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

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

\begin{itemize}
  \item FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
  \item FLG_HEAP_ENABLE_FREE_CHECK (0x20)
  \item FLG_HEAP_VALIDATE_PARAMETERS (0x40)
\end{itemize}
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

12/21/2021

Egyptian Computer Emergency Readiness Team
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.

![FLAGS register diagram]

• 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 methods 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 handles
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

<table>
<thead>
<tr>
<th>Mov</th>
<th>EAX,EBX</th>
<th>Push</th>
<th>ECX</th>
<th>Sub</th>
<th>ESI, 0x8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inc</td>
<td>EAX</td>
<td>Push</td>
<td>EDI</td>
<td>Push</td>
<td>ESI</td>
</tr>
<tr>
<td>Sub</td>
<td>EAX, 0x1</td>
<td>Add</td>
<td>ESP, 0x8</td>
<td>Add</td>
<td>[ESP], 0xC</td>
</tr>
<tr>
<td>Mov</td>
<td>EBX,EAX</td>
<td>Sub</td>
<td>[ESP], 0xC</td>
<td>Pop</td>
<td>ESI</td>
</tr>
</tbody>
</table>

• 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.
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
Proxy Function

```
Push  ebp
Mov   ebp,esp
Push  eax

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