# 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 Bahrein "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)

```
call    ReplaceWindowFunctions
mov     eax, esi
call    RegisterWindowClass
push    0               ; lpParam
push    esi             ; hInstance
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 XRE
                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 0CC000000h, 0CCCCCCCCh
; ---------------------------------------------------------
                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    ecx
                and     dword ptr [ebp-4], 0
                push    0F6DB9D73h
                jmp     loc_4049B1
```

# The first drop

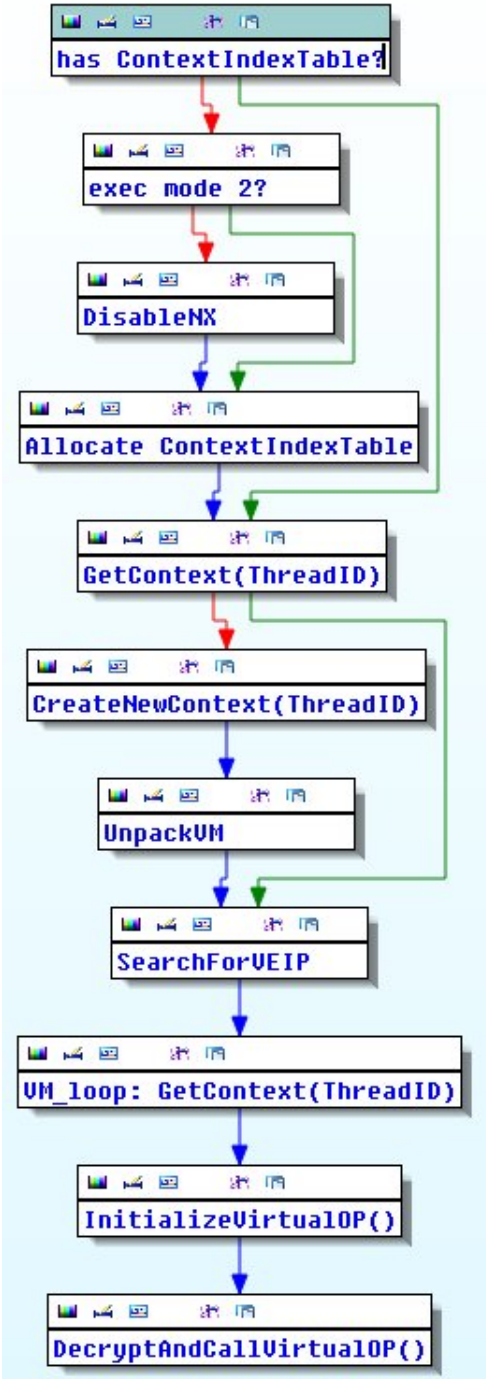
- Very simple obfuscation

```
.text:004049B3                  call    $+5
.text:004049B8                  pop     ebp
.text:004049B9                  lea     eax, [ebp+0Eh]
.text:004049BC                  push    eax
.text:004049BD                  lea     eax, [ebp+3BAh]
.text:004049BD ; ---------------------------------------------------------------------------
.text:004049C3                  db 0EBh ; d
.text:004049C4 ; ---------------------------------------------------------------------------
.text:004049C4                  jmp     eax
.text:004049C6 ; ---------------------------------------------------------------------------
.text:004049C6                  test    eax, eax
.text:004049C8                  jnz     loc_404A57
.text:004049CE                  cmp     dword ptr [ebp+470h], 2
.text:004049D5                  jz      short loc_404A30
.text:004049D7                  push    1000h
.text:004049DC                  lea     eax, [ebp+31h]
.text:004049DF                  push    eax
.text:004049E0                  lea     eax, [ebp+347h]
.text:004049E6
.text:004049E6 loc_4049E6:                              ; CODE XREF: .text:loc_4049E6↑j
.text:004049E6                  jmp     short near ptr loc_4049E6+1
.text:004049E6 ; ---------------------------------------------------------------------------
```

# Virtualization analysis

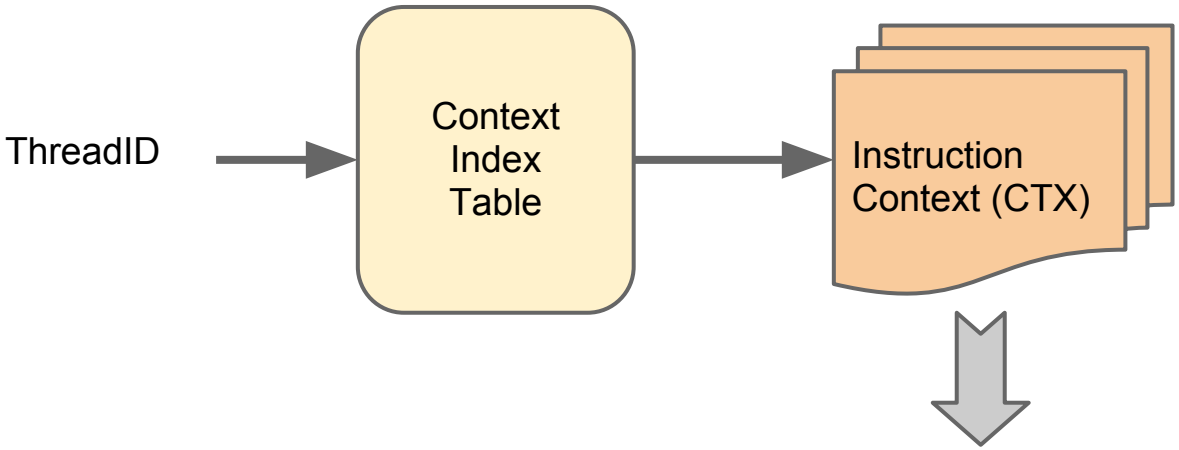
## Basic flow of main loop

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



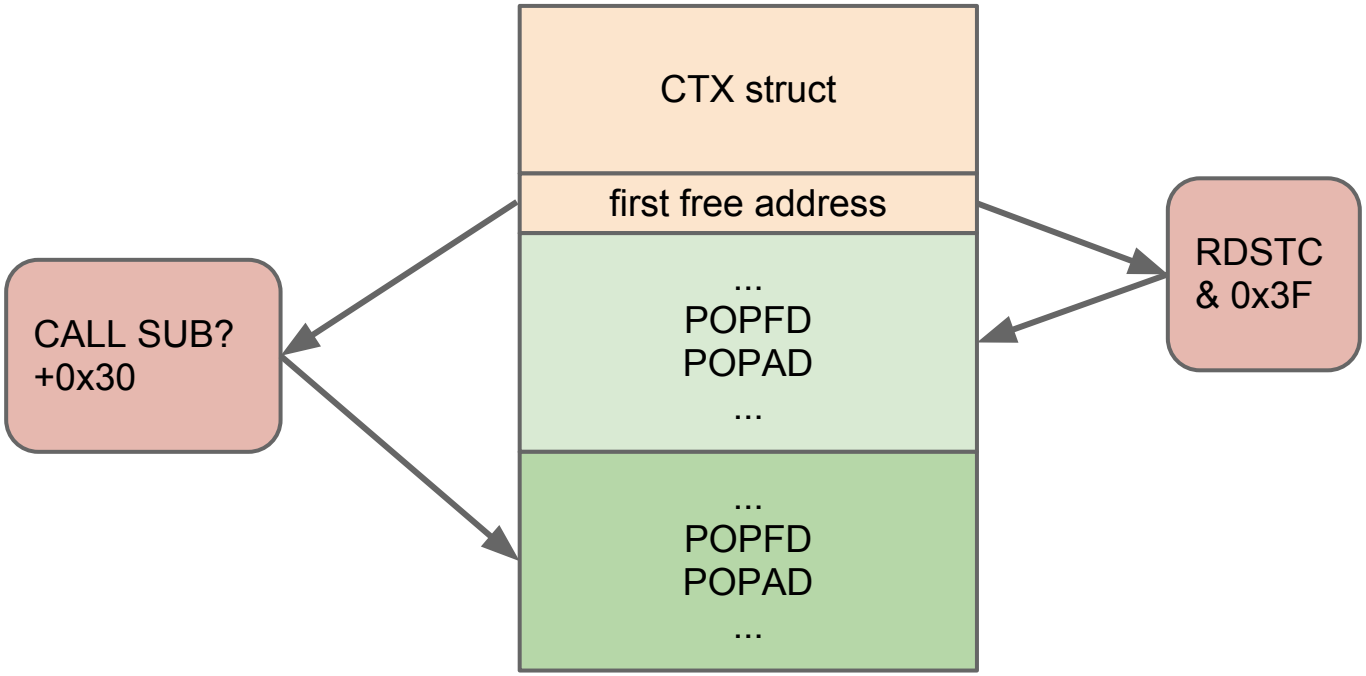

- Offset to VM instruction code
- Max valid address inside context
- Temp register
- Return address
- Return via epilogue
- Obfuscation relative offset
- Process imagebase
- Copy of stack pointer
- Search VirtualEIP function
- Current instruction VEIP
- Opcode
- Relocation information
- Raw bytes
- First 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

```
start+0: POPFD
start+1: POPAD
start+2: <native code>
...
ret_code+0: PUSH <VM_loop>
ret_code+5: RETN
VM_loop+0: PUSHA
VM_loop+1: PUSHF
```

# Virtualization analysis

Opcode 0x06: Native register to temp register

```
idx = 7 - CTX[0x34]
saved_esp = CTX[0x20]
CTX[0x08] = saved_esp[idx*4+4]
CTX[0x00] += 0x18
EAX = VM_loop
ESP = CTX[0x20]
JMP CTX[0x10] ; Ret-to-EAX epilogue
```

# Virtualization analysis

## Opcode 0x06: Native register to temp register

**PUSHFD + PUSHAD**

```
EFLAGS        (+4)
POP EDI       (CTX[0x34] = 7)
POP ESI       (CTX[0x34] = 6)
POP ESP       (CTX[0x34] = 5)
POP EBP       (CTX[0x34] = 4)
POP EBX       (CTX[0x34] = 3)
POP EDX       (CTX[0x34] = 2)
POP ECX       (CTX[0x34] = 1)
POP EAX       (CTX[0x34] = 0)
```

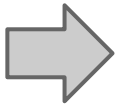
# Virtualization analysis

## CTX[0x10] epilogue

```
[ESP-4] = EAX
POPF
POP EDI
POP ESI
POP EBP
POP EBX
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
```

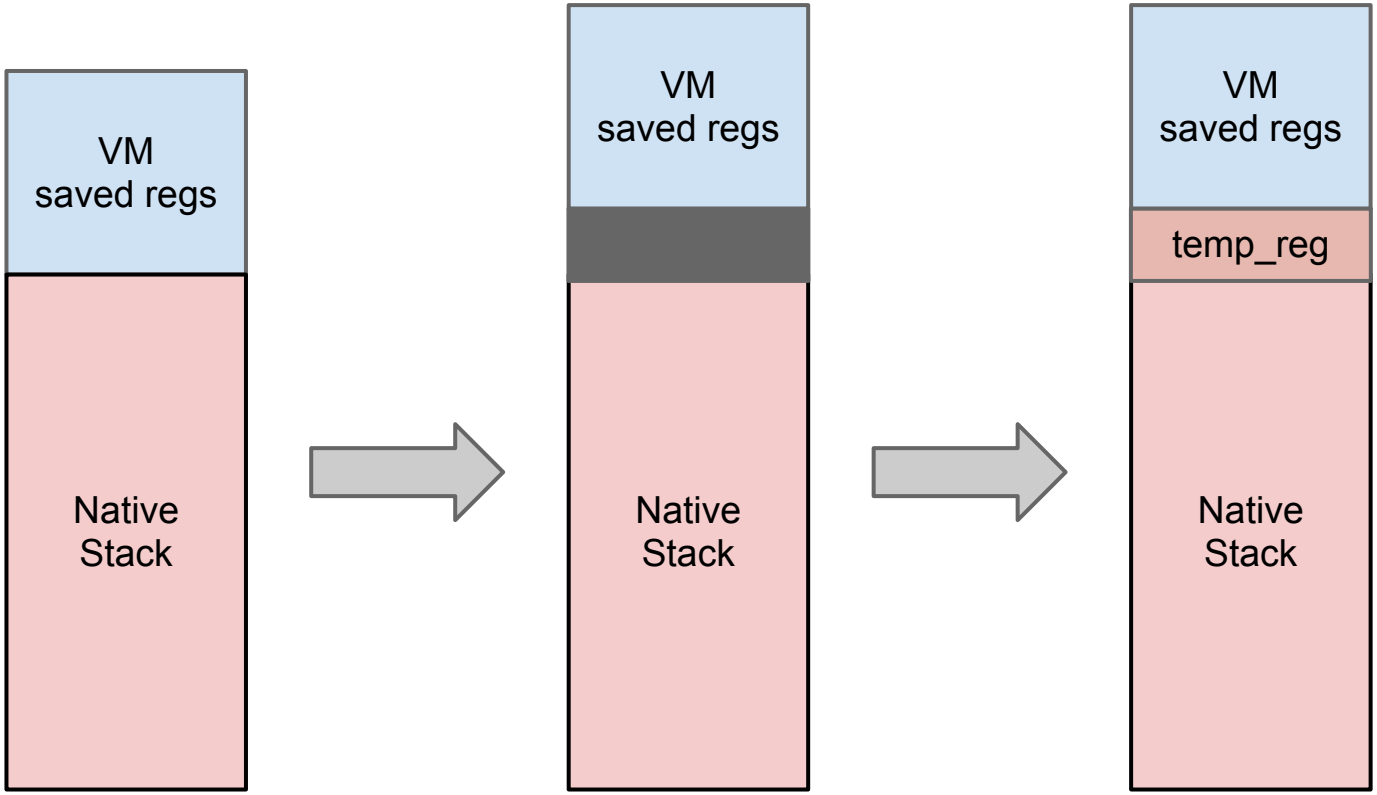
# Virtualization analysis

## Opcode 0x07: Push temp register

```
saved_esp = CTX[0x20]
memmove(saved_esp-4, saved_esp, 0x24)
CTX[0x20] -= 4
temp_register = CTX[0x08]
saved_esp = CTX[0x20]
saved_esp[0x24] = temp_register
CTX[0x00] += 18
EAX = VM_loop
ESP = CTX[0x20]
JMP CTX[0x10] ; Ret-to-EAX epilogue
```

# 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

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

# Virtualization analysis

Ocpode 0x08: Dereference temp register

```
CTX[0x08] = DWORD PTR[CTX[0x08]]
CTX[0x00] += 0x18
EAX = VM_loop
ESP = CTX[0x20]
JMP CTX[0x10] ; Ret-to-EAX epilogue
```

# Virtualization analysis

Opcode 0x09: Temp register to native register

```
index = 7 - BYTE PTR:CTX[0x34]
saved_esp = CTX[0x20]
temp = CTX[0x08]
saved_esp[index*4+4] = temp
CTX[0x00] += 0x18
EAX = VM_loop
ESP = CTX[0x20]
JMP CTX[0x10] ; Ret-to-EAX epilogue
```

# 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)

```
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: XX02 -> 7402 -> JZ start+4
start+2: F8 -> CLC
start+3: B0F9 -> 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 040600008B3D3C10 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 040600008D85C0FD 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 != 6ba1d002-21ed-4dbe-afb5-08cf8b81ca32
  - SOFTWARE\Microsoft\Windows NT\CurrentVersion\DigitalProductId != 55274-649-6478953-23109, A22-00001, 47220
  - HARDWARE\Description\System\SystemBiosDate != 01/02/03
  - GetVersion() != 5 (major version)
  - CS (code segment) == 0x1b || 0x23 (user-mode check?)
  - Hashes module path (and all its substrings) and checks that hash != 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

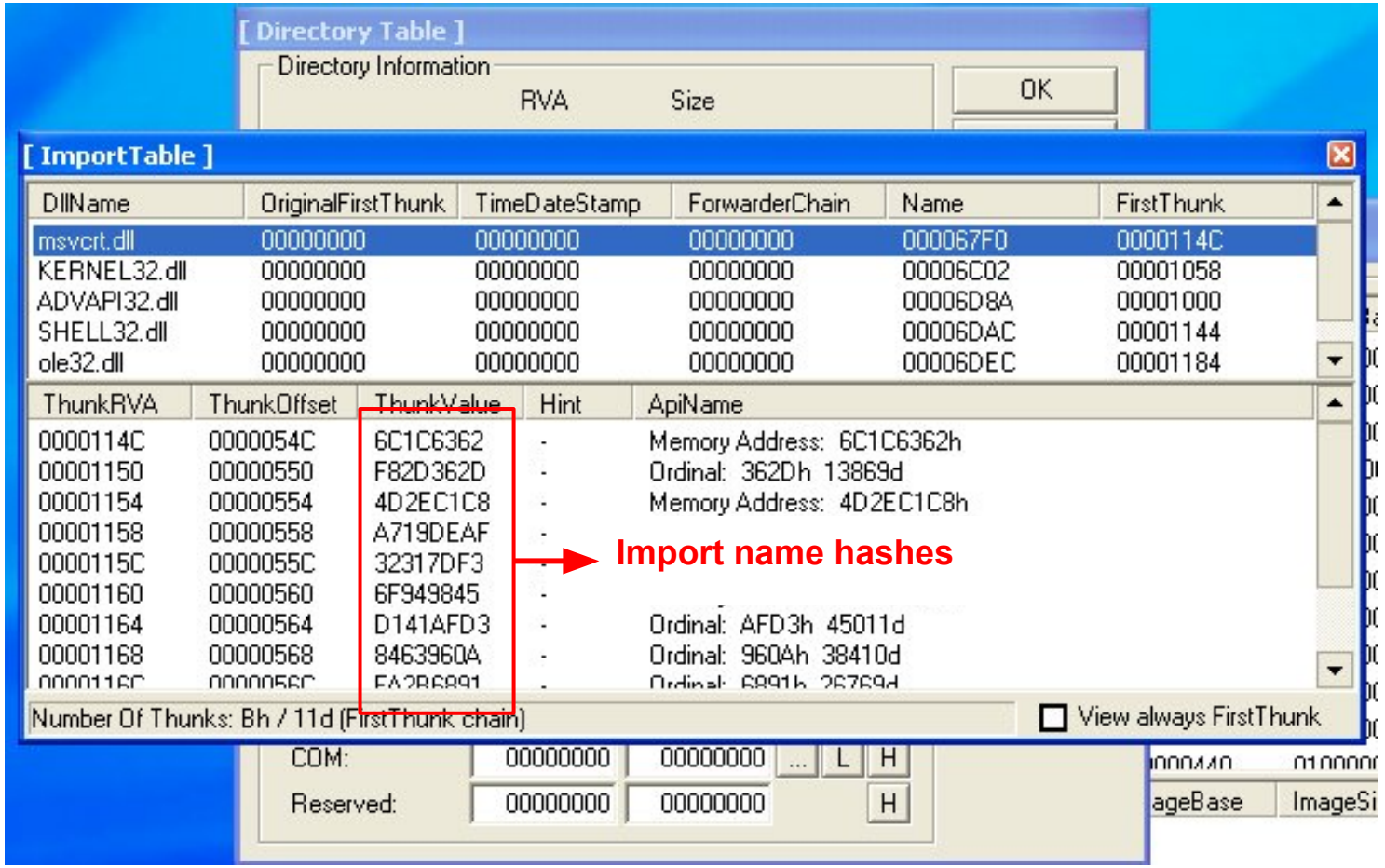
- Misc anti-*:
  - Manual load of DLLs. Open, read, apply relocs and then parse export directory to resolve APIs by hash.
  - Opens JobObjects with names like `Local\COMODO_SANDBOX_0x%X_R%d` (%X is PID and %d is in range [1-6]).
    - If it succeeds, call BasicUIRestrictions and ExtendLimitInformation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

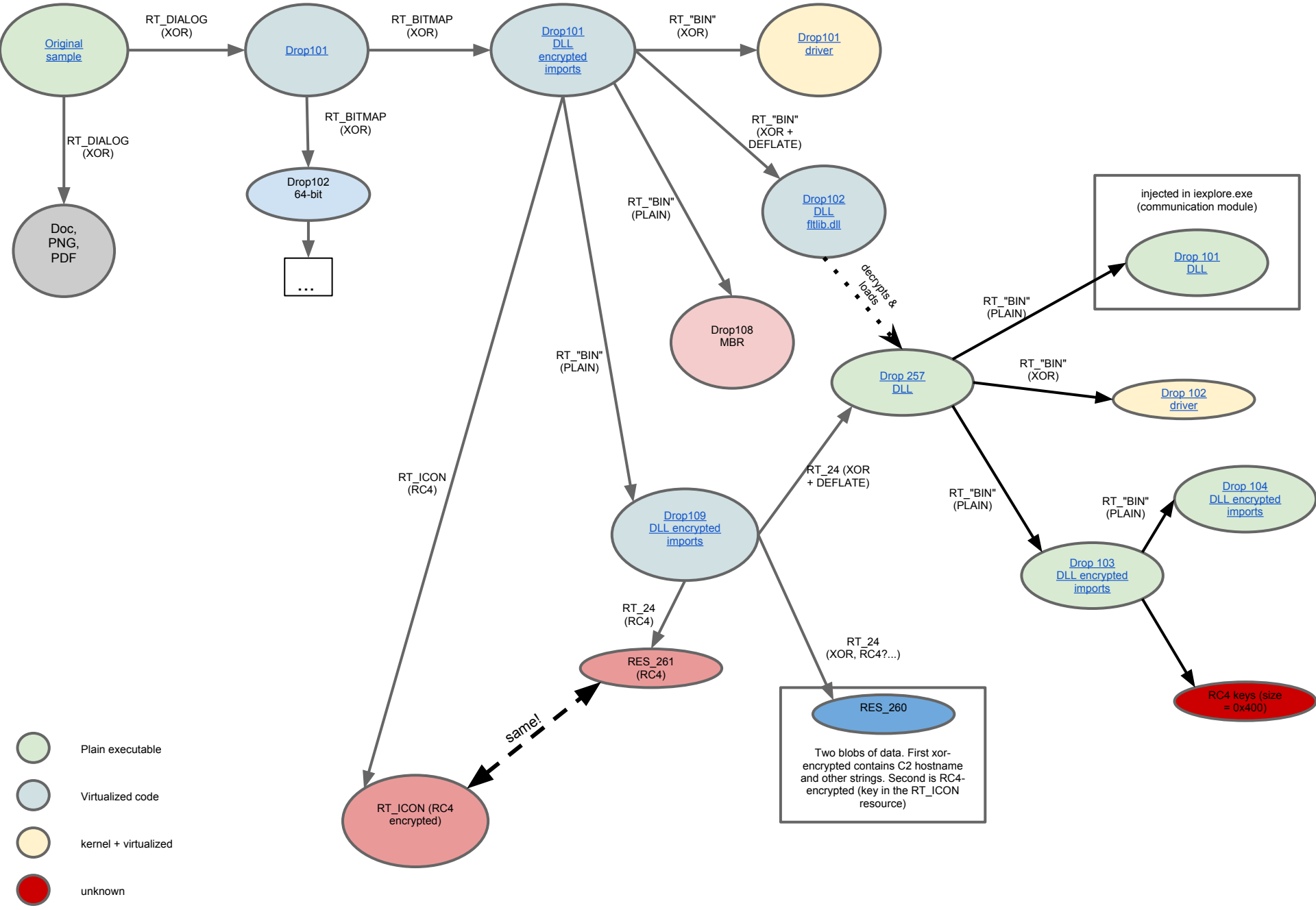
# Analysis of de-virtualized code

- Misc anti-*:
  - DLLs with invalid IATs



# 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...


Original sample
RT_DIALOG (XOR)
Drop101
RT_BITMAP (XOR)
Drop101 DLL encrypted imports
RT_"BIN" (XOR)
Drop101 driver
RT_DIALOG (XOR)
Doc, PNG, PDF
RT_BITMAP (XOR)
Drop102 64-bit
...
RT_"BIN" (XOR + DEFLATE)
Drop102 DLL fitlib.dll
RT_"BIN" (PLAIN)
Drop108 MBR
decrypts & loads
injected in iexplore.exe (communication module)
Drop 101 DLL
RT_"BIN" (PLAIN)
Drop 257 DLL
RT_"BIN" (XOR)
Drop 102 driver
RT_"BIN" (PLAIN)
RT_ICON (RC4)
Drop109 DLL encrypted imports
RT_24 (XOR + DEFLATE)
RT_"BIN" (PLAIN)
Drop 103 DLL encrypted imports
RT_"BIN" (PLAIN)
Drop 104 DLL encrypted imports
RT_24 (RC4)
RES_261 (RC4)
RT_24 (XOR, RC4?...)
RES_260
Two blobs of data. First xor-encrypted contains C2 hostname and other strings. Second is RC4-encrypted (key in the RT_ICON resource)
RC4 keys (size = 0x400)
same!
RT_ICON (RC4 encrypted)
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