Whitepaper de Investigación de ESET // Enero 2018 // Autor: Flip Kafka

GUÍA DE ESET PARA DESOFUSCAR Y DESVIRTUALIZAR FINFISHER

ENJOY SAFER TECHNOLOGY™

CONTENIDO

INTRODUCCIÓN

El spyware FinFisher ha tenido un largo tiempo sin ser explorado, debido a sus fuertes medidas anti-análisis. A pesar de ser una herramienta de vigilancia prominente, solo se han publicado análisis parciales de sus muestras más recientes.

A mediados de 2017, ESET analizó campañas de FinFisher en varios países. En el transcurso de nuestra investigación, identificamos *campañas en las que los proveedores de servicios de Internet* probablemente hayan jugado un rol central para comprometer a las víctimas con FinFisher.

Cuando comenzamos a analizar en profundidad este malware, nuestro esfuerzo se concentró en superar las medidas anti-análisis de FinFisher en sus versiones para Windows. La combinación de técnicas de ofuscación avanzada y virtualización propietaria, hicieron de FinFisher un código malicioso difícil de desmantelar.

Con el fin de compartir lo que aprendimos al desenmascarar este malware, hemos creado la presente guía para ayudar a otros a conocer y analizar FinFisher. Además de ofrecer información práctica sobre el análisis de la máquina virtual de FinFisher, la guía también puede ayudar a los lectores a entender la protección de máquinas virtuales en general – es decir, máquinas virtuales patentadas que se encuentran dentro de un binario y se utilizan para la protección del software. No hablaremos sobre máquinas virtuales utilizadas en lenguajes de programación interpretados, para proporcionar compatibilidad con varias plataformas, como la máquina virtual de Java.

También hemos analizado versiones de FinFisher para Android, cuyo mecanismo de protección está basado en un ofuscador LLVM (antes conocido como Low Level Virtual Machine), de código abierto. No es tan sofisticado o interesante como el mecanismo de protección utilizado en las versiones para Windows, por lo cual no lo abarcaremos en esta guía.

Esperamos que los expertos, desde investigadores de seguridad hasta analistas de malware, hagan uso de esta guía para comprender en mayor medida las herramientas y tácticas de FinFisher, así como para proteger a sus clientes de esta amenaza omnipotente para la seguridad y privacidad.

ANTI – ENSAMBLADO

Cuando abrimos una muestra de FinFisher en IDA Pro, la primera línea de protección que notamos en la función central, es un sencillo pero efectivo, truco anti-desensamblador.

FinFisher utiliza una técnica común anti-desensamblado – ocultando el flujo de ejecución al reemplazar un salto no convencional con dos saltos condicionales complementarios. Estos saltos condicionales apuntan a una misma ubicación, por lo que, independientemente de qué salto se realice, resulta es el mismo efectivo flujo de ejecución de código. Los saltos condicionales son seguidos de bytes de basura. Su función es dirigir el desensamblador en otra dirección, que normalmente no notaría que se trata de código muerto, y trabajará sobre este, desensamblando código basura.

Lo que vuelve a este malware especial es la forma en que utiliza esta técnica, poco utilizada en la gran mayoría de los códigos maliciosos que hemos analizado. Sin embargo, FinFisher utiliza este truco luego de cada instrucción sencilla.

Esta protección es muy efectiva a la hora de engañar al desensamblador, ya que muchas partes del código no se desensamblan correctamente. Y por supuesto, es imposible usar el modo gráfico en IDA Pro. Nuestra primera tarea consiste en deshacernos de esta protección anti-desensamblado.

El código no está ofuscado de forma manual sino mediante una herramienta automatizada, y podemos observar un patrón en todos los pares de saltos.

Existen dos tipos distintos de pares de saltos – near jump con un offset de 32 bits y short jump con un offset de 8 bits.

El código de operación de ambos near jump condicionales (con dword como offset de salto) comienza con un byte 0x0F; mientras que los bytes que le siguen igualan a 0x8?, donde ? en ambas instrucciones de saltos difiere únicamente por un bit. Esto se debe a que los códigos de operación x86 para saltos complementarios son numéricamente consecutivos. Por ejemplo, este esquema de ofuscación siempre agrupa JE con JNE (códigos de operación 0x0F 0x84 vs 0x0F 0x85) JP con JNP (códigos de operación 0x0F 0x8A vs 0x0F 0x8B), y así sucesivamente.

Estos códigos de operación son seguidos por un argumento de 32 bits que especifica el offset al destino del salto. Dado que el tamaño de ambas instrucciones es de 6 bytes, los offset en dos saltos consecutivos difieren exactamente por 6. (Figura 1)

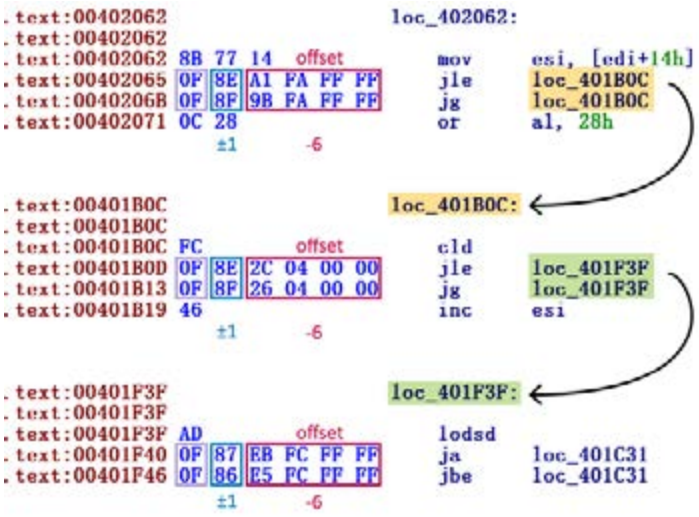

Figura 1 // Captura de pantalla que muestra instrucciones seguidas por dos *near jump* condicionales cada vez.

Por ejemplo, el código aquí debajo puede ser usado para detectar dos de estos saltos condicionales consecutivos:

```
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
```

La desofuscación de *short jump* está basada en la misma idea, solo difieren las constantes.

El código de operación de *short jump* condicional iguala a 0x7?, y es seguido por un byte – el offset del salto. Entonces, nuevamente, cuando queremos detectar dos near jump condicionales consecutivos, debemos buscar los códigos de operación: 0x7?; offset; 0x7? ± 1; offset -2. El primer código de operación es seguido por un byte, que difiere por 2 en dos saltos consecuentes (que es, nuevamente, el tamaño de ambas instrucciones). (Figura 2)

Por ejemplo, este código puede ser utilizado para detector dos *short jump* condicionales:

```
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
```

Tras detectar uno de estos pares de saltos condicionales, desofuscamos este código parchando el primer salto condicional a incondicional (usando el código de operación 0xE9 para los pares de *near jump* y 0xEB para los pares de *short jump*) y el resto de los bytes con instrucciones NOP (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)
```

Aparte de estos dos casos, puede haber algunos sitios en los que un par de saltos consista de un short jump y un near jump, en lugar de dos saltos de la misma categoría. Sin embargo, esto solo ocurre en pocos casos en las muestras de FinFisher, y puede arreglarse manualmente.

Con estos parches implementados, IDA Pro comienza a "entender" el nuevo código y está listo (o al menos más preparado) para crear un gráfico. Puede suceder que todavía necesitemos realizar una mejora más: append tails, es decir,

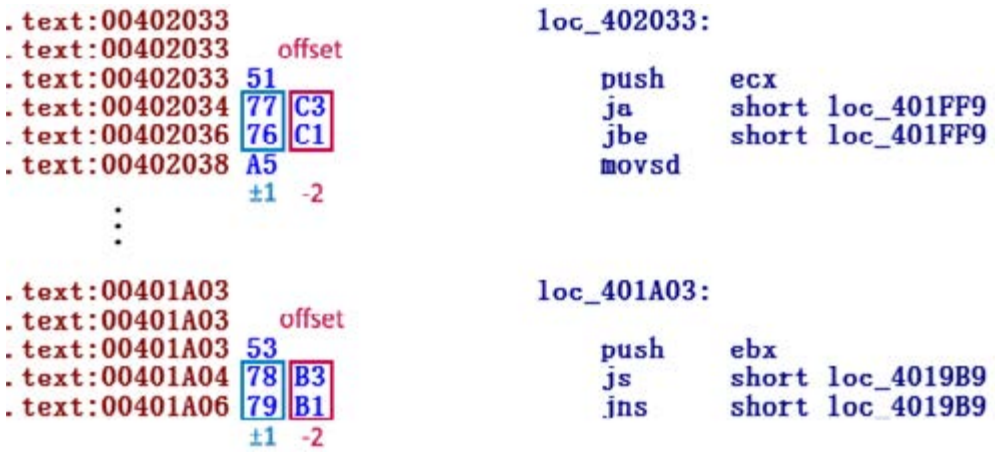

Figura 2 // Ejemplos de instrucciones seguidas por dos *short jump* condicionales cada vez

asignar el nodo con el destino del salto al mismo gráfico en que está localizada la instrucción del salto. Para esto, podemos usar la función `append_func_tail`. de IDA Python.

El último paso para superar los trucos anti-desensambladores consiste en solucionar definiciones de funciones. Puede todavía ocurrir que la instrucción que sigue a los saltos sea `push ebp`, , en cuyo caso IDA Pro trata esto como el comienzo de una función (incorrectamente) y crea una nueva definición de función. En ese caso, debemos remover la definición de función, crear la correcta y volver a incluir la función *append tails*.

Es así como podemos deshacernos de la primera capa de protección de FinFisher – Anti - desensamblaje.

MÁQUINA VIRTUAL DE FINFISHER

Tras una exitosa desofuscación de la primera capa, podemos ver una función principal más clara cuyo único propósito es lanzar una máquina virtual personalizada y dejarla que interprete el bytecode con la carga útil (payload) actual.

A diferencia de un ejecutable regular, uno con una máquina virtual dentro, utiliza un conjunto de instrucciones virtualizadas en lugar de usar directamente las instrucciones del procesador. Las instrucciones virtualizadas son ejecutadas por un procesador virtual, que tiene su propia estructura y no traduce el bytecode en un código nativo de una máquina. El procesador virtual, así como el bytecode (y las instrucciones virtuales) son definidas por el programador de la máquina virtual. (Figura 3)

AComo se mencionó en la introducción, un ejemplo conocido de una máquina virtual es Java Virtual Machine (JVM). Pero en este caso, la máquina virtual se ubica dentro del binario, por lo cual estamos lidiando con una máquina virtual utilizada para la protección contra ingeniería inversa. Existen algunos protectores de máquinas virtuales comerciales conocidos, como VMProtect o Code Virtualizer.

El spyware FinFisher fue compilado desde código fuente, y el binario compilado luego fue protegido con una máquina virtual al momento de ensamblarse. El proceso de protección incluye la traducción de instrucciones del binario original a instrucciones virtuales, creando luego un nuevo binario que contiene el bytecode y el CPU virtual. Las instrucciones nativas del binario original se pierden. La muestra virtualizada protegida debe tener el mismo comportamiento que una muestra sin proteger.

Para analizar un binario protegido por una máquina virtual, se debe:

1. Analizar el CPU virtual.
2. Escribir un desensamblador propio para este CPU virtual personalizado y analizar el bytecode.
3. Paso opcional: compilar el código desensamblador en un archivo binario para deshacerse de la máquina virtual.

Las primeras dos tareas llevan tiempo, y la primera puede también resultar algo difícil. Incluye analizar cada *vm_handler* y comprender cómo se traducen los registros, accesos de memoria, solicitudes, etc.

Figura 3 // Bytecode interpretado por el CPU virtual

Términos y definiciones

No hay un estándar para nombrar partes particulares de una máquina virtual. Por ello, definiremos algunos términos a los que se hará referencia a lo largo de todo el documento.

- Máquina Virtual (MV) – CPU virtual personalizado; contiene partes como *vm_dispatcher*, *vm_start*, *vm_handlers*
- *vm_start* – la parte de inicialización; aquí se ejecutan las distribuciones de memoria y rutinas de descifrado
- (también conocido como pcode) – códigos de operación virtuales de *vm_instructions* donde se almacenan los argumentos.
- *vm_dispatcher* – recupera y decodifica códigos de operación virtuales; en pocas palabras, es una preparación para la ejecución de uno de los vm_handlers.
- *vm_handler* – implementación de una vm_instruction; ejecutar un *vm_handler* significa ejecutar una *vm_instruction*.
- Intérprete/Interpretador (también llamado vm_loop) – *vm_dispatcher* + *vm_handlers* – el CPU virtual.
- Código de operación virtual – un análogo del código de operación nativo.
- *vm_context (vm_structure)* – una estructura interna usada por el intérprete/interpretador. *vi_params* – una estructura dentro de la estructura *vm_context*; los parámetros de instrucción virtual, usado por el *vm_handler*; incluye el *vm_opcode* y argumentos.

Al interpretar el bytecode, la máquina virtual utiliza una pila virtual y un registro virtual único.

- *vm_stack* – un análogo de la pila virtual, que es utilizada por la máquina virtual.
- *vm_register* – un análogo del registro nativo, utilizado por esta máquina virtual; de aquí en adelante, referida como *tmp_REG*.
- *vm_instruction* – una instrucción definida por los desarrolladores de MV; el cuerpo (la implementación) de la instrucción se denomina su *vm_handler*

En las siguientes secciones, describiremos las partes de la máquina virtual en mayor detalle técnico y explicaremos cómo analizarlos.

Un gráfico desofuscado de la función central del malware consiste de tres partes – una parte de inicialización y otras dos que hemos nombrado *vm_start* e intérprete (*vm_dispatcher* + *vm_handlers*).

La parte de inicialización especifica un único identificador de lo que puede ser interpretado como un punto de entrada de bytecode, y lo empuja a la pila. Luego, salta a la parte *vm_start* que es una rutina de inicialización para la máquina virtual en sí misma. Descifra el bytecode y pasa el control al *vm_dispatcher* que hace un loop sobre las instrucciones virtuales del bytecode y las interpreta usando *vm_handlers*. El *vm_dispatcher* comienza con una instrucción pusha y termina con una instrucción `jmp dword ptr [eax+ecx*4]` (o similar), que es un salto al *vm_handler* relevante.

Vm_start

El gráfico creado tras la desofuscación de la primera capa puede verse en la Figura 4. La parte de *vm_start* no es tan relevante para el análisis del intérprete. Sin embargo, puede ayudarnos a entender la implementación completa de la MV; cómo utiliza y maneja banderas virtuales, pila virtual, etc. La segunda parte – el *vm_dispatcher* con *vm_handlers* – es la crucial.

El *vm_start* es solicitado por casi todas las funciones, incluyendo la función central.
La función de solicitud siempre empuja un identificador de instrucción virtual y luego salta a *vm_start*. Cada instrucción virtual tiene su propio identificador virtual. En este ejemplo, el identificador del punto de entrada virtual, donde comienza la ejecución de la función central, es 0x21CD0554. (Figura 5)

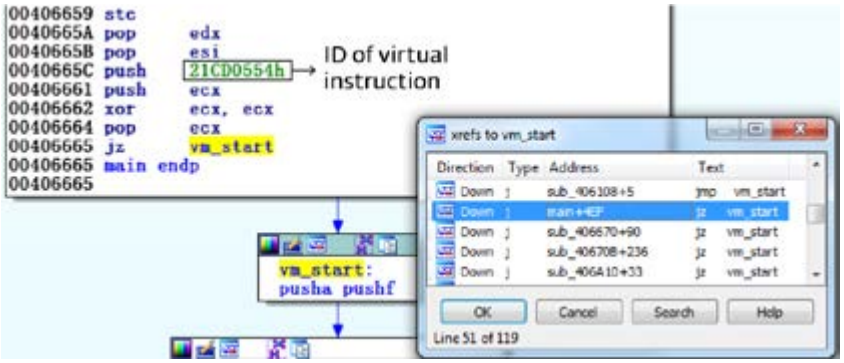

Figura 5 // *vm_start* es solicitado por cada una de las 119 funciones virtualizadas. El ID de la primera instrucción virtual de la respectiva función se da como argumento.

En esta parte, hay bastante código para preparar el *vm_dispatcher* – principalmente para preparar el bytecode y distribuir memoria para el intérprete. Las partes más importantes del código hacen lo siguiente:

1. Asignar 1MB con permiso RWX (lectura, escritura y ejecución) por bytecode y algunas otras variables.
2. Asignar 0x10000 bytes RWX entre variables locales en la máquina virtual para el hilo actual – the *vm_stack*.
3. Descifrar una pieza de código usando una rutina de descifrado XOR. El código descifrado en una rutina desempaquetada aPLib.

La rutina de descifrado XOR utilizada en el ejemplo, es una versión modificada de la rutina clave XOR dword. En realidad, salta la primera de las seis dwords y luego descifra solo las 5 restantes con la llave. Luego está el algoritmo para la rutina (referido de aquí en adelante como XOR decryption_code):

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

4. Solicita la rutina de aPLib para desempaquetar bytecode. Tras hacerlo, los códigos de operación virtuales siguen cifrados. (Figura 6)

Preparar códigos de operación virtuales (pasos 1, 3 y 4) se hace una sola vez – al comienzo – y se salta luego en ejecuciones subsecuentes de vm_start, cuando se ejecutan solo instrucciones para un manejo apropiado de banderas y registros.

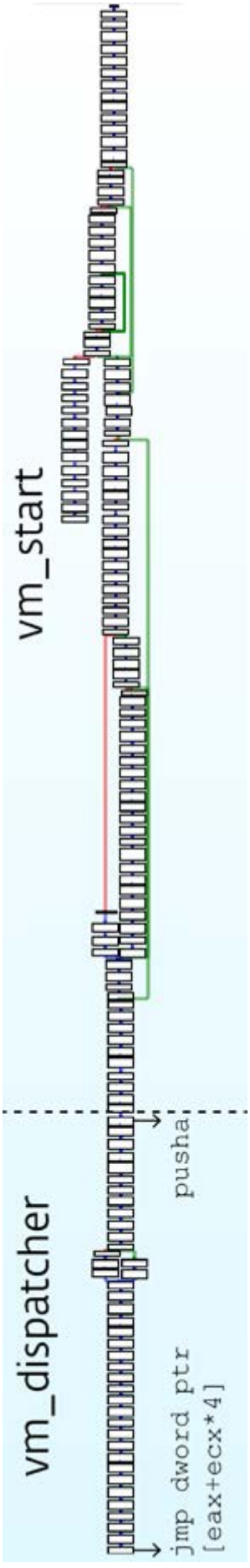

Figura 4 // Gráfico del vm_start y el vm_dispatcher

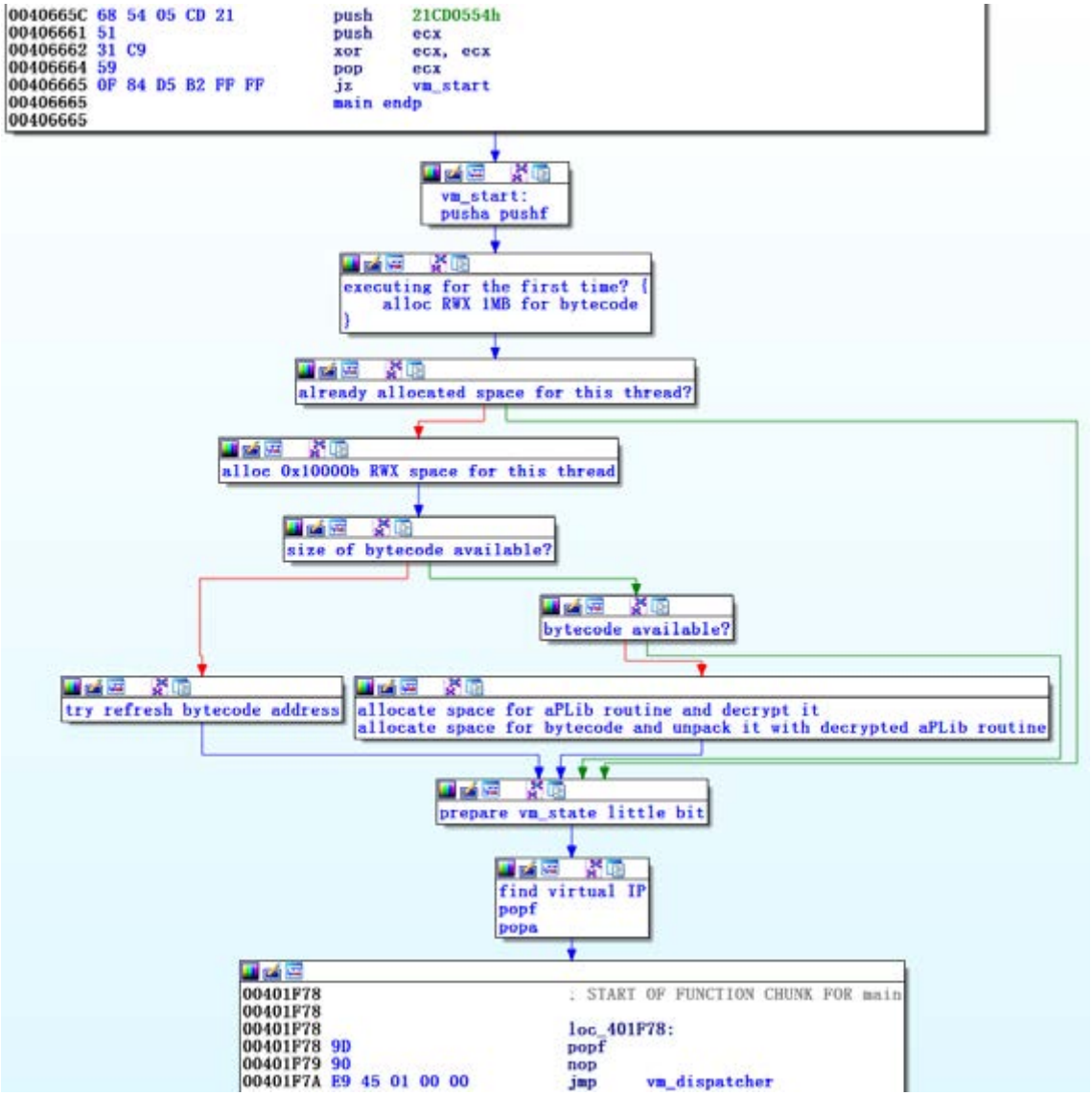

Figure 6 // Todo el código del *vm_start* al *vm_dispatcher* en nodos agrupados nombrados en base a su función. .

INTÉRPRETE DE FINFISHER

Esta parte incluye el *vm_dispatcher* con todos los *vm_handlers* (34 en muestras de FinFisher) y es crucial para analizar y/o desvirtualizar la máquina virtual. El intérprete ejecuta el bytecode.

La instrucción `jmp dword ptr [eax+ecx*4]` salta a uno de los *34 vm_handlers*. Cada *vm_handler* implementa una instrucción de la máquina virtual. Para conocer lo que hace cada *vm_handler*, primero necesitamos comprender el *vm_context* y *vm_dispatcher*.

1.Creando un gráfico IDA

Antes de volcarnos a ello, crear un gráfico bien estructurado puede ayudarnos a entender al intérprete. Recomendamos dividir el gráfico en dos partes – el *vm_start* y el *vm_dispatcher*, por ejemplo, para definir el comienzo de una función en la primera instrucción del *vm_dispatcher*. Lo que todavía falta son los verdaderos *vm_handlers* a los que refiere el *vm_dispatcher*. Para conectar estos handlers

con el gráfico del vm_dispatcher, pueden usarse las siguientes funciones:

```
AddCodeXref(addr_of_jmp_instr,
vm_handler, XREF_USER|fl_JN)
```

añadiendo referencias de la última instrucción del *vm_dispatcher* al inicio de los *vm_handlers*.

```
AppendFchunk
```

añadiendo colas otra vez.

Tras añadir cada *vm_handler* a la función del planificador (dispatcher), el gráfico resultante debería verse como la siguiente figura (Figura 7)

2. Vm_dispatcher

Esta parte es responsable de recuperar y decodificar el bytecode. Realiza los siguientes pasos:

- Ejecuta las instrucciones `pusha` y `pusf` para preparar los registros virtuales y banderas
- Recupera la dirección base de la imagen y la dirección del *vm_stack*
- Lee 24 bytes de bytecode especificando la siguiente *vm_instruction* y sus argumentos.
- Descifra el bytecode con las rutinas de descifrado XOR descritas previamente.

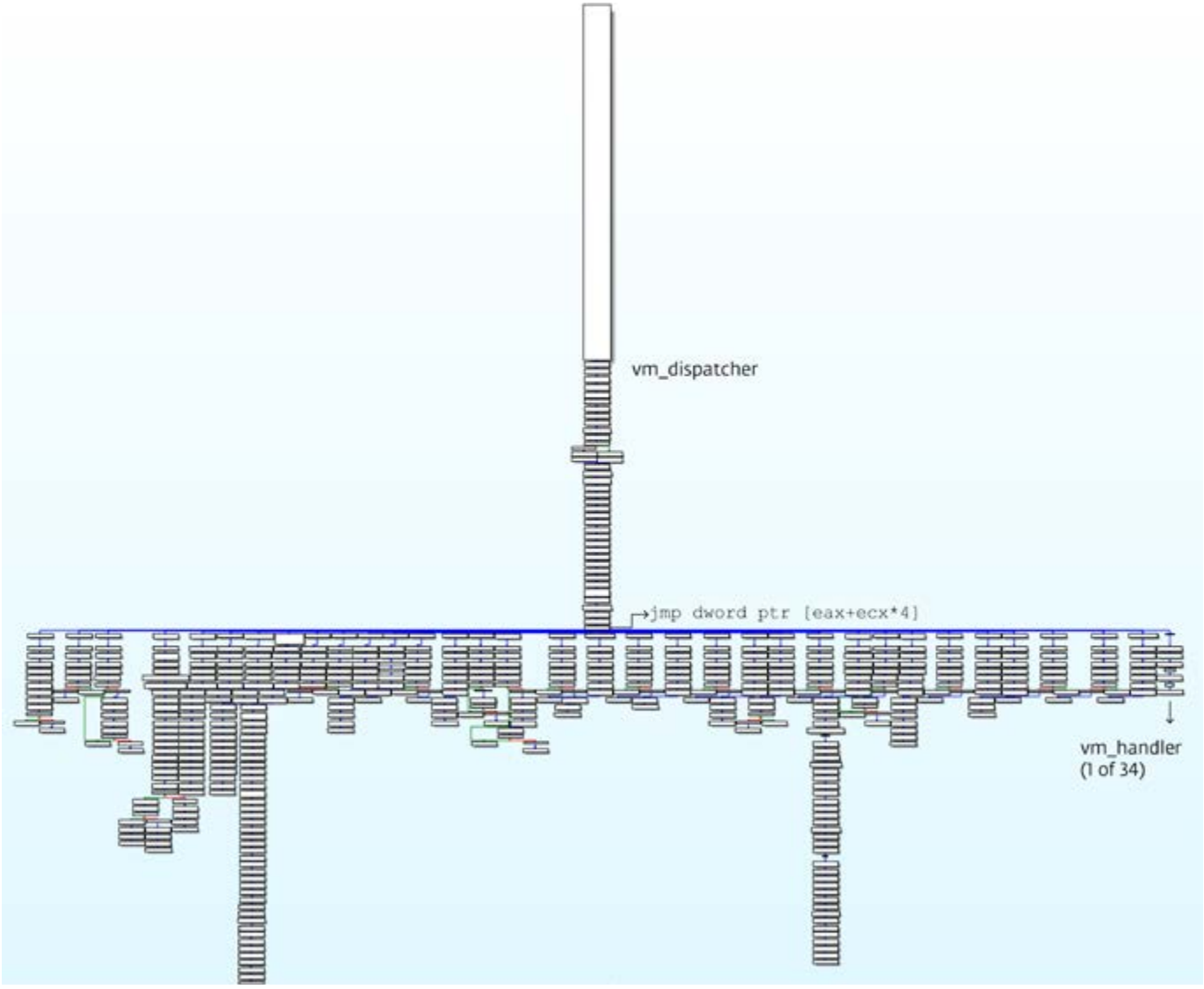

Figure 7 // Gráfico del vm_dispatcher con todos los 34 *vm_handlers*.

- Añade la imagen base al argumento del bytecode en caso que el argumento fuera una variable global.
- Recupera el código de operación virtual (número 0-33) del bytecode descifrado.
- Salta al correspondiente *vm_handler*, que interpreta el código de operación virtual.

Tras haberse ejecutado *vm_handler* para una instrucción, se repite la misma secuencia de pasos para la siguiente, comenzando con la primera instrucción del *vm_dispatcher*.

En cambio, en el caso del handler del *vm_call*, se da el control a la parte del *vm_start* (excepto para instancias en las que siguen funciones no virtualizadas).

3. Vm_context

En esta parte, describiremos el *vm_context* – una estructura usada por la máquina virtual, conteniendo toda la información necesaria para ejecutar el *vm_dispatcher* y cada *vm_handler*.

Al revisar el código del *vm_dispatcher* y los *vm_handlers* en mayor detalle, podemos notar que hay muchas instrucciones de operación de datos, refiriendo a `ebx+offset`, donde offset es un número de 0x00 a 0x50. En la Figura 8, podemos ver cómo se muestra la parte central del *vm_handler* 0x05 en una muestra de FinFisher. (Figura 8)

El registro ebx apunta a una estructura que llamamos *vm_context*. Debemos comprender cómo esta estructura es utilizada – qué son los

Figura 8 // Captura de uno de los vm_handlers

miembros, qué significan y cómo se utilizan. Al resolver este enigma por primera vez, se necesita adivinar un poco para ver cómo se utilizan el *vm_context* y sus elementos.

Por ejemplo, veamos la secuencia de instrucciones al final del *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
```

Ya que sabemos que la última instrucción es un salto al *vm_handler*, podemos concluir que ecx contiene un código de operación virtual, y por lo tanto, el elemento 0x3C de una vm_struct refiere a un número de código de operación virtual.

Hagamos un acierto más. Al final de casi cada *vm_handler*, está la siguiente función:

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

Este mismo miembro del *vm_context* también fue utilizado anteriormente en el código del *vm_dispatcher* – justo antes de saltar al vm_handler. E*vm_dispatcher* copia 24 bytes del miembro de la estructura en una ubicación diferente (`[ebx+38h]`) y lo descifra con la rutina de descifrado XOR para obtener una parte del verdadero bytecode.

Podemos entonces empezar a pensar en el primer elemento del *vm_context* (`[ebx+0h]`) como un *vm_instruction_pointer*, y de la ubicación descifrada (de `[ebx+38h]` a `[ebx+50h]`) como una ID de la instrucción virtual, su código de operación y argumentos. Juntos, llamaremos a esta estructura *vi_params*.

Siguiendo los pasos arriba descritos y utilizando un depurador para ver qué valores se almacenan en los respectivos elementos de las estructuras, podemos hallar a todos los miembros del *vm_context*.

Tras el análisis, podemos reconstruir tanto el vm_context como la estructura *vi_params* de FinFisher:

```
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. Implementando instrucciones virtuales – vm_ handlers

Cada *vm_handler* se ocupa de un código de operación virtual – ya que hay 23 *vm_handlers*, hay como máximo 34 códigos de operación virtuales. Ejecutar un *vm_handler* significa ejecutar una *vm_instruction*, por ello, para revelar qué hace una *vm_instruction*, necesitamos analizar el *vm_handler* correspondiente.

Luego de reconstruir el *vm_context* y nombrar todos los offsets de ebx, el *vm_handler* mostrado previamente pasa a ser uno mucho más legible, como se observa en la Figura 9.

Al final de esta función, notamos una secuencia de instrucciones, empezando con el *vm_instruction_pointer*, incrementándose por 24 el tamaño de cada estructura vi_params de las *vm_instruction*. Ya que esta secuencia se repite al final de prácticamente cada *vm_handler*, concluimos que es el epílogo estándar de una función y que el cuerpo actual del *vm_handler* puede escribirse simplemente como:

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

Hemos analizado la primera instrucción de la máquina virtual. :-)

Figura 9 // TEl vm_ handler previo tras insertar la estructura *vm_context*

Para ilustrar cómo funciona la instrucción analizada cuando la ejecutamos, consideremos la estructura *vi_params* completada de la siguiente manera:

```
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;
};
```

Por lo que hemos mostrado arriba, podemos ver que la siguiente función se ejecutará:

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

En este punto, deberíamos entender lo que hace una de las *vm_instructions*. Los pasos que hemos seguido deberían servir como una demostración decente de cómo funciona el intérprete completo.

Sin embargo, hay algunos *vm_handlers* más difíciles de analizar. Estos saltos condicionales son engañosos y difíciles de entender por la forma en que traducen las banderas.

Como dijimos, el *vm_dispatcher* comienza introduciendo EFLAGS nativos (de vm_code) a la cima de la pila nativa. Por lo tanto, cuando el handler para un salto específico está decidiendo si saltar o no, revisa las EFLAGS de la pila nativa e implementa si propio método de salto. La Figura 10 ilustra cómo el handler JNP virtual se implementa chequeando la bandera de paridad (Figura 10)

Figura 10 // Captura de un *JNP_handler*

Para otros saltos condicionales virtuales, podría ser necesario revisar varias banderas – por ejemplo, el resultado del salto de la función JBE virtualizada depende de los valores de ambos CF y ZF – pero el principio se mantiene.

Tras analizar los 34 *vm_handlers* en la máquina virtual de FinFisher, podemos describir sus instrucciones virtuales de la siguiente manera:

```
.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* con los 34 *vm_handlers* accedidos

Considera que la palabra clave "*tmp_REG*" refiere al registro virtual usado por el registro temporario de la máquina virtual en la estructura del *vm_context*, mientras "reg" refiere a un registro nativo, por ejemplo eax.

Veamos las instrucciones analizadas de la máquina virtual. Por ejemplo, *case_3_vm_ jcc* es un handler de salto general que puede ejecutar cualquier salto nativo, ya sea condicional o incondicional.

Aparentemente, esta máquina virtual no virtualiza cada instrucción nativa – y es aquí donde se vuelven útiles las instrucciones de los casos 4 y 6.

Estos dos *vm_handlers* son implementados para ejecutar código nativo de forma directa – todo lo que hacen es leer el código de operación de una instrucción nativa dada como argumento y ejecuta la instrucción.

Otro aspecto a tener en cuenta es que los *vm_registers* siempre están en la cima de la pila nativa, mientras el identificador del registro a utilizarse se almacena en el último byte de arg0 de la instrucción virtual. El siguiente código

```
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)
```

puede usarse para acceder al registro virtual correspondiente:

5. Escribiendo tu propio desensamblador

Tras haber analizado correctamente todas las *vm_instructions*, existe todavía un paso a realizar antes de comenzar con el análisis de la muestra – debemos escribir nuestro propio desensamblador para el bytecode (analizarlo manualmente sería problemático, dado su tamaño).

Al hacer el esfuerzo y escribir un desensamblador más robusto, podemos ahorrarnos algunos problemas cuando la máquina virtual de FinFisher se cambie y actualice.

Comencemos con el *vm_handler* 0x0C, que ejecuta la siguiente instrucción:

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

Esta instrucción toma exactamente un argumento – el identificador de un registro nativo a utilizarse como `reg`. Este identificador debe corresponderse con un nombre de registro

```
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
```

nativo, utilizando, por ejemplo, una función `resolve_reg` como se mencionó anteriormente.

El siguiente código puede utilizarse para desensamblar este *vm_handler*:

Nuevamente, los *vm_handlers* para saltos son más difíciles de entender. En el caso de saltos, los miembros *vm_context.vi_params.Arg0* y *vm_context. vi_params.Arg1* almacenan el offset por el cual saltar. Este “jump offset” es en realidad un offset en el bytecode. Cuando se analizan saltos, debemos poner un marcador en la ubicación a la cual se salta. Por ejemplo, este código puede utilizarse:

```
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
```

Finalmente, hay un *vm_handler* responsable de ejecutar instrucciones nativas desde argumentos, que necesita trato especial. Para ello, debemos utilizar un desensamblador para instrucciones x86 nativas – por ejemplo, la herramienta de código abierto Distorm.

La longitud de una instrucción se almacena en *vm_context.vi_params.OpCode & 0x0000FF00*. El código de operación de la instrucción nativa que va a ejecutarse es almacenado en los argumentos. El siguiente

código puede utilizarse para analizar el *vm_handler* que ejecuta código nativo:

```
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
```

Hasta ahora, hemos creado funciones en Python para desensamblar cada *vm_handler*. Todas ellas, combinadas con el código responsable de crear locaciones de saltos, hallar el ID de una instrucción virtual tras la solicitud y algunas otras, son necesarias para escribir tu propio desensamblador.

Luego, podemos ejecutar el desensamblador terminado en el bytecode.

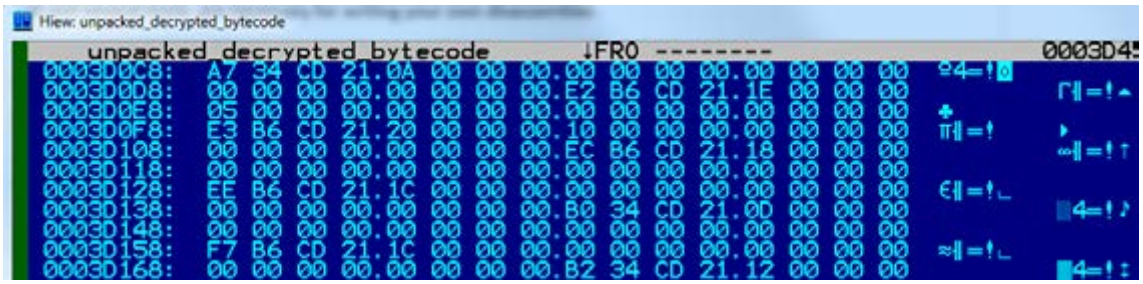

Figura 12 // Parte del bytecode desempaquetado y descifrado de FinFisher

Por ejemplo, de la parte del bytecode mostrada en la Figura 12, podemos obtener el siguiente resultado:

```
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. Comprendiendo la implementación de esta máquina virtual

Tras haber analizado todos los *handlers* y construido un desensamblador propio, podemos ver una vez más las instrucciones virtuales para hacernos una idea general de cómo fueron creadas.

Primero, debemos comprender que la protección de la virtualización fue implementada en la etapa de ensamblado. Los autores tradujeron instrucciones nativas a sus propias y complejas instrucciones, que se ejecutarán en un CPU virtual personalizado. Para lograr esto, se utiliza un "registro" temporal (*tmp_REG*).

Veamos algunos ejemplos para entender cómo funciona esta traducción. Por ejemplo, la instrucción virtual del ejemplo anterior –

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

– fue traducida de la instrucción nativa original `push eax`. Al virtualizarse, se utilizó un registro temporal en un paso intermedio para modificar la instrucción y hacerla más compleja.Veamos otro ejemplo:

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

Las instrucciones nativas que fueron traducidas a estas instrucciones virtualizadas, fueron las siguientes (siendo reg uno de los registros nativos):

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

Sin embargo, no es esta la única manera de virtualizar un conjunto de instrucciones. Existen otros protectores de máquinas virtuales con diferentes abordajes. Por ejemplo, uno de los protectores comerciales traduce cada instrucción de operación matemática en lógica NOR, utilizándose un número de registros temporales, en lugar de uno.

Contrariamente, la máquina virtual de FinFisher no llegó tan lejos como para cubrir todas las instrucciones nativas. Si bien muchas de ellas pueden ser virtualizadas, otras no – instrucciones matemáticas como `add`, `imul` y `div`, por ejemplo. Si estas instrucciones aparecen en el binario original, se acude al *vm_handler* encargado de ejecutar instrucciones nativas para que las maneje en el binario protegido. El único cambio es que las EFLAGS y los registros nativos son eliminados de la pila nativa justo antes de que se ejecute la instrucción nativa, y vueltos a insertar tras ser ejecutada. De esta manera, las instrucciones de pop y push permiten guardar un registro de la pila y recuperarlo luego, y es así como se evitó la virtualización de cada instrucción nativa.

Una gran desventaja al proteger binarios con una máquina virtual es el impacto en el desempeño. En el caso de la máquina virtual de FinFisher, estimamos una ralentización de más de 600 veces que el código nativo, con base en el número de instrucciones que debe ejecutarse para llevar a cabo cada *vm_instruction* (*vm_dispatcher + vm_handler*).

Por ello, resulta lógico proteger solo ciertas partes del binario – y este es también el caso en las muestras que analizamos de FinFisher.

Además, como mencionamos anteriormente, algunos handlers de máquinas virtuales pueden invocar funciones nativas de forma directa. Como resultado, los usuarios de la protección de máquina virtual (por ejemplo, los autores de FinFisher) pueden ver las funciones al momento de ensamblado y marcar cuáles deben ser protegidas por la máquina virtual. Para aquellas que fueron marcadas, sus instrucciones serán virtualizadas; para las que no, las funciones originales serán solicitadas por el handler virtual correspondiente. En consecuencia, la ejecución podría consumir menos tiempo, mientras que las partes más interesantes del binario se mantienen protegidas. (Figura 13)

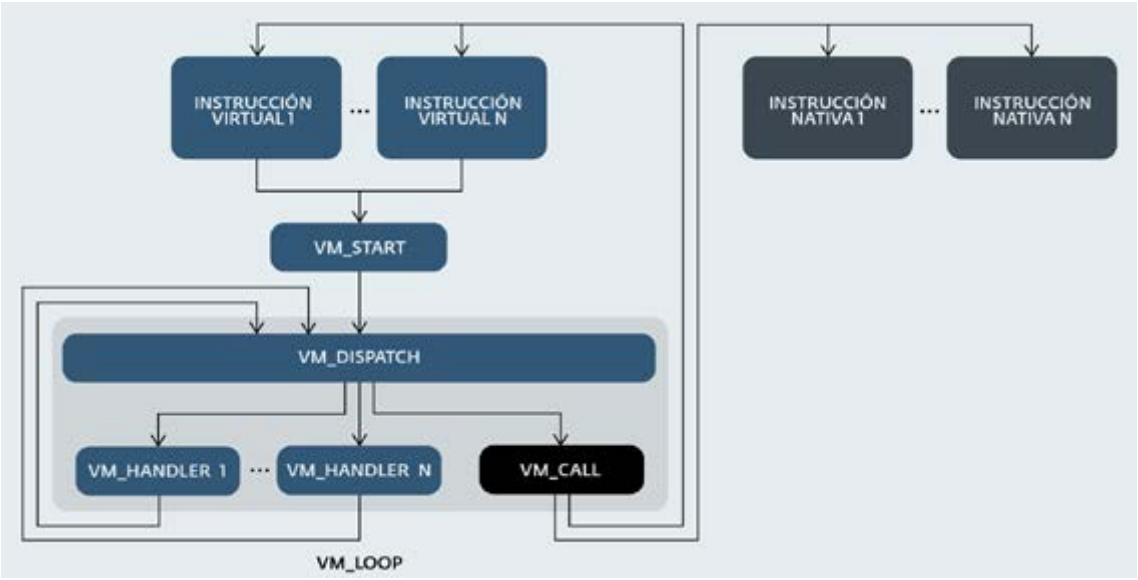

Figura 13 // Esquema que representa la protección de máquina virtual completa y cómo la ejecución puede pasar por fuera de la mv

7. Automatizando el proceso desensamblador para más muestras de FinFisher

Además de la extensión del bytecode que debe procesar nuestro analizador sintáctico, debemos tener en cuenta que hay cierta aleatoriedad alrededor de varias muestras de FinFisher. Si bien se ha utilizado la misma máquina virtual para la protección, el flujo de datos entre los códigos de operación virtuales y los *vm_handlers* no siempre es constante. Estos pueden ser (y son) agrupados de forma aleatoria y diferente para cada muestra de FinFisher que analizamos. Esto significa que si el *vm_handler* para el código virtual 0x5 en esta muestra lleva a cabo la instrucción `mov [tmp_REG], arg0` se le podría asignar otro código de operación virtual en otra muestra protegida.

Para abordar este problema, podemos utilizar una firma por cada *vm_handler* analizado. El script de IDA Python en el Apéndice A puede aplicarse luego de haberse generado un gráfico, como se mostró en la Figura 7 (es particularmente importante eliminar la ofuscación del salto jz/jnz – como se describe en la primera sección de esta guía) para nombrar los handlers con base en sus firmas. (Con una pequeña modificación, el script también puede ser utilizado para recrear las firmas si los *vm_handlers* son modificados en una futura actualización de FinFisher.)

Como se mencionó, el primer *vm_handler* en la muestra de FinFisher que analizas puede diferir de JL, como en el ejemplo de la muestra de FinFisher, pero el script identificará todos los *vm_handlers* correctamente.

8. Compilando código desensamblado sin la MV

Luego del desensamblaje y tras algunas modificaciones, es posible compilar el código. Trataremos las instrucciones virtuales como instrucciones nativas. Como resultado, obtendremos un binario puro, sin la protección.

La mayoría de las *vm_instructions* pueden compilarse de forma inmediata utilizando copiar-pegar, ya que el resultado de nuestro desensamblador consiste, principalmente, de instrucciones que parecen nativas. Pero algunos casos requieren tratamiento especial:

- *tmp_REG* – Una vez definido *tmp_REG* como una variable global, necesitamos

ajustar el código para casos en los que una dirección allí almacenada no está siendo referenciada. (Ya que no es posible referenciar una dirección que está en una variable global en el conjunto de instrucciones x86.) Por ejemplo, la MV contiene la instrucción virtual *mov tmp_REG, [tmp_REG]* que necesita se reescrita de la siguiente manera:

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

- Banderas – Las instrucciones virtuales no modifican las banderas, pero las instrucciones matemáticas nativas sí. Por ello, debemos asegurar que las instrucciones matemáticas virtuales tampoco modificarán banderas en el binario desvirtualizado, lo que significa que debemos guardarlas antes de ejecutar esta instrucción y recuperarlas tras la ejecución.
- Saltos y llamados – debemos poner un marcador a la instrucción virtual de destino (saltos) o función (llamadas/solicitudes).
- Solicitudes de función API – en la mayoría de los casos, las funciones API se cargan dinámicamente, mientras que en otras son referenciadas del IAT (*Import Address Table*) del binario, por lo que estos casos requieren un manejo acorde.
- Variables globales, código nativo – Algunas variables globales deben ser mantenidas en el binario desvirtualizado. Además, en el dropper de FinFisher, hay una función para mutar a x64 de x86 que se ejecuta de forma nativa (en verdad, se realiza únicamente con la instrucción `retf`). Todos estos deben mantenerse en el código al momento de compilar.

Dependiendo del resultado del desensamblador, podrías necesitar hacer todavía algunas modificaciones más, para obtener las instrucciones nativas puras que pueden ser compiladas. Luego, puedes compilar el código con tu ensamblador-compilador favorito a un binario sin la MV.

CONCLUSIÓN

En esta guía, hemos descrito cómo FinFisher utiliza dos técnicas elaboradas para proteger su principal carga útil (payload). La intención principal de esta protección no es evadir la detección antivirus, sino cubrir los archivos de configuración y nuevas técnicas implementadas en el spyware, al dificultar su análisis con ingeniería inversa. Ya que, hasta el día de la fecha, no se ha publicado otro análisis detallado de la ofuscación del spyware FinFisher, parece ser que los desarrolladores de estos mecanismos de protección han sido exitosos.

Hemos mostrado cómo podemos sobrepasar la capa de anti-desensamblación de forma automática, y cómo la máquina virtual puede ser analizada de forma eficiente.

Esperamos que esta guía ayude al análisis, mediante ingeniería inversa, de muestras de FinFisher protegidas por máquina virtual, así como a comprender en profundidad otros protectores de máquina virtual en general.

APÉNDICE A

Script de IDA Python para nombrar los *vm_handler* de FinFisher

El script también está disponible en el repositorio GitHub de ESET:
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', '8d4b408b432c8b0a94000075a90
800f95c2a980000f95c03ac275ff631c' : 'case_2_JLE_loc1', '8d4b408b7b508b432c83e02f8
dbc38311812b5c787cfe7ed4ae92f8b066c787d3e7e
4af9b8e80000588d80' : 'case_3_vm_jcc', '8b7b508b432c83e02f3f85766c77ac66681373167
83c728d7340fb64b3df3a4c67e98037818b43c89471c64756c80775af83318588b632c' : 'case_4
_exec_native_code', '8d4b408b98b438898833188b43c8b632c' : 'case_5_mov_tmp_REGref_
arg0', '8b7b508b432c83e02f3f85766c77ac6668137316783c728d7340fb64b3df3a4c67e980378
18b43c89471c64756c80775af83318588b632c' : 'case_6_exec_native_code', '8d4b408b432
c8b0a94000075ff631c' : 'case_7_JZ_loc1', '8d4b408b432c8b0a94000075a90800f95c2a980
000f95c03ac275ff6318' : 'case_8_JG_loc1', '8d43408b089438833188b43c8b632c' : 'cas
e_9_mov_tmp_REG_arg0', '33c9894b8833188b632c8b43c' : 'case_A_zero_tmp_REG', '8d4b
408b432c8b0a98000075ff631c' : 'case_B_JS_loc1', '8d4b40fb69b870002bc18b4b2c8b5481
48b4b88911833188b43c8b632c' : 'case_C_mov_tmp_REGDeref_tmp_REG', '8d4b40fb69b8700
02bc18b4b2c8b44814894389438833188b43c8b632c' : 'case_D_mov_tmp_REG_tmp_REG', '8d4b408
b432c8b0a9100075ff631c' : 'case_E_JB_loc1', '8d4b408b432c8b0a9100075a94000075ff63
1c' : 'case_F_JBE_loc1', '8d4b408b432c8b0a94000074ff631c' : 'case_10_JNZ_loc1', '
8d4b408b432c8b0a9080074ff631c' : 'case_11_JNO_loc1', '8b7b50834350308d4b408b41434
3285766c773f50668137a231c6472c280772aa8d57d83c73891783ef3c7477a300080777cb83c7889
783ef8c647cf28077c3183c7dc67688b383c0188947183c7566c7777fe668137176283c72c672d803
745895f183c75c67848037df478b4314c67408037288947183c75c67928037515f8b632c' : 'case
_12_vm_call', '8d4b40b870002b18b532c8b44824894389438833188b43c8b632c' : 'case_13_mov_
tmp_REG_tmp_REG_notRly', '8d4b408b432c8b0a9400075ff631c' : 'case_14_JP_loc1', '8d
4b40fb69b870002bc18b4b2c8b53889548148338833188b43c8b632c' : 'case_15_mov_tmp_REG_tmp_
REG', '8d4b408b432c8b0a9080075ff631c' : 'case_16_JO_loc1', '8d4b408b432c8b0a90800
f95c2a980000f95c03ac274ff631c' : 'case_17_JGE_loc1', '8b4388b089438833188b43c8b63
2c' : 'case_18_deref_tmp_REG', '8d4b408b4388b9d3e089438833188b43c8b632c' : 'case_
19_shl_tmp_REG_arg0l', '8d4b408b432c8b0a98000074ff631c' : 'case_1A_JNS_loc1', '8d
4b408b432c8b0a9100074ff631c' : 'case_1B_JNB_loc1', '8b7b2c8b732c83ef4b924000fcf3a
4836b2c48b4b2c8b438894124833188b43c8b632c' : 'case_1C_push_tmp_REG', '8d4b408b432
c8b0a94000075a9100075ff6318' : 'case_1D_JA_loc1', '8d4b40b870002b18b532c8b4482414
38833188b43c8b632c' : 'case_1E_add_stack_val_to_tmp_REG', '8b7b508343503066c77ac3
766813731565783c728d4b40c672e803746fb6433d3c783c058947183c758d714fb64b3df3a45ac67
1280377a8b383c0188947183c7566c777f306681371fac83c72c671f803777895f183c75c67708037
2b47c6798037618b4b14894f183c75c67778037b48b632c8d12' : 'case_1F_vm_jmp', '8d4b408
b914b8833188b43c8b632c' : 'case_20_add_arg0_to_tmp_REG', '8d4b408b98b438891833188
b632c8b43c' : '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)
```