Devirtualizing FinSpy

by Tora
Outline

- Background info
- Main binary
- The first drop
  - Virtualization analysis
  - De-virtualization
- Further analysis
  - Collection of anti-tricks
  - The big picture
  - Crypto, MBR...
- Lessons learned
Background

- "From Bahrain with Love" post at citizenlab
  - Emails from a fake Aljazeera reporter account sent to Bahrain "activists".
  - Using the RTL trick to pretend to be .jpg's
  - citizenlab analyzed the malware and announced it as a component of FinFisher from Gamma Intl.
  - The post provides hashes for all the samples analyzed. Let's take a look at
    49000fc53412bfda157417e2335410cf69ac26b66b0818a3be7eff589669d040
Main sample

- Looks like an apparently harmless Windows application (WndProc does nothing)

```assembly
    call    ReplaceWindowFunctions
    mov     eax, esi
    call    RegisterWindowClass
    push    0       ; lpParam
    push    esi     ; hWnd
    push    0       ; hMenu
    push    0       ; hWndParent
    push    0       ; nHeight
    push    80000000h ; nWidth
    push    0       ; Y
    push    80000000h ; X
    push    0CF0000h ; dwStyle
    push    offset Buffer ; lpWindowName
    push    offset class_name ; lpClassName
    push    0       ; dwExStyle
    mov     hInstance, esi
    call    ds:CreateWindowExW ; call to replaced FakeCreateWindowExW
    mov     edi, eax
    test    edi, edi
    jz      loc_4023E4
    mov     eax, [esp+24h+nShowCmd]
    push    eax     ; nCmdShow
    push    edi     ; hWnd
    call    ds:ShowWindow
```
The first drop

- Entry point looks normal, but then...

```
winmain:                ; CODE XREF
    mov    edi, edi
    push   ebp
    mov    ebp, esp
    sub    esp, 25Ch
    mov    eax, __security_cookie
    xor    eax, ebp
    mov    [ebp-4], eax
    push   ebx
    push   esi
    push   edi
    push   0F6DB9A6Ah
    jmp    loc_4049B1

;---------------------------------------------------------------
    align 10h
    dd    7Bh dup(0)
    db    5 dup(0CCh)

;---------------------------------------------------------------
    mov    edi, edi
    push   ebp
    mov    ebp, esp
    push   0F6DB9D41h
    jmp    loc_4049B1

;---------------------------------------------------------------
    dd    9 dup(0)
    dd    00000000h, 00000000h

;---------------------------------------------------------------
    mov    edi, edi
    push   ebp
    mov    ebp, esp
    push   ecx
    and    dword ptr [ebp-4], 0
    push   0F6DB9D73h
    jmp    loc_4049B1
```
The first drop

- Very simple obfuscation
Virtualization analysis

Basic flow of main loop

1. Disable NX if possible
2. Allocate an array of "VM context" handles
3. Allocate a context for current thread (CTX)
4. Unpack VM
5. Search for entry point
6. Prepare VM OP instruction
7. Decrypt VM code
8. Execute virtual OP
9. Goto 6
Virtualization analysis

- VM setup

- Offset to VM instruction code
- Max valid address inside context
- Temp register
- Return address
- Return via epilogue
- Obfuscation relative offset
- Process imagebase

- Copy of stack pointer
- Search VirtualEIP function
- Current instruction VEIP
- Opcode
- Relocation information
- Raw bytes
- First free address
Virtualization analysis

- Opcodes: 11 opcodes used. Two types
  - Native: "Raw bytes" are used to construct x86 native code and executed.
  - VM-level: just modifications on the CTX structure, basically operations with the temp register
Virtualization analysis

- Native-execution opcodes
Virtualization analysis

Opcodes 0x01 and 0x04: Execute native code

\[
\begin{align*}
\text{start+0: } & \text{ POPFD} \\
\text{start+1: } & \text{ POPAD} \\
\text{start+2: } & \text{ <native code>} \\
& \ldots \\
\text{ret\_code+0: } & \text{ PUSH } <\text{VM\_loop}> \\
\text{ret\_code+5: } & \text{ RETN} \\
\text{VM\_loop+0: } & \text{ PUSHA} \\
\text{VM\_loop+1: } & \text{ PUSHF}
\end{align*}
\]
Virtualization analysis

Opcode 0x06: Native register to temp register

\[
\begin{align*}
idx &= 7 - CTX[0x34] \\
saved_{esp} &= CTX[0x20] \\
CTX[0x08] &= saved_{esp}[idx*4+4] \\
CTX[0x00] &=CTX[0x00] += 0x18 \\
EAX &= VM\_loop \\
ESP &= CTX[0x20] \\
JMP CTX[0x10] &: Ret-to-EAX epilogue
\end{align*}
\]
Virtualization analysis

Opcode 0x06: Native register to temp register

<table>
<thead>
<tr>
<th>PUSHFD + PUSHAD</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS</td>
<td>(+4)</td>
</tr>
<tr>
<td>POP EDI</td>
<td>(CTX[0x34] = 7)</td>
</tr>
<tr>
<td>POP ESI</td>
<td>(CTX[0x34] = 6)</td>
</tr>
<tr>
<td>POP ESP</td>
<td>(CTX[0x34] = 5)</td>
</tr>
<tr>
<td>POP EBP</td>
<td>(CTX[0x34] = 4)</td>
</tr>
<tr>
<td>POP EBX</td>
<td>(CTX[0x34] = 3)</td>
</tr>
<tr>
<td>POP EDX</td>
<td>(CTX[0x34] = 2)</td>
</tr>
<tr>
<td>POP ECX</td>
<td>(CTX[0x34] = 1)</td>
</tr>
<tr>
<td>POP EAX</td>
<td>(CTX[0x34] = 0)</td>
</tr>
</tbody>
</table>
Virtualization analysis

CTX[0x10] epilogue

\[ [ESP-4] = EAX \]

POPF

POP EDI
POP ESI
POP EBP
POP EBX
POP EDX
POP ECX
POP EAX

JMP [ESP-0x28]; Initial EAX value

POPAD does

POP EDI
POP ESI
POP ESP
POP EBP
POP EBX
POP EDX
POP ECX
POP EAX

POPAD
MOV EBP, ESP
Virtualization analysis

Opcode 0x07: Push temp register

\[
\begin{align*}
\text{saved} \_ \text{esp} &= \text{CTX}[0x20] \\
\text{memmove} (\text{saved} \_ \text{esp}-4, \text{saved} \_ \text{esp}, 0x24) \\
\text{CTX}[0x20] &= 4 \\
\text{temp} \_ \text{register} &= \text{CTX}[0x08] \\
\text{saved} \_ \text{esp} &= \text{CTX}[0x20] \\
\text{saved} \_ \text{esp}[0x24] &= \text{temp} \_ \text{register} \\
\text{CTX}[0x00] &= 18 \\
\text{EAX} &= \text{VM} \_ \text{loop} \\
\text{ESP} &= \text{CTX}[0x20] \\
\text{JMP} \text{CTX}[0x10] &= \text{Ret-to-EAX epilogue}
\end{align*}
\]
Virtualization analysis

Opcode 0x07: Push temp register
Virtualization analysis

Opcode 0x03: Call native/imports

start+0: POPFD; POPAD
start+2: PUSH <api_ret>
start+7: <native_jmp>
api_ret+0: PUSH <VirtualEIP>
api_ret+5: PUSHAD; PUSHFD
api_ret+7: PUSH <&CTX>
api_ret+8: POP EBX
api_ret+9: PUSH <call_epilogue>
api_ret+A: RETN
Virtualization analysis

call epilogue

  EAX = VirtualEIP
  offset = VEIPToOffset(EAX, VM_code)
  memmove(ESP+4, ESP, 0x24)
  ESP += 4
  CTX[0x00] = offset
  CTX[0x44] -= 0x30
  EAX = VM_loop
  JMP CTX[0x10] ; Ret-to-EAX epilogue
Virtualization analysis

Opcode 0x05: Move raw value to temp register

\[
\begin{align*}
\text{CTX}[0x08] &= \text{CTX}[0x34] \\
\text{CTX}[0x00] &= \text{CTX}[0x00] + 0x18 \\
\text{EAX} &= \text{VM\_loop} \\
\text{ESP} &= \text{CTX}[0x20] \\
\text{JMP CTX}[0x10] &; \text{Ret-to-EAX epilogue}
\end{align*}
\]
Virtualization analysis

Opcodes 0x08: Dereference temp register

\[
\text{CTX}[0x08] = \text{DWORD PTR}[\text{CTX}[0x08]]
\]

\[
\text{CTX}[0x00] += 0x18
\]

EAX = VM_loop

ESP = CTX[0x20]

\[
\text{JMP CTX}[0x10] \; \text{; Ret-to-EAX epilogue}
\]
Virtualization analysis

Opcode 0x09: Temp register to native register

\[
\begin{align*}
\text{index} &= 7 - \text{BYTE PTR} \cdot \text{CTX}[0x34] \\
\text{saved}_\text{esp} &= \text{CTX}[0x20] \\
\text{temp} &= \text{CTX}[0x08] \\
\text{saved}_\text{esp}[\text{index} \cdot 4 + 4] &= \text{temp} \\
\text{CTX}[0x00] &= 0x18 \\
\text{EAX} &= \text{VM\_loop} \\
\text{ESP} &= \text{CTX}[0x20] \\
\text{JMP CTX}[0x10] \; ; \; \text{Ret-to-EAX epilogue}
\end{align*}
\]
Virtualization analysis

Opcode 0x0A: Temp register to address

address = CTX[0x34]
[address] = CTX[0x08]
CTX[0x00] += 0x18
EAX = VM_loop
ESP = CTX[0x20]
JMP CTX[0x10] ; Ret-to-EAX epilogue
Virtualization analysis

Opcode 0x02: Call native (direct)

```plaintext
start+0: POPFD+POPAD
start+2: PUSH <native_ret>
start+7: PUSH <target>
start+C: RETN
native_ret+0: PUSH <VirtualEIP>
native_ret+5: PUSHAD; PUSHFD
native_ret+7: PUSH <&CTX>; POP EBX
native_ret+D: PUSH <CTX[0x10]>; RETN
```
Virtualization analysis

Opcode 0x00: Conditional jump

from VM code: POPFD
start: **xx**02 -> 7402 -> JZ start+4
start+2: F8 -> CLC
start+3: B0 F9 -> MOV AL, 0xF9
start+4: F9 -> STC
start+5: MOV EAX, <condition_check>
start+A: JMP EAX
Virtualization analysis

Opcode 0x00: Conditional jump

JB <jump_taken>
CTX[0x00] += 0x18
[...]
JMP CTX[0x10] ; Ret-to-EAX epilogue

jump_taken: if CTX[0x35] == 0:
    VEIPtoOffset()
    CTX[0x00] += 0x18 [...]
else
    EAX = imagebase + CTX[0x39]
    JMP CTX[0x10] ; Ret-to-EAX epilogue
Virtualization analysis

How disasembly actually looks like:

0xf6db9a6a  0406000008B3D3C10  mov edi,[0x40103c]  KERNEL32.dll_GetModuleHandleW
0xf6db9a70  0402000033F60000  xor esi,esi
0xf6db9a74  0600000006000000  mov temp, esi
0xc27f370e  0700000000000000  push temp   (esi)
0xf6db9a75  030200007B9ADBF6  jmp edi ; jmp TAG:0xf6db9a7b
[...]
0xf6db9a86  0600000000000000  mov temp, eax
0xdd2ca350  0A000000807F4000  mov [0x407f80], temp   (eax)
0xf6db9a8f  0406000008D85C0FD  lea eax,[ebp-0x240]
0xf6db9a97  0600000006000000  mov temp, esi
0x27227d8a  0700000000000000  push temp   (esi)
0xf6db9aa0  0600000000000000  mov temp, eax
0x5d32a971  0700000000000000  push temp   (eax)
0xf6db9aa1  02000000A99ADBF6  call 0x405cf0; jmp TAG:0xf6db9aa9
0xf6db9aa9  0500000008020000  mov temp, 0x208
De-virtualization

- Scan code for jump to the VM (PUSH <VirtualEIP> + JMP VM_start)
- Calculate padding to next function (optional)
- Unpack and decrypt VM code
- Search for each VirtualEIP
- Translate VM into x86 code (easy!)
- Overwrite padding with generated x86 code
  - Stop when VirtualEIP is referenced by another VM jump, as that's the entry point of another function.
  - Yes, we're lucky that instructions are sequential ;)


Analysis of de-virtualized code

- De-virtualized code contains several anti-* tricks
- All of them are known, so not so much fun
- Lots of blacklisted id's (who were they trying to avoid?)
Analysis of de-virtualized code

- Blacklisted values
  - SOFTWARE\Microsoft\Cryptography\MachineGuid $\neq$ 6ba1d002-21ed-4dbe-afb5-08cf8b81ca32
  - SOFTWARE\Microsoft\Windows NT\CurrentVersion\DigitalProductId $\neq$ 55274-649-6478953-23109, A22-00001, 47220
  - HARDWARE\Description\System\SystemBiosDate $\neq$ 01/02/03
  - GetVersion() $\neq$ 5 (major version)
  - CS (code segment) $\equiv$ 0x1b || 0x23 (user-mode check?)
  - Hashes module path (and all its substrings) and checks that hash $\neq$ 0xA51198F4
Analysis of de-virtualized code

- Anti-debug:
  - Checks PEB for the BeingDebugged flag
  - Replace `DbgBreakPoint` function (is a single int3) with a NOP.
  - `ZwSetInformationThread` with `ThreadInformationClass == 0x11` (detach debugger)
  - `CloseHandle()` with invalid handle
  - `ZwQueryInformationProcess` with `ThreadInformationClass == 0x7` `ProcessDebugPort` and `0x1E` `ProcessDebugObjectHandle`
  - `ZwSetInformationThread` enabling `ThreadHideFromDebugger`
Analysis of de-virtualized code

- Misc anti-*:
  - Manual load of DLLs. Open, read, apply relocs and then parse export directory to resolve APIs by hash.
  - Opens JobObjects with names like `Local\COMODO_SANDBOX_0x%X_R%d` (%X is PID and %d is in range [1-6]).
    - If it succeeds, call BasicUIRestrictions and ExtendLimitInformation (seems limiting memory usage to a really low limit)
  - (Continues...)
Analysis of de-virtualized code

- Misc anti-*:
  - Check running process and modules looking for:
    - cmdguard.sys and cfp.exe for Comodo
    - klif.sys and avp.exe for Kaspersky
    - bdspy.sys and bullguard.exe for BullGuard
    - ccsvchst.exe for Symantec
    - fsm32.exe and fsma32.exe for F-Secure
    - rfwtdi.sys and rsfwdrv.sys for Beijing Rising
  - No AVKills, but depending on present AV the sample uses different drop/inject methods
  - For Kaspersky, it even opens the avp.exe file and checks for the version inside (ver 0xB000)
Analysis of de-virtualized code

- Misc anti-*:
  - DLLs with invalid IATs

![Import name hashes](image)
Analysis of de-virtualized code

- Enough anti-stuff, what is the payload??
- Actually it just drops more samples, depending on the environment.
  - Sample drops a 32-bit or 64-bit DLL depending on the OS
  - DLL is loaded/injected depending on what AV product is present
- So, all this boring stuff just to get a couple of dropped files? Now what?
  - Now we have a de-virtualizer, we can automate and get rid of all this much faster...
Two blobs of data. First xor-encrypted contains C2 hostname and other strings. Second is RC4-encrypted (key in the RT_ICON resource) injected in iexplore.exe (communication module).

Plain executable
Virtualized code
kernel + virtualized
unknown

Plain executable
Virtualized code
kernel + virtualized
unknown
Analysis of de-virtualized code

- **Crypto**
  - XOR for most drops (key fixed or in some cases key is timestamp from PE header)
  - RC4 for critical data resources, keys are stored in a common config file.
  - In some cases, filename is the key.

- **MBR**
  - Probably worth another talk ;)
  - Is in charge to load the hiding driver during boot
  - MBR payload is constructed from a template, so component that installs it has to "fill the blanks" like disk geometry params and payloads.
  - Infection check: if MBR[0x2C:0x2D] == CD 18 (int18h), then you may have a problem
Lessons learned

• VM really well designed
  ○ Same VM works for x86-32 and 64bit
  ○ The conditional jump emulation was the key to avoid having to worry about EFLAGS emulation.

• Complex malware == modular project.
  ○ However modular means you can face older/buggy versions of components you already analyzed (ex: APLib).

• Removing virtualization is sometimes possible (cost < benefit)
  ○ In this case, benefit was obvious because of the number of virtualized modules using the same VM