# FinFisher exposed: A researcher's tale of defeating traps, tricks, and complex virtual machines | Microsoft Security Blog

By Microsoft Threat Intelligence

Published: 2018-03-01 · Archived: 2026-04-05 12:48:26 UTC

Office 365 Advanced Threat Protection (Office 365 ATP) blocked many notable zero-day exploits in 2017. In our analysis, one activity group stood out: NEODYMIUM. This threat actor is remarkable for two reasons:

- Its access to sophisticated zero-day exploits for Microsoft and Adobe software
- Its use of an advanced piece of government-grade surveillance spyware FinFisher, also known as FinSpy and detected by Microsoft security products as Wingbird

FinFisher is such a complex piece of malware that, like other researchers, we had to devise special methods to crack it. We needed to do this to understand the techniques FinFisher uses to compromise and persist on a machine, and to validate the effectiveness of Office 365 ATP detonation sandbox, Windows Defender Advanced Threat Protection (Windows Defender ATP) generic detections, and other Microsoft security solutions.

This task proved to be nontrivial. FinFisher is not afraid of using all kinds of tricks, ranging from junk instructions and "spaghetti code" to multiple layers of virtual machines and several known and lesser-known anti-debug and defensive measures. Security analysts are typically equipped with the tools to defeat a good number of similar tricks during malware investigations. However, FinFisher is in a different category of malware for the level of its anti-analysis protection. It's a complicated puzzle that can be solved by skilled reverse engineers only with good amount of time, code, automation, and creativity. The intricate anti-analysis methods reveal how much effort the FinFisher authors exerted to keep the malware hidden and difficult to analyze.

This exercise revealed tons of information about techniques used by FinFisher that we used to make Office 365 ATP more resistant to sandbox detection and Windows Defender ATP to catch similar techniques and generic behaviors. Using intelligence from our in-depth investigation, Windows Defender ATP can raise alerts for malicious behavior employed by FinFisher (such as memory injection in persistence) in different stages of the attack kill chain. Machine learning in Windows Defender ATP further flags suspicious behaviors observed related to the manipulation of legitimate Windows binaries.

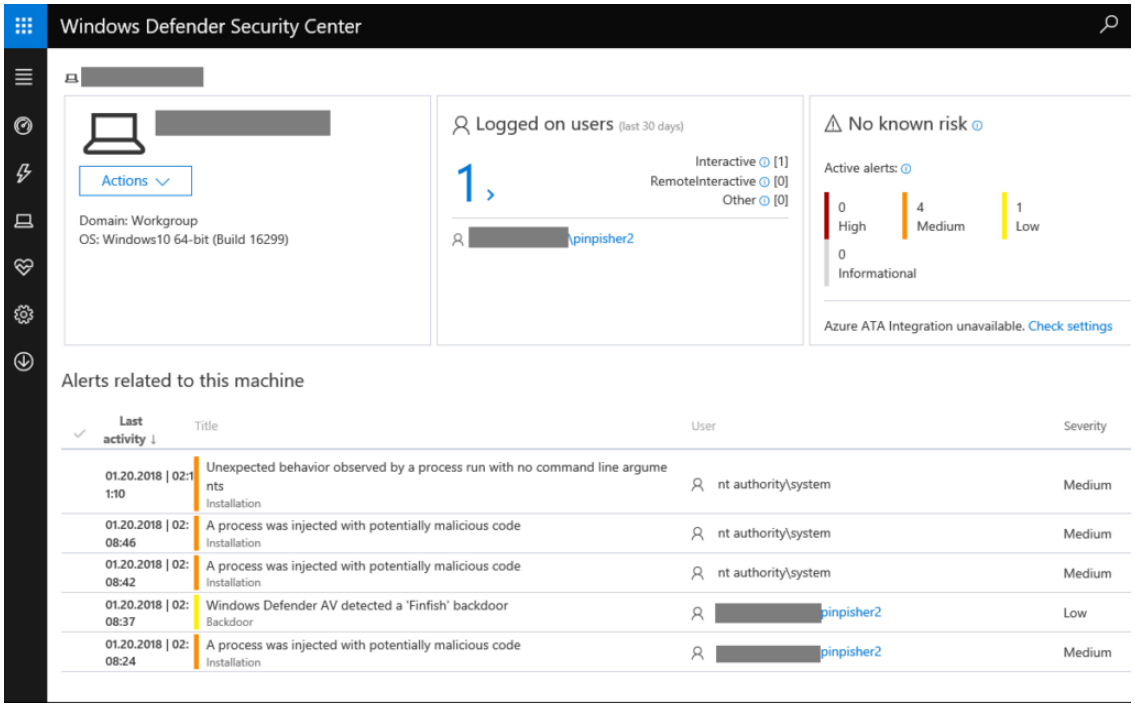


*Figure 1. Generic Windows Defender ATP detections trigger alerts on FinFisher behavior*

While our analysis has allowed us to immediately protect our customers, we'd like to share our insights and add to the growing number of published analyses by other talented researchers (listed below this blog post). We hope that this blog post helps other researchers to understand and analyze FinFisher samples and that this industry-wide information-sharing translate to the protection of as many customers as possible.

In analyzing FinFisher, the first obfuscation problem that requires a solution is the removal of junk instructions and "spaghetti code", which is a technique that aims to confuse disassembly programs. Spaghetti code makes the program flow hard to read by adding continuous code jumps, hence the name. An example of FinFisher's spaghetti code is shown below.

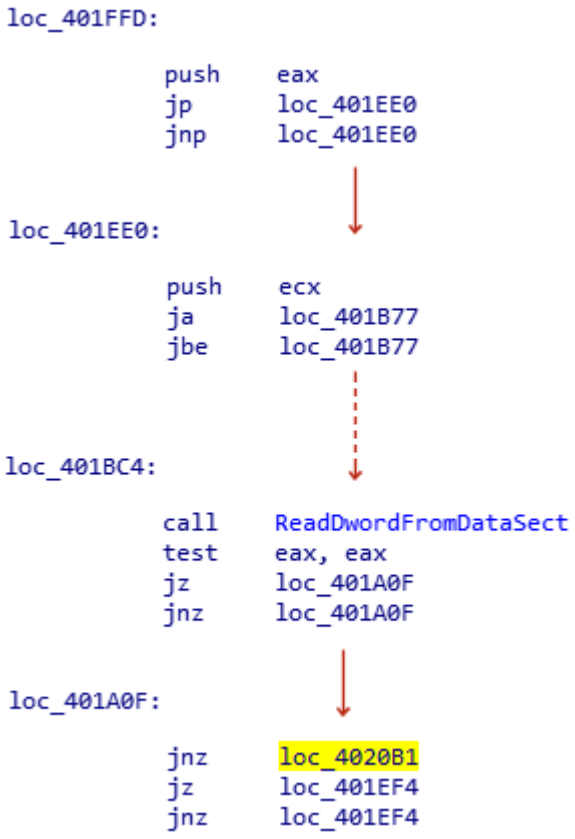


*Figure 2. The spaghetti code in FinFisher dropper*

This problem is not novel, and in common situations there are known reversing plugins that may help for this task. In the case of FinFisher, however, we could not find a good existing interactive disassembler (IDA) plugin that can normalize the code flow. So we decided to write our own plugin code using IDA Python. Armed with this code, we removed this first layer of anti-analysis protection.

Removing the junk instructions revealed a readable block of code. This code starts by allocating two chunks of memory: a global 1 MB buffer and one 64 KB buffer per thread. The big first buffer is used as index for multiple concurrent threads. A big chunk of data is extracted from the portable executable (PE) file itself and decrypted two times using a custom XOR algorithm. We determined that this chunk of data contains an array of opcode instructions ready to be interpreted by a custom virtual machine program (from this point on referenced generically as “VM”) implemented by FinFisher authors.

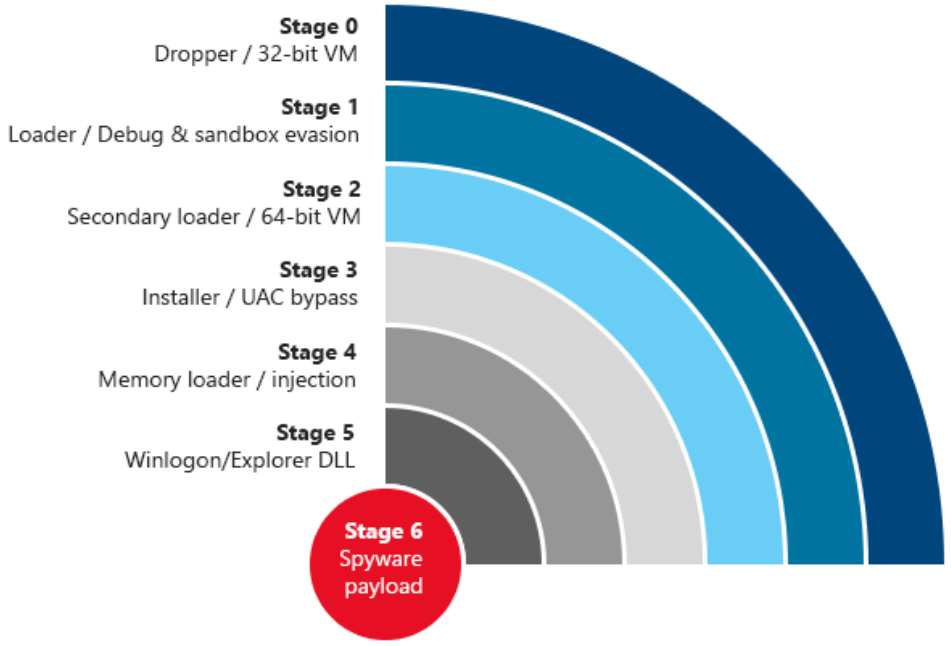


*Figure 3. The stages of the FinFisher multi-layered protection mechanisms*

## Stage 0: Dropper with custom virtual machine

The main dropper implements the VM dispatcher loop and can use 32 different opcodes handlers. Th 64KB buffer is used as a VM descriptor data structure to store data and the just-in-time (JIT) generated code to run. The VM dispatcher loop routine ends with a JMP to another routine. In total, there are 32 different routines, each of them implementing a different opcode and some basic functionality that the malware program may execute.

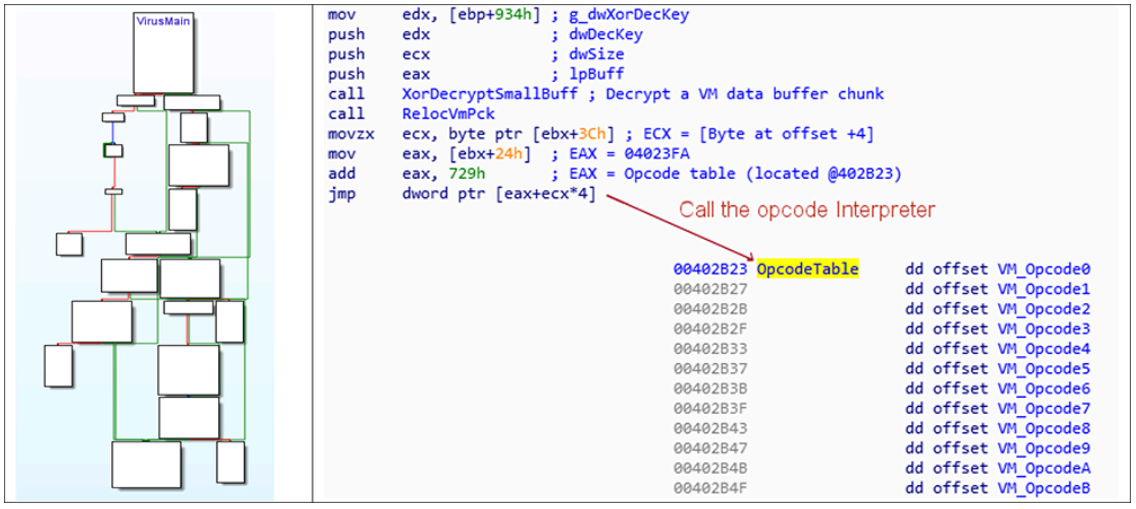


*Figure 4. A snapshot of the code that processes each VM opcode and the associate interpreter*

The presence of a VM and virtualized instruction blocks can be described in simpler terms: Essentially, the creators of FinFisher interposed a layer of dynamic code translation (the virtual machine) that makes analysis using regular tools practically impossible. Static analysis tools like IDA may not be useful in analyzing custom code that is interpreted and executed through a VM and a new set of instructions. On the other hand, dynamic

analysis tools (like debuggers or sandbox) face the anti-debug and anti-analysis tricks hidden in the virtualized code itself that detects sandbox environments and alters the behavior of the malware.

At this stage, the analysis can only continue by manually investigating the individual code blocks and opcode handlers, which are highly obfuscated (also using spaghetti code). Reusing our deobfuscation tool and some other tricks, we have been able to reverse and analyze these opcodes and map them to a finite list that can be used later to automate the analysis process with some scripting.

The opcode instructions generated by this custom VM are divided into different categories:

1. Logical opcodes, which implement bit-logic operators (OR, AND, NOT, XOR) and mathematical operators
2. Conditional branching opcodes, which implement a code branch based on conditions (equals to JC, JE, JZ, other similar branching opcodes)
3. Load/Store opcodes, which write to or read from particular addresses of the virtual address space of the process
4. Specialized opcodes for various purposes, like execute specialized machine instruction that are not virtualized

We are publishing below the (hopefully) complete list of opcodes used by FinFisher VM that we found during our analysis and integrated into our de-virtualization script:

| INDEX | MNEMONIC | DESCRIPTION |
|---|---|---|
| 0x0 | EXEC | Execute machine code |
| 0x1 | JG | Jump if greater/Jump if not less or equal |
| 0x2 | WRITE | Write a value into the dereferenced internal VM value (treated as a pointer) |
| 0x3 | JNO | Jump if not overflow |
| 0x4 | JLE | Jump if less or equal (signed) |
| 0x5 | MOV | Move the value of a register into the VM descriptor (same as opcode 0x1F) |
| 0x6 | JO | Jump if overflow |
| 0x7 | PUSH | Push the internal VM value to the stack |
| 0x8 | ZERO | Reset the internal VM value to 0 (zero) |
| 0x9 | JP | Jump if parity even |
| 0xA | WRITE | Write into an address |
| 0xB | ADD | Add the value of a register to the internal VM value |

| | | |
|---|---|---|
| 0xC | JNS | Jump if not signed |
| 0xD | JL | Jump if less (signed) |
| 0xE | EXEC | Execute machine code and branch |
| 0xF | JBE | Jump if below or equal or Jump if not above |
| 0x10 | SHL | Shift left the internal value the number of times specified into the opcodes |
| 0x11 | JA | Jump if above/Jump if not below or equal |
| 0x12 | MOV | Move the internal VM value into a register |
| 0x13 | JZ | JMP if zero |
| 0x14 | ADD | Add an immediate value to the internal Vm descriptor |
| 0x15 | JB | Jump if below (unsigned) |
| 0x16 | JS | Jump if signed |
| 0x17 | EXEC | Execute machine code (same as opcode *0x0*) |
| 0x18 | JGE | Jump if greater or equal/Jump if not less |
| 0x19 | DEREF | Write a register value into a dereferenced pointer |
| 0x1A | JMP | Special obfuscated “Jump if below” opcode |
| 0x1B | * | Resolve a pointer |
| 0x1C | LOAD | Load a value into the internal VM descriptor |
| 0x1D | JNE | Jump if not equal/Jump if not zero |
| 0x1E | CALL | Call an external function or a function located in the dropper |
| 0x1F | MOV | Move the value of a register into the VM descriptor |
| 0x20 | JNB | Jump if not below/Jump if above or equal/Jump if not carry |
| 0x21 | JNP | Jump if not parity/Jump if parity odd |

Each virtual instruction is stored in a special data structure that contains all the information needed to be properly read and executed by the VM. This data structure is 24 bytes and is composed of some fixed fields and a variable portion that depends on the opcode. Before interpreting the opcode, the VM decrypts the opcode’s content (through a simple XOR algorithm), which it then relocates (if needed), using the relocation fields.

Here is an approximate diagram of the opcode data structure:

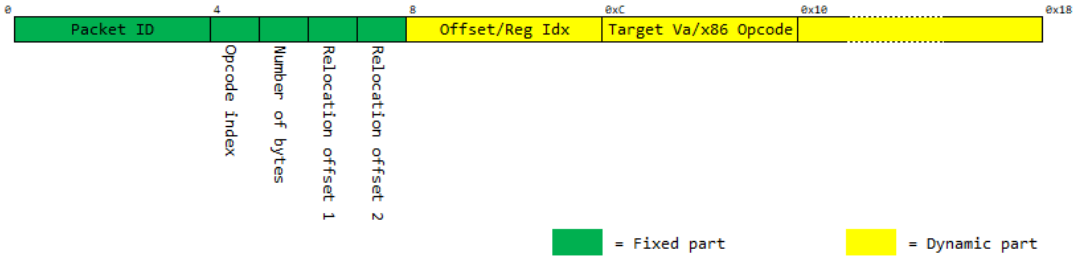


*Figure 5. A graphical representation of the data structure used to store each VM opcode*

The VM handler is completely able to generate different code blocks and deal with relocated code due to address space layout randomization (ASLR). It is also able to move code execution into different locations if needed. For instance, in the case of the "Execute" opcode (*0x17*), the 32-bit code to run is stored entirely into the variable section with the value at offset 5 specifying the number of bytes to be copied and executed. Otherwise, in the case of conditional opcodes, the variable part can contain the next JIT packet ID or the next relative virtual address (RVA) where code execution should continue.

Of course, not all the opcodes are can be easily read and understood due to additional steps that the authors have taken to make analysis extremely complicated. For example, this is how opcode *0x1A* is implemented: The opcode should represent a JB (Jump if below) function, but it's implemented through set carry (STC) instruction followed by a JMP into the dispatcher code that will verify the carry flag condition set by STC.

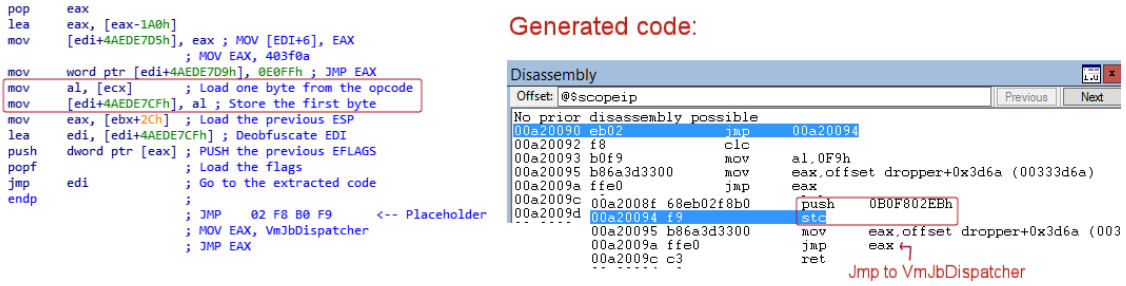


*Figure 6. One of the obfuscation tricks included by the malware authors in a VM opcode dispatcher*

Even armed with the knowledge we have described so far, it still took us many hours to write a full-fledged opcode interpreter that's able to reconstruct the real code executed by FinFisher.

## Stage 1: Loader malware keeps sandbox and debuggers away

The first stage of FinFisher running through this complicated virtual machine is a loader malware designed to probe the system and determine whether it's running in a sandbox environment (typical for cloud-based detonation solution like Office 365 ATP).

The loader first dynamically rebuilds a simple import address table (IAT), resolving all the API needed from Kernel32 and NtDll libraries. It then continues executing in a spawned new thread that checks if there are additional undesired modules inside its own virtual address space (for example, modules injected by certain

security solutions). It eventually kills all threads that belong to these undesired modules (using *ZwQueryInformationThread* native API with *ThreadQuerySetWin32StartAddress* information class).

The first anti-sandbox technique is the loader checking the code segment. If it's not *0x1B* (for 32-bit systems) or *0x23* (for 32-bit system under Wow64), the loader exits.

Next, the dropper checks its own parent process for indications that it is running in a sandbox setup. It calculates the MD5 hash of the lower-case process image name and terminates if one of the following conditions are met:

1. The MD5 hash of the parent process image name is either D0C4DBFA1F3962AED583F6FCE666F8BC or 3CE30F5FED4C67053379518EACFCF879
2. The parent process's full image path is equal to its own process path

If these initial checks are passed, the loader builds a complete IAT by reading four imported libraries from disk (*ntdll.dll*, *kernel32.dll*, *advapi32.dll*, and *version.dll*) and remapping them in memory. This technique makes use of debuggers and software breakpoints useless. During this stage, the loader may also call a certain API using native system calls, which is another way to bypass breakpoints on API and security solutions using hooks.

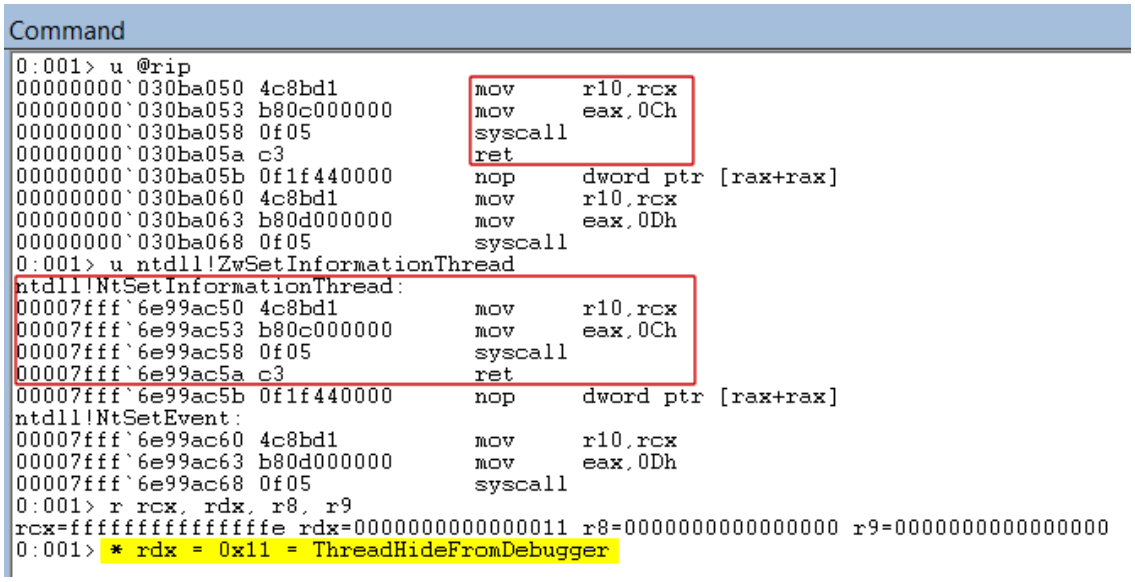


*Figure 7. FinFisher loader calling native Windows API to perform anti-debugging tricks*

At this point, the fun in analysis is not over. A lot of additional anti-sandbox checks are performed in this exact order:

1. Check that the malware is not executed under the root folder of a drive
2. Check that the malware file is readable from an external source
3. Check that the hash of base path is not 3D6D62AF1A7C8053DBC8E110A530C679
4. Check that the full malware path contains only human readable characters ("a-z", "A-Z", and "0-9")
5. Check that no node in the full path contains the MD5 string of the malware file
6. Fingerprint the system and check the following registry values:
    1. *HKLM\SOFTWARE\Microsoft\Cryptography\MachineGuid* should not be "6ba1d002-21ed-4dbe-afb5-08cf8b81ca32"

2. *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\DigitalProductId* should not be "55274-649-6478953-23109", "A22-00001", or "47220"
3. *HARDWARE\Description\System\SystemBiosDate* should not contain "01/02/03"

7. Check that the mutex *WininetStartupMutex0* does not already exist
8. Check that no DLL whose base name has hash value of 0xC9CEF3E4 is mapped into the malware address space

The hashes in these checks are most likely correspond to sandbox or security products that the FinFisher authors want to avoid.

Next, the loader checks that it's not running in a virtualized environment (VMWare or Hyper-V) or under a debugger. For the hardware virtualization check, the loader obtains the hardware device list and checks if the MD5 of the vendor ID is equal to a predefined list. In our tests, the malware sample was able to easily detect both VMWare and Hyper-V environments through the detection of the virtualized peripherals (for example, Vmware has VEN_15AD as vendor ID, HyperV has VMBus as bus name). Office 365 ATP sandbox employs special mechanisms to avoid being detected by similar checks.

The loader's anti-debugger code is based on the following three methods:

1. The first call aims to destroy the debugger connection:

```
ZwSetInformationThread(GetCurrentProcess(), ThreadHideFromDebugger, NULL, 0);
```

NOTE: This call completely stops the execution of WinDbg and other debuggers

2. The second call tries to detect the presence of a debugger:

```
ZwQueryInformationProcess(CurProc, ProcessDebugPort, &debugPort, sizeof(ULONG_PTR))
ZwQueryInformationProcess(CurProc, ProcessDebugObjectHandle, &debugObjHandle, sizeof(ULONG_PTR))
```

3. The final call tries to destroy the possibility of adding software breakpoint:

```
VirtualProtectEx(CurProc, &DbgBreakPoint, 1, PAGE_RWX, &oldProtect)
DbgBreakPoint[0] = 0x90;                // Place a NOP opcode instead the standard INT 3
```

Finally, if the loader is happy with all the checks done so far, based on the victim operating system (32 or 64-bit) it proceeds to decrypt a set of fake bitmap resources (stage 2) embedded in the executable and prepares the execution of a new layer of VM decoding.

Each bitmap resource is extracted, stripped of the first 0x428 bytes (BMP headers and garbage data), and combined into one file. The block is decrypted using a customized algorithm that uses a key derived from the original malware dropper's *TimeDateStamp* field multiplied by 5.

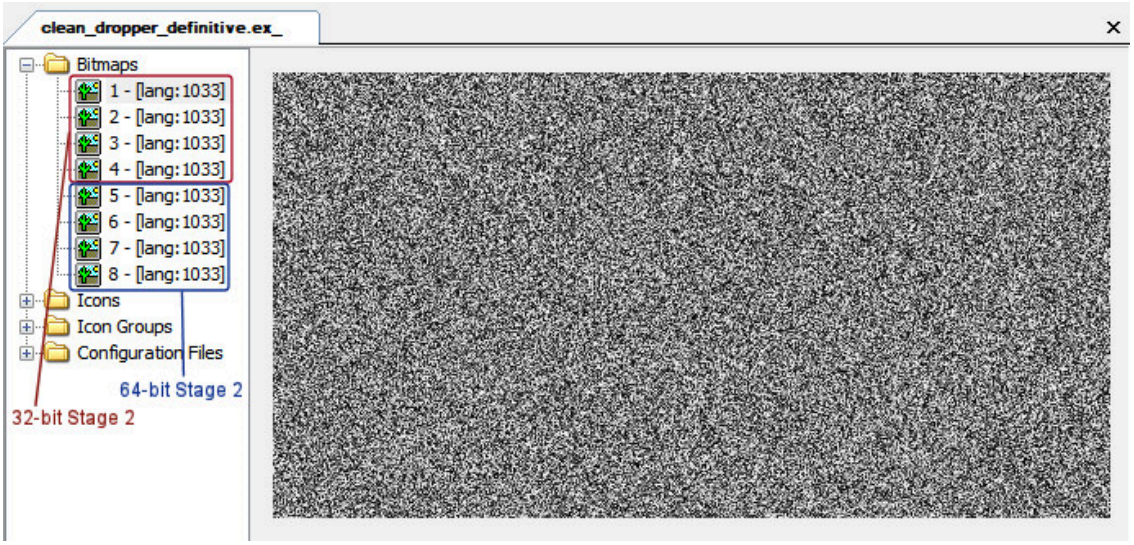


*Figure 8. The fake bitmap image embedded as resource*

The 32-bit stage 2 malware uses a customized loading mechanism (i.e., the PE file has a scrambled IAT and relocation table) and exports only one function. For the 64-bit stage 2 malware, the code execution is transferred from the loader using a well-known technique called Heaven's Gate. In the next sections, for simplicity, we will continue the analysis only on the 64-bit payload.

```
; int __cdecl __far SwitchTo64Bit(LPVOID lpNewPeBaseAddr)
SwitchTo64Bit proc far

lpNewPeBaseAddr= dword ptr  0Ch

  push    ebp
  mov     ebp, esp
  mov     eax, [ebp+8]  ; EAX = Extracted PE base address
  push    33h
  push    offset Virus_64BitRoutine
  retf                  ; Switch to 64 bit
```

*Figure 9. Heaven's gate is still in use in 2017*

## Stage 2: A second multi-platform virtual machine

The 64-bit stage 2 malware implements another loader combined with another virtual machine. The architecture is quite similar to the one described previously, but the opcodes are slightly different. After reversing these opcodes, we were able to update our interpreter script to support both 32-bit and 64-bit virtual machines used by FinFisher.

| INDEX | MNEMONIC | DESCRIPTION |
|---|---|---|
| 0x0 | JMP | Special obfuscated conditional Jump (always taken or always ignored) |
| 0x1 | JMP | Jump to a function (same as opcode 0x10) |
| 0x2 | CALL | Call to the function pointed by the internal VM value |

| | | |
|---|---|---|
| 0x3 | CALL | Optimized CALL function (like the 0x1E opcode of the 32-bit VM) |
| 0x4 | EXEC | Execute code and move to the next packet |
| 0x5 | JMP | Jump to an internal function |
| 0x6 | NOP | No operation, move to the next packet |
| 0x7 | CALL | Call an imported API (whose address is stored in the internal VM value) |
| 0x8 | LOAD | Load a value into the VM descriptor structure * |
| 0x9 | STORE | Store the internal VM value inside a register |
| 0xA | WRITE | Resolve a pointer and store the value of a register in its content |
| 0xB | READ | Move the value pointed by the VM internal value into a register |
| 0xC | LOAD | Load a value into the VM descriptor structure (not optimized) |
| 0xD | CMP | Compare the value pointed by the internal VM descriptor with a register |
| 0xE | CMP | Compare the value pointed by the internal VM descriptor with an immediate value |
| 0xF | XCHG | Exchange the value pointed by the internal VM descriptor with a register |
| 0x10 | SHL | Jump to a function (same as opcode 0x1) |

This additional virtual machine performs the same duties as the one already described but in a 64-bit environment. It extracts and decrypts the stage 3 malware, which is stored in encrypted resources such as fake dialog boxes. The extraction method is the same, but the encryption algorithm (also XOR) is much simpler. The new payload is decrypted, remapped, and executed in memory, and represents the installation and persistence stage of the malware.

## Stage 3: Installer that takes DLL side-loading to a new level

Stage 3 represents the setup program for FinFisher. It is the first plain stage that does not employ a VM or obfuscation. The code supports two different installation methods: setup in a UAC-enforced environment (with limited privileges), or an installation with full-administrative privileges enabled (in cases where the malware gains the ability to run with elevated permissions). We were a bit disappointed that we did not see traces of a true privilege escalation exploit after all this deobfuscation work, but it seems these FinFisher samples were designed to work just using UAC bypasses.

The setup code receives an installation command from the previous stage. In our test, this command was the value *3*. The malware creates a global event named *0x0A7F1FFAB12BB2* and drops some files under a folder located in *C:\ProgramData* or in the user application data folder. The name of the folder and the malware configuration are read from a customized configuration file stored in the resource section of the setup program.

Here the list of the files potentially dropped during the installation stage:

| FILE NAME | STAGE | DESCRIPTION |
|---|---|---|
| d3d9.dll | Stage 4 | Malware loader used for UAC environments with limited privileges; also protected by VM obfuscation |
| aepic.dll, sspisrv.dll, userenv.dll | Stage 4 | Malware loader used in presence of administrative privileges; executed from (and injected into) a fake service; also protected by VM obfuscation |
| msvcr90.dll | Stage 5 | Malware payload injected into the *explorer.exe* or *winlogon.exe* process; also protected by VM obfuscation |
| <randomName>.cab | Config | Main configuration file; encrypted |
| setup.cab | Unknown | Last section of the setup executable; content still unknown |
| <randomName>.7z | Plugin | Malware plugin used to spy the victim network communications |
| wsecedit.rar | Stage 6 | Main malware executable |

After writing some of these files, the malware decides which kind of installation to perform based on the current privilege provided by the hosting process (for example, if a Microsoft Office process was used as exploit vector):

1. **Installation process under UAC**

When running under a limited UAC account, the installer extracts d3d9.dll and creates a persistence key under *HKCU\Software\Microsoft\Windows\Run*. The malware sets a registry value (whose name is read from the configuration file) to *"C:\Windows\system32\rundll32.exe c:\ProgramData\AuditApp\d3d9.dll, Control_Run"*. Before doing this, the malware makes a screenshot of the screen and displays it on top of all other windows for few seconds. This indicates that the authors are trying to hide some messages showed by the system during the setup process.

When loaded with startup command *2*, the installer can copy the original *explorer.exe* file inside its current running directory and rename *d3d9.dll* to *uxtheme.dll*. In this case the persistence is achieved by loading the original explorer.exe from its startup location and, using DLL side-loading, passing the execution control to the stage 4 malware (discussed in next section).

Finally, the malware spawns a thread that has the goal to load, remap, and relocate the stage 5 malware. In this context, there is indeed no need to execute the stage 4 malware. The *msvcr90.dll* file is opened, read, and decrypted, and the code execution control is transferred to the *RunDll* exported routine.

In the case of 32-bit systems, the malware may attempt a known UAC bypass by launching *printui.exe* system process and using token manipulation with *NtFilterToken* as described in this blog post.

2. **Installation process with administrative privilege**

This installation method is more interesting because it reveals how the malware tries to achieve stealthier persistence on the machine. The method is a well-known trick used by penetration testers that was automated and generalized by FinFisher

The procedure starts by enumerating the *KnownDlls* object directory and then scanning for section objects of the cached system DLLs. Next, the malware enumerates all .exe programs in the *%System%* folder and looks for an original signed Windows binary that imports from at least one *KnownDll* and from a library that is not in the *KnownDll* directory. When a suitable .exe file candidate is found, it is copied into the malware installation folder (for example, *C:\ProgramData*). At this point the malware extracts and decrypts a stub DLL from its own resources (ID 101). It then calls a routine that adds a code section to a target module. This section will contain a fake export table mimicking the same export table of the original system DLL chosen. At the time of writing, the dropper supports *aepic.dll*, *sspisrv.dll*, *ftllib.dll*, and *userenv.dll* to host the malicious FinFisher payload. Finally, a new Windows service is created with the service path pointing to the candidate .exe located in this new directory together with the freshly created, benign-looking DLL.

In this way, when the service runs during boot, the original Windows executable is executed from a different location and it will automatically load and map the malicious DLL inside its address space, instead of using the genuine system library. This routine is a form of generic and variable generator of DLL side-loading combinations.

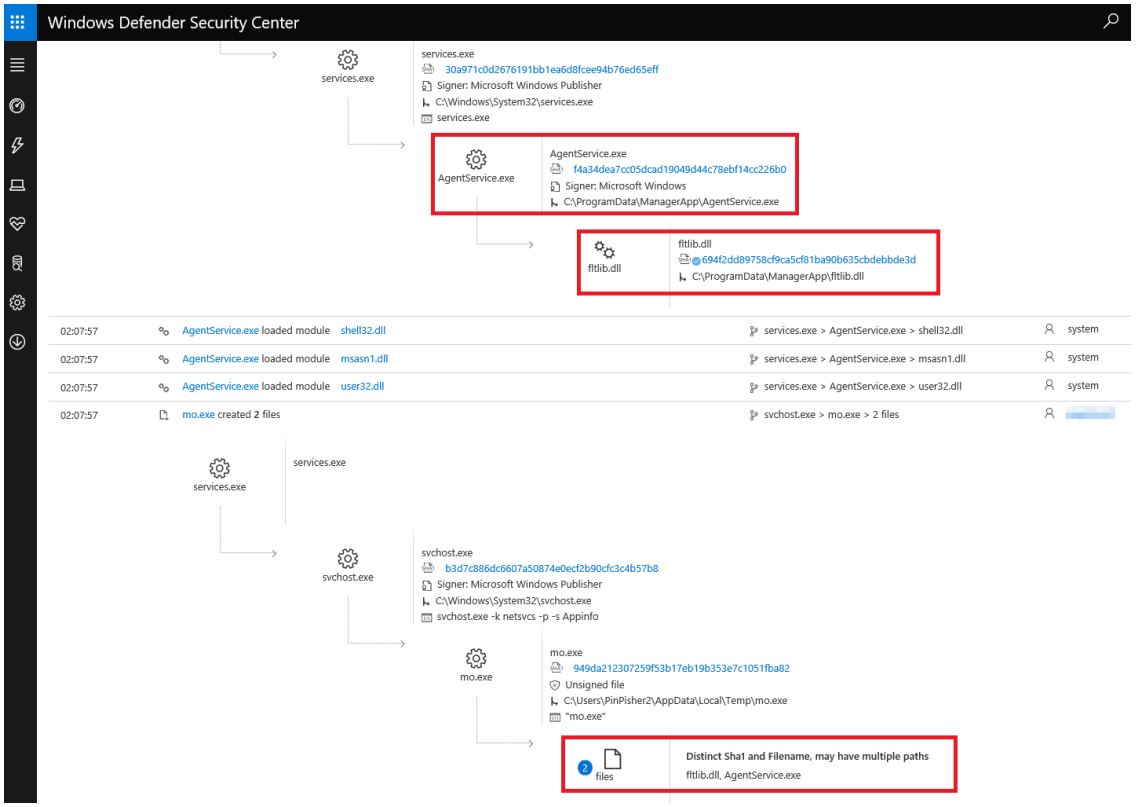


*Figure 10. Windows Defender ATP timeline can pinpoint the service DLL side-loading trick (in this example, using fltlib.dll).*

In the past, we have seen other activity groups like LEAD employ a similar attacker technique named “proxy-library” to achieve persistence, but not with this professionalism. The said technique brings the advantage of avoiding auto-start extensibility points (ASEP) scanners and programs that checks for binaries installed as service (for the latter, the service chosen by FinFisher will show up as a clean Windows signed binary).

The malware cleans the system event logs using *OpenEventLog/ClearEventLog* APIs, and then terminates the setup procedure with a call to *StartService* to run the stage 4 malware.

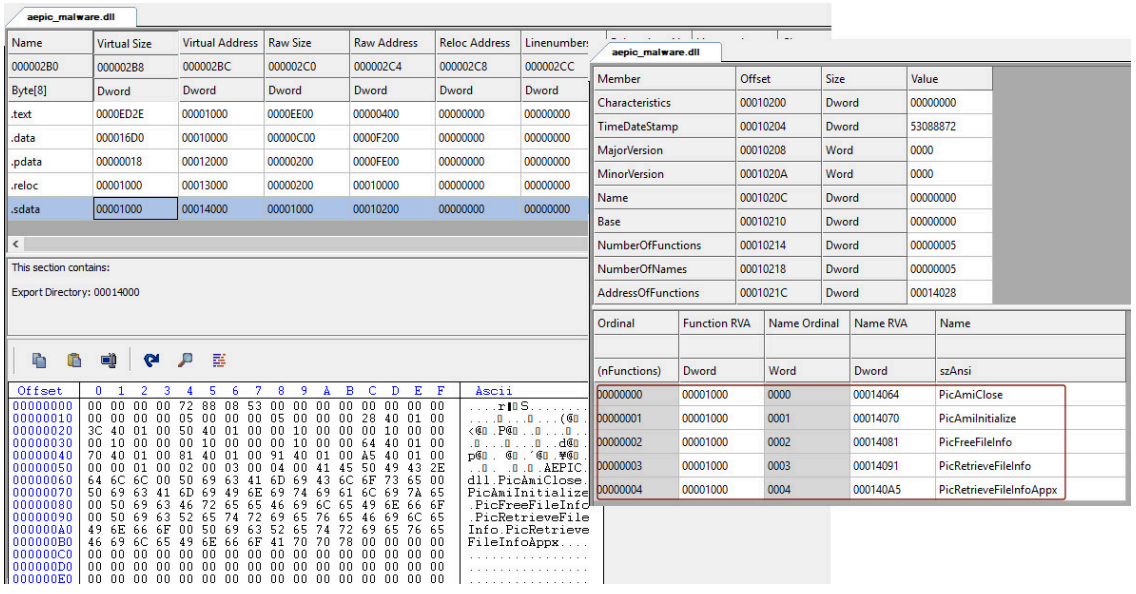

*Figure 11. The DLL side-loaded stage 4 malware mimicking a real export table to avoid detection*

## Stage 4: The memory loader – Fun injection with GDI function hijacking

Depending on how stage 4 was launched, two different things may happen:

- In the low-integrity case (under UAC) the installer simply injects the stage 5 malware into the bogus *explorer.exe* process started earlier and terminates
- In the high-integrity case (with administrative privileges or after UAC bypass), the code searches for the process hosting the Plug and Play service (usually *svchost.exe*) loaded in memory and injects itself into it

For the second scenario, the injection process works like this:

1. The malware opens the target service process.
2. It allocates and fills four chunks of memory inside the service process. One chunk contains the entire malware DLL code (without PE headers). Another chunk is used to copy a basic *Ntdll* and *Kernel32* import address table. Two chunks are filled with an asynchronous procedure call (APC) routine code and a stub.
3. It opens the service thread of the service process and uses the *ZwQueueApcThread* native API to inject an APC.

The APC routine creates a thread in the context of the svchost.exe process that will map and execute the stage 5 malware into the *winlogon.exe* process.

The injection method used for *winlogon.exe* is also interesting and quite unusual. We believe that this method is engineered to avoid trivial detection of process injection using the well-detected *CreateRemoteThread* or *ZwQueueApcThread API*.

The malware takes these steps:

1. Check if the system master boot record (MBR) contains an infection marker (*0xD289C989C089* 8-bytes value at offset *0x2C*), and, if so, terminate itself
2. Check again if the process is attached to a debugger (using the techniques described previously)
3. Read, decrypt, and map the stage 5 malware (written in the previous stage in *msvcr90.dll*)
4. Open *winlogon.exe* process
5. Load *user32.dll* system library and read the *KernelCallbackTable* pointer from its own process environment block (PEB) (Note: The *KernelCallbackTable* points to an array of graphic functions used by Win32 kernel subsystem module *win32k.sys* as call-back into user-mode.)
6. Calculate the difference between this pointer and the User32 base address.
7. Copy the stage 5 DLL into winlogon.exe
8. Allocate a chunk of memory in winlogon.exe process and copy the same APC routine seen previously
9. Read and save the original pointer of the *__fnDWORD* internal User32 routine (located at offset *+0x10* of the *KernelCallbackTable*) and replace this pointer with the address of the APC stub routine

After this function pointer hijacking, when *winlogon.exe* makes any graphical call (GDI), the malicious code can execute without using *CreateRemoteThread* or similar triggers that are easily detectable. After execution it takes care of restoring the original *KernelCallbackTable*.

## Stage 5: The final loader takes control

The stage 5 malware is needed only to provide one more layer of obfuscation, through the VM, of the final malware payload and to set up a special Structured Exception Hander routine, which is inserted as *Wow64PrepareForException* in *Ntdll*. This special exception handler is needed to manage some memory buffers protection and special exceptions that are used to provide more stealthy execution.

After the VM code has checked again the user environment, it proceeds to extract and execute the final un-obfuscated payload sample directly into winlogon.exe (alternatively, into *explorer.exe*) process. After the payload is extracted, decrypted, and mapped in the process memory, the malware calls the new DLL entry point, and then the *RunDll* exported function. The latter implements the entire spyware program.

## Stage 6: The payload is a modular spyware framework for further analysis

Our journey to deobfuscating FinFisher has allowed us to uncover the complex anti-analysis techniques used by this malware, as well as to use this intel to protect our customers, which is our top priority. Analysis of the additional spyware modules is future work.

It is evident that the ultimate goal of this program is to steal information. The malware architecture is modular, which means that it can execute plugins. The plugins are stored in its resource section and can be protected by the same VM. The sample we analyzed in October, for example, contains a plugin that is able to spy on internet connections, and can even divert some SSL connections and steal data from encrypted traffic.

Some FinFisher variants incorporate an MBR rootkit, the exact purpose of which is not clear. Quite possibly, this routine targets older platforms like Windows 7 and machines not taking advantage of hardware protections like UEFI and SecureBoot, available on Windows 10. Describing this additional piece of code in detail is outside the scope of this analysis and may require a new dedicated blog post.

## Defense against FinFisher

Exposing as much of FinFisher's riddles as possible during this painstaking analysis has allowed us to ensure our customers are protected against this advanced piece of malware.

Windows 10 S devices are naturally protected against FinFisher and other threats thanks to the strong code integrity policies that don't allow unknown unsigned binaries to run (thus stopping FinFisher's PE installer) or loaded (blocking FinFisher's DLL persistence). On Windows 10, similar code integrity policies can be configured using Windows Defender Application Control.

Office 365 Advanced Threat Protection secures mailboxes from email campaigns that use zero-day exploits to deliver threats like FinFisher. Office 365 ATP blocks unsafe attachments, malicious links, and linked-to files using time-of-click protection. Using intel from this research, we have made Office 365 ATP more resistant to FinFisher's anti-sandbox checks.

Generic detections, advanced behavioral analytics, and machine learning technologies in Windows Defender Advanced Threat Protection detect FinFisher's malicious behavior throughout the attack kill chain and alert

SecOps personnel. Windows Defender ATP also integrates with the Windows protection stack so that protections from Windows Defender AV and Windows Defender Exploit Guard are reported in Windows Defender ATP portal, enabling SecOps personnel to centrally manage security, and as well as promptly investigate and respond to hostile activity in the network.

We hope that this writeup of our journey through all the multiple layers of protection, obfuscation, and anti-analysis techniques of FinFisher will be useful to other researchers studying this malware. We believe that an industry-wide collaboration and information-sharing is important in defending customers against this complex piece of malware. For further reading, we recommend these other great references:

- Devirtualizing FinSpy [PDF], Tora (2012)
- Finfisher rootkit analysis, Artem Baranov (2017)
- A Walk-Through Tutorial, with Code, on Statically Unpacking the FinSpy VM: Part One, x86 Deobfuscation, Rolf Rolles (2018)
- FinSpy VM Part 2: VM Analysis and Bytecode Disassembly, Rolf Rolles (2018)
- ESET's guide to deobfuscating and devirtualizing FinFisher [PDF], Filip Kafka (2018)

To test how Windows Defender ATP can help your organization detect, investigate, and respond to advanced attacks, **sign up for a free trial**.

***Andrea Allievi**, Office 365 ATP Research team*
*with **Elia Florio**, Windows Defender ATP Research team*

Sample analyzed:

MD5: a7b990d5f57b244dd17e9a937a41e7f5
SHA-1: c217d48c4ac1555491348721cc7cfd1143fe0b16
SHA-256: b035ca2d174e5e4fd2d66fd3c8ce4ae5c1e75cf3290af872d1adb2658852afb8

---

## Talk to us

Questions, concerns, or insights on this story? Join discussions at the Microsoft community and Windows Defender Security Intelligence.

Follow us on Twitter @WDSecurity and Facebook Windows Defender Security Intelligence.

---