Anti-Analysis Techniques

Prepared By

EG|CERT Team

Agenda

- Introduction
- Types of Anti-Analysis Approaches
- Avoid Being Analyzed by Tools Approach
  - Debugger Detection
  - Virtualization Detection
  - Analysis Tools Detection
- Avoid Being Analyzed by Analyst Approach
  - TLS Callback
  - Junk Code Insertions
  - Code Transportation
  - Proxy Function
  - Anti-Disassembler

Introduction

Types of Anti-Analysis Approaches

- To avoid being analyzed by tools
  - Depending on detecting analysis tools and avoiding execution under these tools.
- To avoid being analyzed by analyst
  - Depending on adding multiple layers of complexity to make analysis process more difficult.
  - Confuse analyst by adding non-related codes and intersected execution flow.

Avoid Analysis by Tools Approach

- Debugger Detection
  - Detection by flags
  - Detection by parent process
  - Detection by execution timing
  - Detection by breakpoints

Debugger Detection by Flags

- *BeingDebugged*
- The simplest way to detect debugger.
- Could be done using Windows APIs

IsDebuggerPresent

CheckRemoteDebuggerPresent

- Could be done manually by checking *BeingDebugged* flag in PEB.

```
mov eax, dword ptr fs:[30h] ; PEB
cmp byte ptr [eax+2], 1 ; PEB.BeingDebugged
jz <debugged>
```

Debugger Detection by Flags

- *NtGlobalFlag*
- It exists in PEB at offset 0x68 in 32-bit systems and 0xBC in 64-bit systems.
- In normal execution its value is **zero**
- Under debugger its value is **0x70**

FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)

Debugger Detection by Flags

```
.text:00403594    mov    eax, large fs:30h      ; PEB struct loaded into EAX
.text:0040359A    mov    eax, [eax+68h]         ; NtGlobalFlag (offset 0x68 relative to PEB) saved to EAX
.text:0040359E    sub    eax, 70h               ; Value 0x70 corresponds to all flags on
                                                (FLG_HEAP_ENABLE_TAIL_CHECK,
                                                FLG_HEAP_ENABLE_FREE_CHECK,
                                                FLG_HEAP_VALIDATE_PARAMETERS)
.text:004035A1    mov    [ebp+var_1828], eax
.text:004035A7    cmp    [ebp+var_1828], 0      ; Check whether 3 debug flags were on (result of subtraction
                                                should be 0 if debugged)
.text:004035AE    jnz    short loc_4035B5       ; No debugger, program continues...
.text:004035B0    call   s_selfDelete           ; ...else, malware deleted
```

Debugger Detection by Parent Process

- In normal execution, process of malware should be a child process of the process of infection vector (browser, mail client, …etc)
- While debugging, the parent process will be the debugger process.
- Malware could check its parent process to detect debugger.
- This technique done by getting parent process ID then getting parent process name or parent process file name.

Debugger Detection by Parent Process

- By using *CreateToolhelp32Snapshot*, *Process32First* and *Process32Next* Windows APIs, malware can loop all running processes to determine its parent process
- Also by using undocumented *NtQueryInformationProcess* API, malware can get its parent process ID then by using *GetProcessImageFileNameA* get its file name.
- To use *GetProcessImageFileNameA* you need to have a handle to the process which could be got by using *openprocess* API with PROCESS_QUERY_INFORMATION permission.

Debugger Detection by Execution Timing

- This technique depends on the difference between machine speed and human speed.
- Malware uses *rdtsc* instruction that get system time in sequence then calculate difference between two results which should be in range of milliseconds.
- There are different API doing the same function of *rdtsc*

  GetLocalTime
  GetSystemTime
  GetTickCount
  QueryPerformanceCounter
  timeGetTime
  timeGetSystemTime

Debugger Detection by Execution Timing

```
.text:00401010    RDTSC                          ;get time 1st value
.text:00401012    PUSH    EAX                    ;save 1st value in the stack
.text:00401013    XOR     EAX, EAX
.text:00401015    RDTSC                          ;get time 2nd value
.text:00401017    SUB     EAX, dword ptr [ESP]   ;calculate the difference
.text:0040101a    CMP     EAX, 0x20              ; more than 20 milliseconds, detect a single-
                                                 stepping
.text:0040101d    JA      Debugger_Detected
.text:0040101f    PUSH    0x00
.text:00401021    CALL    ExitProcess
```

Debugger Detection by Break Points

- Single-stepping is the basic concept of break points.
- *Trap flag* is a flag in FLAGS register that interrupt execution after each instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	O	D	I	T	S	Z	X	Ac	X	P	X	Cy

- Detecting *trap flag* means malware is being debugged.
- Detecting *trap flag* is very hard for different reasons
  - FLAGS register is not readable like other registers.
  - Trap flag is always cleared after returning execution to the debugger.

Debugger Detection by Break Points

- The instruction *pop SS* is a special one.
- Due to its nature, exception not allowed during its execution.
- These unallowed exception includes single-stepping.

```
PUSH    SS
POP     SS
PUSHFD
NOP             ; debugger will bypass previous instruction and stop on this instruction
                The value of FLAG register will be stored in stack
```

Debugger Detection by Break Points

- Hardware breakpoints are stored in DR0➔DR3
- The main way to detect hardware breakpoint is via **Thread Context**.
- Two common ways to do that:
  - Using SEH (Structured Exception Handling) adding exception callback that check and modify debug registers, then to cause an exception.
  - Using Windows APIs

    GetThreadContext
    SetThreadContext

Debugger Detection by Break Points

- Software breakpoints is just a code patching.
- It could be detected by different ways:
  - Searching code for 0xcc (INT3).
  - Calculate checksum of the code then compare it with a precalculated one.
  - Read the code section from the malware file and compare it with the one in memory.

Avoid Analysis by Tools Approach

- Virtualization Detection
  - Detection by Processes
  - Detection by WMI
  - Detection by Registry
  - Detection by MMX Registers

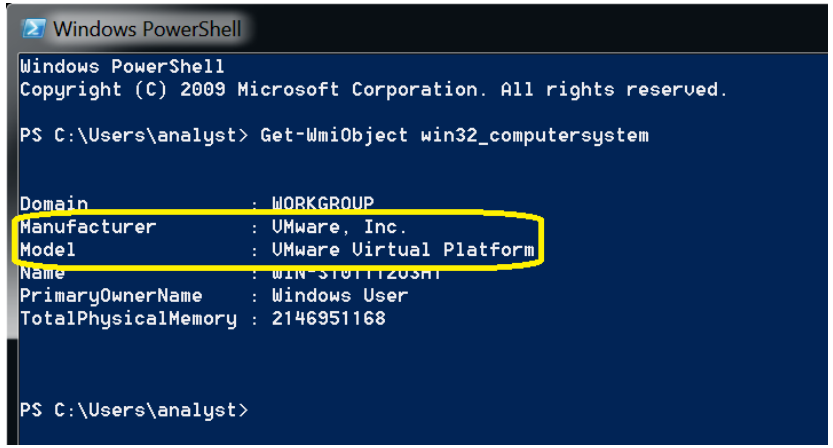
Virtualization Detection by Processes

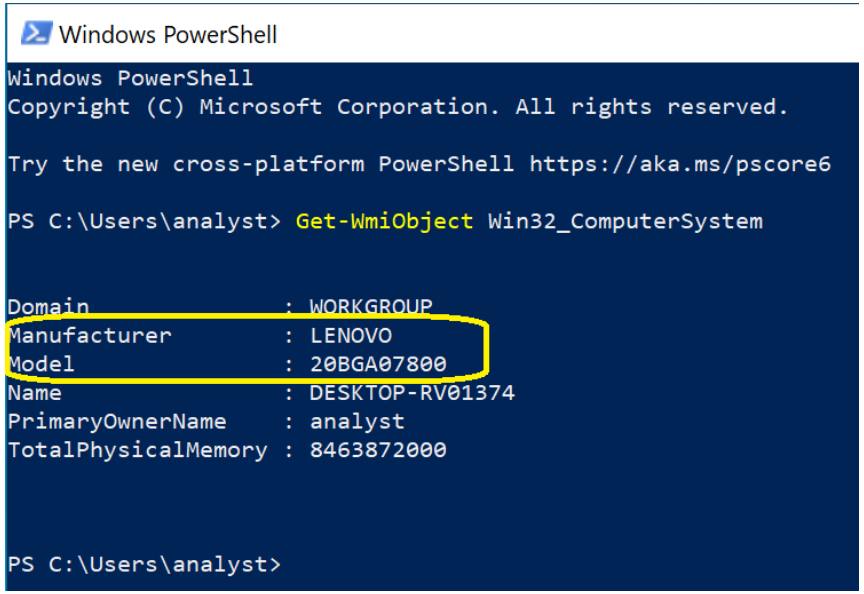
- Virtual machines are using some processes to enhance its functionality.
- Detecting these processes reveal that the malware executed in virtual environment.
- As previous, we can use *CreateToolhelp32Snapshot*, *Process32First* and *Process32Next* Windows APIs to scan running processes.

vmtoolsd.exe
vmacthlp.exe
VMwareUser.exe
VMwareService.exe
VMwareTray.exe
VBoxService.exe
VBoxTray.exe

Virtualization Detection by WMI

- Through powershell, the tool *Get-WmiObject* could provide information about the system





Virtualization Detection by WMI

- This information can also be accessed through a WMI query, such as the following:

*SELECT * FROM Win32_ComputerSystem WHERE Manufacturer LIKE "%VMware%" AND Model LIKE "%VMware Virtual Platform%"*

- For Microsoft Hyper-V, it would be as follows:

*SELECT * FROM Win32_ComputerSystem WHERE Manufacturer LIKE "%Microsoft Corporation%" AND Model LIKE "%Virtual Machine%"*

Virtualization Detection by Registry

- In virtual machines there should be registry keys related to hardware like hard desk or running processes like vmtools.
- Detecting these keys reveal that the malware executed in virtual environment.

- *HKLM\HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id 0*
- *HKLM\SOFTWARE\VMware, Inc.\VMware Tools*
- *HKLM\HARDWARE\\ACPI\\DSDT\\VBOX__*
- *HKLM\SOFTWARE\\Oracle\\VirtualBox Guest Additions*

Virtualization Detection by MMX Registers

- MMX registers are a set of registers that were introduced by Intel that help speed up graphics calculations.
- Some virtualization tools don't support them.
- Malware tries to use them to check their presence. If they weren’t exist, then it could be virtual environment.

Avoid Analysis by Tools Approach

- Analysis Tools Detection
  - Detection by Name
  - Detection by Window Attributes
  - SandBox Detection

Tools Detection by Name

- Again by using *CreateToolhelp32Snapshot*, *Process32First* and *Process32Next* Windows APIs, malware can scan all running processes and compare their names with the list of well-known analysis tools
- On matching, malware has two options
  - Terminate itself
  - Terminate the matched process using *TerminateProcess* API

Tools Detection by Window Attributes

- To avoid detection, some analysts change the name of analysis tools.
- Malware could also search for tools by window name or window class.
- Window name is more descriptive then process name.

Tools Detection by Window Attributes

- There are two method to detect analysis tools by window name or class.
  - *FindWindow* API and search for a list of well-known analysis tools
  - *EnumWindows* to enumerate all windows then *GetWindowText* API with the enumerated handls

SandBox Detection

- Malware could detect SandBox by different methods
  - Detecting human behavior (like mouse pointer movements)
  - Checking presence of some SB-related libraries.

    api_log.dll

    dir_watch.dll

    sbiedll.dll
  - Delaying execution as SB execute malware for certain limited time then generate its report.

Avoid Analysis by Analyst Approach

- TLS Callback
- Junk Code Insertions
- Code Transportation
- Proxy Function
- Anti-Disassembler

TLS Callback

- Thread Local Storage callback.
- A functions that were designed mainly to initialize and clear TLS data objects.
- TLS callback always executed before the malware entry point.

Junk Code Insertion

- The malware author inserts lots of code that never gets executed.
- Junk code inserted after unconditional jump or conditional jump with conditions that would never be met.
- Other technique is to add code with no effect

```
Mov     EAX,EBX
Inc     EAX
Sub     EAX, 0x1
Mov     EBX,EAX
```

```
Push    ECX
Push    EDI
Add     ESP, 0x8
```

```
Sub     ESI, 0x8
Push    ESI
Add     [ESP], 0xC
Sub     [ESP], 0x4
Pop     ESI
```

- The purpose of this code is to waste the analyst's time and make the code graph look more complicated than it actually is.

Code Transportation

- A technique that depends on rearranging the code by using different jump instructions.

```
00401005  v EB 20          JMP SHORT Test.00401027
00401007    53             PUSH EBX
00401008    3E:8F05 74F940 POP DWORD PTR DS:[40F974]
0040100F    D3DB           RCR EBX,CL
00401011    0FCB           BSWAP EBX
00401013    68 5C104000    PUSH Test.0040105C
00401018    5B             POP EBX
00401019    3E:8903        MOV DWORD PTR DS:[EBX],EAX
0040101C    43             INC EBX
0040101D    0FBDC2         BSR EAX,EDX
00401020    A9 46A978DC    TEST EAX,DC78A946
00401025  v EB 0B          JMP SHORT Test.00401032
00401027    8BF0           MOV ESI,EAX
00401029    3E:8A00        MOV AL,BYTE PTR DS:[EAX]
0040102C    84C0           TEST AL,AL
0040102E  v 74 2A          JE SHORT Test.0040105A
00401030  ^ EB D5          JMP SHORT Test.00401007
00401032    8BC2           MOV EAX,EDX
00401034    52             PUSH EDX
00401035    B6 86          MOV DH,86
00401037    B3 27          MOV BL,27
00401039    B8 7CFAA17F    MOV EAX,7FA1FA7C
0040103E  v EB 01          JMP SHORT Test.00401041
00401040    90             NOP
00401041    0FBCC2         BSF EAX,EDX
00401044    3E:C705 FC8841 MOV DWORD PTR DS:[4188FC],0
0040104F    2D 210DE8B9    SUB EAX,B9E80D21
00401054    69DA E577D49D  IMUL EBX,EDX,9DD477E5
```

Proxy Function

- A technique that uses a proxy function to calculate the address of the required function then move execution to it.
- This technique was used widely in Nymaim banking Trojan.

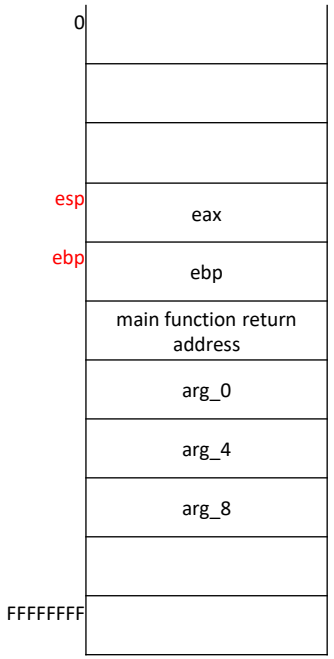
Proxy Function

```
:01FA16B5 6A FF                push    0FFFFFFFFh
:01FA16B7 50                   push    eax
:01FA16B8 68 04 1F 29 12       push    12291F04h
:01FA16BD 68 A9 5A 27 12       push    12275AA9h
:01FA16C2 E8 C7 EC 01 00       call    sub_1FC038E
```

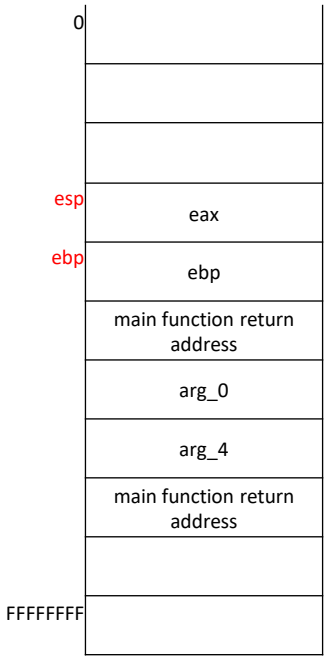
```
; Attributes: bp-based frame

sub_1FC038E proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h

; FUNCTION CHUNK AT 01FA1E8D SIZE 00000008 BYTES

push    ebp
mov     ebp, esp
push    eax
mov     eax, [ebp+4]
mov     [ebp+arg_8], eax
mov     eax, [ebp+arg_4]
sub     eax, [ebp+arg_0]
jmp     loc_1FA1E8D
sub_1FC038E endp
```

```
; START OF FUNCTION CHUNK FOR sub_1FC038E

loc_1FA1E8D:
add     [ebp+4], eax
pop     eax
leave
retn    8
; END OF FUNCTION CHUNK FOR sub_1FC038E
```

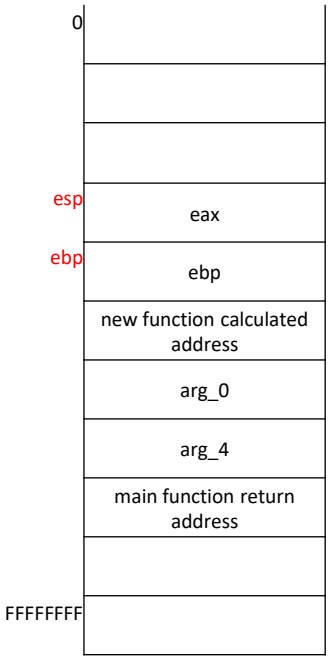
Proxy Function



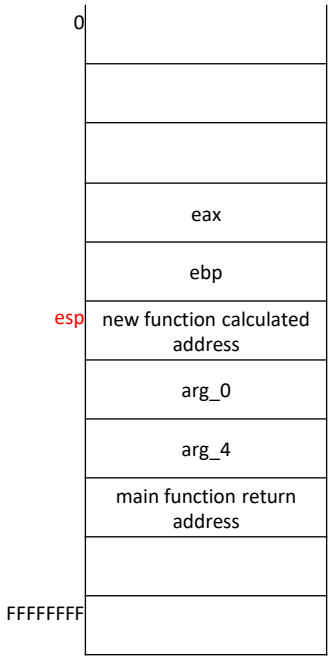

```
Push    ebp
Mov     ebp,esp
Push    eax
```




```
Mov     eax,[ebp+4]
Mov     [ebp+arg_8],eax
```




```
Mov     eax,[ebp+arg_4]
Sub     eax,[ebp+arg_0]
Jmp     loc_1FA1E8D
Add     [ebp+4],eax
```




```
Pop     eax
Leave
Retn    8
```

Anti-Disassembler

- A techniques designed to frustrate the disassembly process.
- It aims to prevent the disassembler from finding the correct starting address for one or more instructions.
- Forcing the disassembler to lose track of itself in this manner usually results in a failed or, at a minimum, incorrect disassembly listing.

Anti-Disassembler

```
.text:0A04B0D1    call    near ptr loc_A04B0D6+1
.text:0A04B0D6    mov     dword ptr [eax-73h], 0FFEB0A40h
.text:0A04B0DD    loopne loc_A04B06F
.text:0A04B0DF    mov     dword ptr [eax+56h], 5CDAB950h
.text:0A04B0E6    iret
.text:0A04B0E7    db      47h
.text:0A04B0E8    db      31h, 0FFh, 66h
.text:0A04B0EB    mov     edi, 0C7810D98h
```

Anti-Disassembler

```
.text:0A04B0D1    call    loc_A04B0D7
.text:0A04B0D6    db       0C7h
.text:0A04B0D7    pop     eax
.text:0A04B0D8    lea     eax, [eax+0Ah]
.text:0A04B0DB    jmp     short near ptr loc_A04B0DB+1
.text:0A04B0DD    db      0E0h
```

Anti-Disassembler

```
.text:0A04B0D1    call    loc_A04B0D7
.text:0A04B0D6    db      0C7h
.text:0A04B0D7    pop     eax
.text:0A04B0D8    lea     eax, [eax+0Ah]
.text:0A04B0DB    db      0EBh
.text:0A04B0DC    jmp     eax
```


Questions