A dd offset case_18_deref_tmpREG; mov tmpREG, [tmpREG]
.text:00402B1E dd offset case_19_shl_tmpREG_Arg0; shl tmpREG, (byte)Arg0
.text:00402B22 dd offset case_1A_JNS_loc1
.text:00402B26 dd offset case_1B_JNB_loc1
.text:00402B2A dd offset case_1C_push_tmpREG; push tmpREG
.text:00402B2E dd offset case_1D_JA_loc1
.text:00402B32 dd offset case_1E_add_tmpREG_reg; add tmpREG, reg
.text:00402B36 dd offset case_1F_vm_jmp
.text:00402B3A dd offset case_20_add_tmpREG_arg0
.text:00402B3E dd offset case_21_mov_tmpREG_to_Arg0Deref; mov [Arg0], REG
```

Figure 11 // *vm_table* with all 34 *vm_handlers* accessed

Please note that the keyword "*tmp_REG*" refers to a virtual register used by the virtual machine –temporary register in the *vm_context* structure, while "*reg*" refers to a native register, e.g. eax.

Let's have a look at the analyzed instructions of the virtual machine. For example, *case_3_vm_jcc* is a general jump handler that can execute any native jump, either conditional or unconditional.

Apparently, this virtual machine does not virtualize every native instruction – that's where instructions in cases 4 and 6 come in handy.

These two vm_handlers are implemented to execute native code directly – all they do is to read the opcode of a native instruction given as an argument and execute the instruction.

One more thing to note is that the vm_registers are always at the top of the native stack, while the identifier of the register to be used is stored in the last byte of arg0 of the virtual instruction. The following code can be used to access the respective virtual register:

```
def resolve_reg(reg_pos):
    stack_regs = ['eax', 'ecx', 'edx', 'ebx', 'esp', 'ebp', 'esi', 'edi']
    stack_regs.reverse()
    return stack_regs[reg_pos]
reg_pos = 7 - (state[arg0] & 0x000000FF)
reg = resolve_reg(reg_pos)
```

5. Writing your own disassembler

After we have correctly analyzed all the *vm_instructions*, there is still one step to be done before we can start the analysis of the sample – we need to write our own disassembler for the bytecode (parsing it manually would be problematic due to its size).

By putting in the effort and writing a more robust disassembler we can save ourselves some trouble when FinFisher's virtual machine is changed and updated.

Let's start with the *vm_handler* 0x0C, which executes the following instruction:

```
mov [tmp_REG], reg
```

This instruction takes exactly one argument – the identifier of a native register to be used as `reg`. This identifier must be mapped into a native register name, for instance using a `resolve_reg` function as described above.

The following code can be used to dissasemble this *vm_handler*:

```
def vm_0C(state, vi_params):
    global instr
    reg_pos = 7 - (vi_arams[arg0]
    & 0x000000FF)
    tmpinstr = "mov [tmp_REG],
    %s" % resolve_reg(reg_pos)
    instr.append(tmpinstr)
    return
```

Again, *vm_handlers* for jumps are harder to understand. In case of jumps, members *vm_context.vi_params.Arg0* and *vm_context.vi_params.Arg1* store the offset by which to jump. This "jump offset" is actually an offset in the bytecode. When parsing jumps, we need to put a marker to the location to which it jumps. For example, this code can be used:

```
def computeLoc1(pos, vi_params):
    global instr

    jmp_offset = (vi_params[arg0]
& 0x00FFFFFF) + (vi_params[arg1]
& 0xFF000000)

    if jmp_offset < 0x7FFFFFFF:
        jmp_offset /= 0x18 # their
        increment by 0x18 is my
        increment by 1
    else:
        jmp_offset = int((-
        0x100000000 + jmp_offset)
        / 0x18)

    return pos+jmp_offset
```

Finally, there is a vm_handler responsible for executing native instructions from arguments, which needs special treatment. For this, we have to use a disassembler for native x86 instructions – for example, the open source tool Distorm.

The length of an instruction is stored in vm_context.vi_params.OpCode & 0x0000FF00. The opcode of the native instruction that will be executed is stored in the arguments. The following code can be used to parse the vm_handler that executes native code:

```
def vm_04(vi_params, pos):
    global instr

    nBytes = vi_params[opCode] & 0x0000FF00
    dyn_instr = pack("<LLLL", vi_params[arg0], vi_params[arg4],
vi_params[arg8], vi_params[argC])[0:nBytes]
    dec_instr = distorm3.Decode(0x0, dyn_instr, distorm3.Decode32Bits)

    tmpinstr = "%s" % (dec_instr[0][2])
    instr.append(tmpinstr)
    return
```

Up to this point, we have created Python functions to disassemble each vm_handler. All of these, combined with the code responsible for marking jump locations, finding the ID of a virtual instruction after the call and a few others, are necessary for writing your own disassembler.

Afterwards, we can run the finished disassembler on the bytecode.

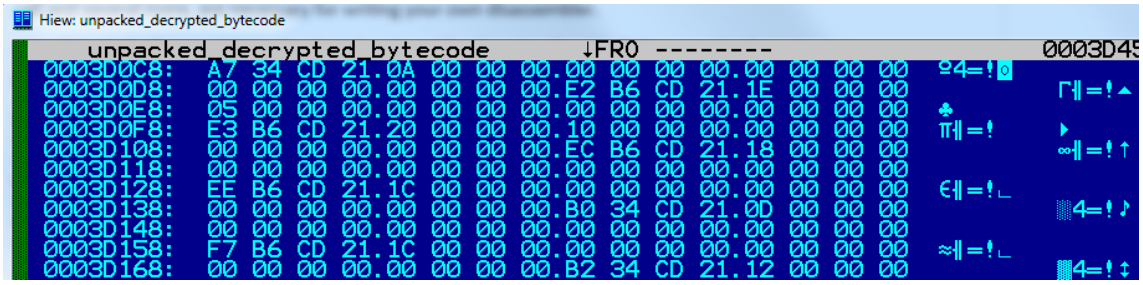


Figure 12 // Part of the unpacked and decrypted FinFisher bytecode

For example, from the part of the bytecode shown in Figure 12, we may get the following output:

```
mov tmp_REG, 0
add tmp_REG, EBP
add tmp_REG, 0x10
mov tmp_REG, [tmp_REG]
push tmp_REG
mov tmp_REG, EAX
push tmp_REG
```

6. Understanding the implementation of this virtual machine

After we have analyzed all the virtual handlers and constructed a custom disassembler, we can have one more look at the virtual instructions to get an overall idea of how they were created.

First, we must understand that the virtualization protection was implemented at the assembly level. The authors translated native instructions into their own, somewhat complicated instructions, which are to be executed by a custom virtual CPU. To achieve this, a temporary "register" (*tmp_REG*) is used.

We can look at some examples to see how this translation works. For example, the virtual instruction from the previous example –

```
mov tmp_REG, EAX
push tmp_REG
```

– was translated from the original native instruction `push eax`. When virtualized, a temporary register was used in an intermediate step to change the instruction into something more complicated.

Let's consider another example:

```
mov tmp_REG, 0
add tmp_REG, EBP
add tmp_REG, 0x10
mov tmp_REG, [tmp_REG]
push tmp_REG
```

The native instructions that were translated into these virtualized instructions were the following (with reg being one of the native registers):

```
mov reg, [ebp+0x10]
push reg
```

This is, however, not the only way to virtualize a set of instructions. There are other virtual machine protectors with other approaches. For instance, one of the commercial vm protectors translates each math operation instruction into NOR logic, with a number of temporary registers being used instead of one.

Conversely, FinFisher's virtual machine did not go as far as to cover all the native instructions. While many of them can be virtualized, some can't – math instructions, such as `add`, `imul` and `div`, being some examples. If these instructions appear in the original binary, the *vm_handler* responsible for executing native instructions is called to handle them in the protected binary. The only change is that EFLAGS and native registers are popped from the native stack just before the native instruction is executed, and pushed back after it is executed. This is how the virtualization of every native instruction was avoided.

A significant drawback of protecting binaries with a virtual machine is the performance impact. In the caseof FinFisher's virtual machine, we estimate it to be more than one hundred times slower than native code, based on the number of instructions that have to be executed to handle every single *vm_instruction* (*vm_dispatcher* + *vm_handler*).

Therefore, it makes sense to protect only selected parts of the binary– and this is also the case in the FinFisher samples we analyzed.

Moreover, as mentioned before, some of the virtual machine handlers can call native functions directly. As a result, the users of the virtual machine protection (i.e. the authors of FinFisher) can look at the functions at the assembly level and mark which of them are to be protected by the virtual machine. For those that are marked, their instructions will be virtualized, for those that are not, the original functions will be called by the respective virtual handler. Thus, the execution might be less time-consuming while the most interesting parts of the binary stay protected. (Figure 13)

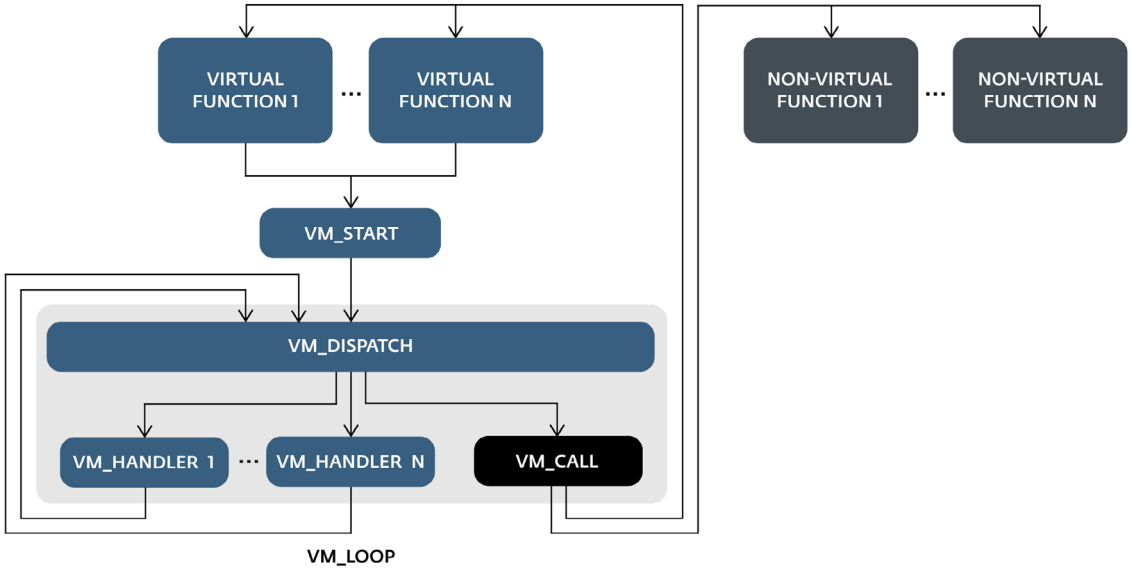


Figure 13 // Scheme representing FinFisher's entire vm protection and how the execution can jump out of the vm

7. Automating the disassembly process for more FinFisher samples

In addition to the length of the bytecode our parser has to process, we have to keep in mind that there is some randomization across various FinFisher samples. Although the same virtual machine has been used for the protection, the mapping between the virtual opcodes and the *vm_handlers* is not always constant. They can be (and are) paired randomly and differently for each of the FinFisher samples we analyzed. It means that if the *vm_handler* for the 0x5 virtual opcode in this sample handles the `mov [tmp_REG], arg0` instruction, it may be assigned a different virtual opcode in another protected sample.

To address this issue, we can use a signature for each of the analyzed *vm_handlers*. The IDA Python script in Appendix A can be applied after we have generated a graph as shown in Figure 7 (it is particularly important to have the jz/jnz jump obfuscation eliminated – as described in the first section of this guide) to name the handlers based on their signatures. (With a small modification, the script can also be used to recreate the signatures in case the *vm_handlers* are changed in a future FinFisher update.)

As mentioned above, the first *vm_handler* in the FinFisher sample you analyze may be different than JL, as in the example FinFisher sample, but the script will identify all of the *vm_handlers* correctly.

8. Compiling disassembled code without the VM

After disassembly and after a few modifications, it is possible to compile the code. We will treat virtual instructions as native instructions. As a result, we will get a pure binary without the protection.

Most of the vm_instructions can be compiled immediately using copy-paste, since the output of our disassembler mostly consists of native-looking instructions. But some cases need special treatment:

- tmp_REG – since we defined tmp_REG as a global variable, we need to make code adjustments for cases when an address stored in it is being dereferenced. (Since

dereferencing an address which is in a global variable is not possible in the x86 instruction set.) For example, the vm contains the virtual instruction `mov tmp_REG, [tmp_REG]` which needs to be rewritten as follows:

```
push eax
mov eax, tmp_REG
mov eax, [eax]
mov tmp_REG, eax
pop eax
```

- Flags – Virtual instructions do not change the flags, but native math instructions do. Therefore, we need to make sure that virtual math instruction won't change flags in the devirtualized binary either, which means we have to save flags before executing this instruction and restore them after the execution.
- Jumps and calls – we have to put a marker to the destination virtual instruction (jumps) or function (calls).
- API function calls – in most cases, API functions are loaded dynamically, whereas in others they are referenced from the IAT of the binary, so these cases need to be handled accordingly.
- Global variables, native code – Some global variables need to be kept in the devirtualized binary. Also in the FinFisher dropper, there is a function for switching to x64 from x86 that is executed natively (actually it is done only with the `retf` instruction). All these must be kept in the code when compiling.

Depending on the output of your disassembler, you may still need to do a few more modifications to get pure native instructions that can be compiled. Then, you can compile the code with your favorite assembly-compiler into a binary without the VM.

CONCLUSION

In this guide, we have described how FinFisher uses two elaborate techniques to protect its main payload. The primary intention of this protection is not to avoid AV detection, but to cover the configuration files and new techniques implemented in the spyware by hindering analysis by reverse engineers. As no other detailed analysis of the obfuscated FinFisher spyware has been published to date, it seems the developers of these protection mechanisms have been successful.

We have shown how we can overcome the anti-disassembly layer automatically, and how the virtual machine can be efficiently analyzed.

We hope this guide can help reverse engineers analyze vm-protected FinFisher samples, as well to better understand other virtual machine protectors in general.

Appendix A

IDA Python script for naming FinFisher vm_handlers

The script is also available on ESET's GitHub repository:
https://github.com/eset/malware-research/blob/master/finfisher/ida_finfisher_vm.py

```
import sys

SIGS = { '8d4b408b432c8b0a90800f95c2a980000f95c03ac275ff631c' : 'case_0_JL
_loc1', '8d4b408b432c8b0a9400074ff631c' : 'case_1_JNP_loc1', '8d4b408b432c
8b0a94000075a90800f95c2a980000f95c03ac275ff631c' : 'case_2_JLE_loc1', '8d4
b408b7b508b432c83e02f8dbc38311812b5c787cfe7ed4ae92f8b066c787d3e7e
4af9b8e80000588d80' : 'case_3_vm_jcc', '8b7b508b432c83e02f3f85766c77ac6668
137316783c728d7340fb64b3df3a4c67e98037818b43c89471c64756c80775af83318588b6
32c' : 'case_4_exec_native_code', '8d4b408b98b438898833188b43c8b632c' : 'c
ase_5_mov_tmp_REGref_arg0', '8b7b508b432c83e02f3f85766c77ac6668137316783c7
28d7340fb64b3df3a4c67e98037818b43c89471c64756c80775af83318588b632c' : 'cas
e_6_exec_native_code', '8d4b408b432c8b0a94000075ff631c' : 'case_7_JZ_loc1'
, '8d4b408b432c8b0a94000075a90800f95c2a980000f95c03ac275ff6318' : 'case_8_
JG_loc1', '8d43408b089438833188b43c8b632c' : 'case_9_mov_tmp_REG_arg0', '3
3c9894b8833188b632c8b43c' : 'case_A_zero_tmp_REG', '8d4b408b432c8b0a980000
75ff631c' : 'case_B_JS_loc1', '8d4b40fb69b870002bc18b4b2c8b548148b4b889118
33188b43c8b632c' : 'case_C_mov_tmp_REGDeref_tmp_REG', '8d4b40fb69b870002bc
18b4b2c8b4481489438833188b43c8b632c' : 'case_D_mov_tmp_REG_tmp_REG', '8d4b
408b432c8b0a9100075ff631c' : 'case_E_JB_loc1', '8d4b408b432c8b0a9100075a94
000075ff631c' : 'case_F_JBE_loc1', '8d4b408b432c8b0a94000074ff631c' : 'cas
e_10_JNZ_loc1', '8d4b408b432c8b0a9080074ff631c' : 'case_11_JNO_loc1', '8b7
b50834350308d4b408b414343285766c773f50668137a231c6472c280772aa8d57d83c7389
1783ef3c7477a300080777cb83c7889783ef8c647cf28077c3183c7dc67688b383c0188947
183c7566c7777fe668137176283c72c672d803745895f183c75c67848037df478b4314c674
08037288947183c75c67928037515f8b632c' : 'case_12_vm_call', '8d4b40b870002b
18b532c8b4482489438833188b43c8b632c' : 'case_13_mov_tmp_REG_tmp_REG_notRly
', '8d4b408b432c8b0a9400075ff631c' : 'case_14_JP_loc1', '8d4b40fb69b870002
bc18b4b2c8b5388954814833188b43c8b632c' : 'case_15_mov_tmp_REG_tmp_REG', '8
d4b408b432c8b0a9080075ff631c' : 'case_16_JO_loc1', '8d4b408b432c8b0a90800f
95c2a980000f95c03ac274ff631c' : 'case_17_JGE_loc1', '8b4388b089438833188b4
3c8b632c' : 'case_18_deref_tmp_REG', '8d4b408b4388b9d3e089438833188b43c8b6
32c' : 'case_19_shl_tmp_REG_arg0l', '8d4b408b432c8b0a98000074ff631c' : 'ca
se_1A_JNS_loc1', '8d4b408b432c8b0a9100074ff631c' : 'case_1B_JNB_loc1', '8b
7b2c8b732c83ef4b924000fcf3a4836b2c48b4b2c8b438894124833188b43c8b632c' : 'c
ase_1C_push_tmp_REG', '8d4b408b432c8b0a94000075a9100075ff6318' : 'case_1D_
JA_loc1', '8d4b40b870002b18b532c8b448241438833188b43c8b632c' : 'case_1E_ad
d_stack_val_to_tmp_REG', '8b7b508343503066c77ac3766813731565783c728d4b40c6
72e803746fb6433d3c783c058947183c758d714fb64b3df3a45ac671280377a8b383c01889
47183c7566c777f306681371fac83c72c671f803777895f183c75c677080372b47c6798037
618b4b14894f183c75c67778037b48b632c8d12' : 'case_1F_vm_jmp', '8d4b408b914b
8833188b43c8b632c' : 'case_20_add_arg0_to_tmp_REG', '8d4b408b98b4388918331
88b632c8b43c' : 'case_21_mov_tmp_REG_to_arg0Dereferenced' }

SWITCH = 0 # addr of jmp     dword ptr [eax+ecx*4] (jump to vm_handlers)
SWITCH_SIZE = 34

sig = []

def append_bytes(instr, addr):
    for j in range(instr.size):
        sig.append(Byte(addr))
```

```
        addr += 1
    return addr

def makeSigName(sig_name, vm_handler):
    print "naming %x as %s" % (vm_handler, sig_name)
    MakeName(vm_handler, sig_name)
    return

if SWITCH == 0:
    print "First specify address of switch jump - jump to vm_handlers!"
    sys.exit(1)

for i in range(SWITCH_SIZE):
    addr = Dword(SWITCH+i*4)
    faddr = addr

    sig = []

    while 1:

        instr = DecodeInstruction(addr)
        if instr.get_canon_mnem() == "jmp" and (Byte(addr) == 0xeb or Byte
(addr) == 0xe9):
            addr = instr.Op1.addr
            continue
        if instr.get_canon_mnem() == "jmp" and Byte(addr) == 0xff and Byte
(addr+1) == 0x63 and (Byte(addr+2) == 0x18 or Byte(addr+2) == 0x1C):
            addr = append_bytes(instr, addr)
            break
        if instr.get_canon_mnem() == "jmp" and Byte(addr) == 0xff:
            break
        if instr.get_canon_mnem() == "jz":
            sig.append(Byte(addr))
            addr += instr.size
            continue
        if instr.get_canon_mnem() == "jnz":
            sig.append(Byte(addr))
            addr += instr.size
            continue
        if instr.get_canon_mnem() == "nop":
            addr += 1
            continue
        addr = append_bytes(instr, addr)

    sig_str = "".join([hex(l)[2:] for l in sig])
    hsig = ''.join(map(chr, sig)).encode("hex")

    for key, value in SIGS.iteritems():

        if len(key) > len(sig_str):
            if key.find(sig_str) >= 0:
                makeSigName(value, faddr)
        else:
            if sig_str.find(key) >= 0:
                makeSigName(value, faddr)
```