06 - Anti Dynamic Analysis

CYS5120 - Malware Analysis

Bahcesehir University

Cyber Security Msc Program

Dr. Ferhat Ozgur Catak [1] Mehmet Can Doslu [2]

[1] ozgur.catak@tubitak.gov.tr

[2] mehmetcan.doslu@tubitak.gov.tr

2017-2018 Fall

Table of Contents

Table of Contents

Anti-Debugging

Anti-Debugging

- **Anti-debugging** is a popular **anti-analysis technique** used by malware to recognize when it is under the control of a debugger or to thwart debuggers.
- Malware authors know that **malware analysts use debuggers** to figure out how malware operates, and the authors use anti-debugging techniques in an attempt to **slow down the analyst as much as possible**.
  - Once malware realizes that **it is running in a debugger**, it may **alter its normal code execution path** or **modify the code to cause a crash**.
  - The analysts' attempt to understand it, and adding time and additional overhead to their efforts.

Windows Debugger Detection

Using the Windows API

- The Windows API provides several functions that can be used by a program to determine if it is being debugged.
  - Some of these functions were designed for debugger detection;
  - others were designed for different purposes but can be repurposed to detect a debugger.

Using the Windows API I

IsDebuggerPresent

```
BOOL WINAPI IsDebuggerPresent(void);
```

- This function searches the **Process Environment Block (PEB) structure** for the field *IsDebugged*.
- If the current process is running in the context of a debugger, the return value is nonzero.
- If the current process is not running in the context of a debugger, the return value is zero.
- **Remark**
  - This function allows an application to determine whether or not it is being debugged, so that it can modify its behavior.
  - For example, an application could provide additional information using the *OutputDebugString* function if it is being debugged.
- To determine whether a remote process is being debugged, use the *CheckRemoteDebuggerPresent* function.

Using the Windows API II

CheckRemoteDebuggerPresent

```
BOOL WINAPI CheckRemoteDebuggerPresent(
_In_    HANDLE hProcess,
_Inout_ PBOOL  pbDebuggerPresent);
```

- **Parameters**
  - *hProcess* [in] : A handle to the process.
  - *pbDebuggerPresent* [in, out] : A pointer to a variable that the function sets to **TRUE** if the specified process is being debugged, or **FALSE** otherwise.
- **Return value**
  - If the function succeeds, the return value is nonzero otherwise is zero.
- Determines whether the specified process is being debugged.
- The *remote* in *CheckRemoteDebuggerPresent* does not imply that the debugger necessarily resides on a different computer; instead, **it indicates that the debugger resides in a separate and parallel process**.
- Use the IsDebuggerPresent function to detect whether **the calling process is running under the debugger**.
- **This function takes a process handle as a parameter and will check if that process has a debugger attached.**

Using the Windows API III

NtQueryInformationProcess

```
NTSTATUS WINAPI NtQueryInformationProcess(
_In_      HANDLE           ProcessHandle,
_In_      PROCESSINFOCLASS ProcessInformationClass,
_Out_     PVOID            ProcessInformation,
_In_      ULONG            ProcessInformationLength,
_Out_opt_ PULONG           ReturnLength);
```

- Retrieves information about the specified process.
- The first parameter to this function is a process handle;
- The second is used to tell the function the type of process information to be retrieved.
  - For example, using the value ProcessDebugPort (value 0x7) for this parameter will tell you if the process in question is currently being debugged.
  - **If the process is not being debugged, a zero will be returned**; otherwise, a port number will be returned.

Using the Windows API IV

OutputDebugString

```
void WINAPI OutputDebugString(
_In_opt_ LPCTSTR lpOutputString);
```

- This function is used to send a string to a debugger for display.

```
DWORD errorValue = 12345;
SetLastError(errorValue);

OutputDebugString("Test for Debugger");

if(GetLastError() == errorValue)
{
  ExitProcess();
}
else
{
  RunMaliciousPayload();
}
```

- If *OutputDebugString* is called and there is no debugger attached, *GetLastError* should no longer contain our arbitrary value,
- because an error code will be set by the OutputDebugString function if it fails.

Manually Checking Structures I

Manually Checking Structures

- There are many reasons why malware authors are discouraged from using the Windows API for anti-debugging.
- For example, the API calls could be hooked by a rootkit to return false information.
- Therefore, malware authors often choose to perform the functional equivalent of the API call manually, rather than rely on the Windows API.

Manually Checking Structures II

Checking the BeingDebugged Flag

- A Windows PEB structure is maintained by the OS for **each running process**.
- It contains all user-mode parameters associated with a process.
- These parameters include the process's environment data
  - environment variables
  - the loaded modules list
  - addresses in memory
  - debugger status

```
typedef struct _PEB {
  BYTE Reserved1[2];
  BYTE BeingDebugged;
  BYTE Reserved2[1];
  PVOID Reserved3[2];
  PPEB_LDR_DATA Ldr;
  PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
  BYTE Reserved4[104];
  PVOID Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE Reserved6[128];
  PVOID Reserved7[1];
  ULONG SessionId;
} PEB, *PPEB;
```

Manually Checking Structures III

mov method	push/pop method

`mov eax, dword ptr fs:[30h]`
`mov ebx, byte ptr [eax+2]`
`test ebx, ebx`
`jz NoDebuggerDetected`
	`push dword ptr fs:[30h]`
	`pop edx`
	`cmp byte ptr [edx+2], 1`
	`je DebuggerDetected`

- While a process is running, the location of the PEB can be referenced by the location *fs:[30h]*.
- For anti-debugging, malware will use that location to check the *BeingDebugged flag*, which indicates whether the specified process is being debugged.

Manually Checking Structures IV

Checking the ProcessHeap Flag

- *Reserved4* array, known as ProcessHeap, is set to the location of a process's first heap allocated by the loader.
- ProcessHeap is located at 0x18 in the PEB structure.
- **Offset 0x10** in the heap header is the **ForceFlags** field on **Windows XP**, but for Windows 7, it is at offset **0x44 for 32-bit applications**.

```
mov eax, large fs:30h          ; PEB saved to EAX
mov eax, dword ptr [eax+18h]   ; ProcessHeap (offset 0x18 relative to PEB)
                               ; saved to EAX
cmp dword ptr ds:[eax+10h], 0  ; Check whether ForceFlags field
                               ; (offset 0x10 relative to ProcessHeap) is 0
jne DebuggerDetected           ; If previous test returned non zero,
                               ; debugger is present
```

Manually Checking Structures V

Checking NTGlobalFlag

- The PEB has a field called NtGlobalFlag (offset 0x68) which programs can challenge to identify whether they are being debugged.
- Normally, when a process is not being debugged, the *NtGlobalFlag* field contains the value **0x0**.
- When the process is being debugged, the field will usually contain the value 0x70 which indicates that the following flags are set:
  (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)

```
mov eax, large fs:30h
cmp dword ptr ds:[eax+68h], 70h
jz DebuggerDetected
```

Checking for System Residue I

Checking for System Residue

- When analyzing malware, we typically use debugging tools, which **leave residue on the system**.
- Malware can search for this residue in order to determine when you are attempting to analyze it,
  - Such as by searching registry keys for references to debuggers. `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug`
- Malware can also **search the system for files and directories**,
  - such as common debugger program executables, which are typically present during malware analysis.
- Or the malware can detect residue in live memory, by viewing the current process listing or, more commonly, by performing a *FindWindow* in search of a debugger

Checking for System Residue II

```
HWND WINAPI FindWindow(
_In_opt_ LPCTSTR lpClassName,
_In_opt_ LPCTSTR lpWindowName);
```

```
if(FindWindow("OLLYDBG", 0) == NULL)
{
//Debugger Not Found
}
else
{
//Debugger Detected
}
```

Identifying Debugger Behavior

Identifying Debugger Behavior

- debuggers can be used to set breakpoints or to single-step through a process in order to aid the malware analyst in reverse-engineering.
- Several anti-debugging techniques are used by malware to detect this sort of debugger behavior:
  - INT scanning
  - checksum checks
  - timing checks

INT Scanning I

INT Scanning

- INT 3 is the **software interrupt used by debuggers** to temporarily replace an instruction in a running program and to call the debug exception handler.
- Whenever you set a breakpoint at a location, **the debugger replaces the FIRST byte of that instruction with INT 3** (a one-byte instruction), and saves the old byte.
- Whenever the program executes to that location, an interrupt is generated and the debugger is called to handle that exception.
- The opcode for INT 3 is 0xCC.

```
call $+5
pop edi
sub edi, 5
mov ecx, 400h
mov eax, 0CCh
repne scasb
jz DebuggerDetected
```

Performing Code Checksums

Performing Code Checksums

- Malware can calculate a checksum on a section of its code to accomplish the same goal as scanning for interrupts.
- This check simply performs a cyclic redundancy check (CRC) or a MD5 checksum of the opcodes in the malware.

Timing Checks I

Timing Checks

- The most popular ways for malware to detect debuggers because processes run more slowly when being debugged.
- There are a couple of ways to use timing checks to detect a debugger:
  - Record a timestamp, perform a couple of operations, take another timestamp, and then compare the two timestamps. If **there is a lag**, you can assume **the presence of a debugger**.
  - Take a timestamp before and after raising an exception. If a process is not being debugged, the exception will be handled really quickly; a debugger will handle the exception much more slowly.
- Using the **rdtsc** Instruction

Timing Checks II

Using the *rdtsc* Instruction

- The most common timing check method uses the *rdtsc* instruction (opcode 0x0F31),
  - which returns the count of the number of ticks since the last system reboot as a 64-bit value placed into EDX:EAX.
- Malware will simply execute this instruction twice and compare the difference between the two readings.

```
rdtsc
xor ecx, ecx
add ecx, eax
rdtsc
sub eax, ecx
cmp eax, 0xFFF ❶
jb NoDebuggerDetected
rdtsc
push eax ❷
ret
```

Timing Checks III

Using QueryPerformanceCounter and GetTickCount

- *QueryPerformanceCounter* can be called to query this counter twice in order to get a time difference for use in a comparison.
- The function GetTickCount returns the number of milliseconds that have elapsed since the last system reboot.

```
a = GetTickCount();
MaliciousActivityFunction();
b = GetTickCount();
delta = b-a;
if ((delta) > 0x1A)
{
//Debugger Detected
}
else
{
//Debugger Not Found
}
```

Interfering with Debugger Functionality

Interfering with Debugger Functionality

- Malware can use several techniques to interfere with normal debugger operation:
  - thread local storage (TLS) callbacks
  - exceptions
  - interrupt insertion

Using TLS Callbacks I

Using Thread Local Storage (TLS) Callbacks

- TLS (thread local storage) calls are subroutines that are executed before the entry point.
- There is a section in the PE header that describes the place of a TLS callback.
- Malwares employ TLS callbacks to evade debugger messages.
- A TLS callback can be used to execute code before the entry point and therefore execute secretly in a debugger.

Debugger Vulnerabilities

Debugger Vulnerabilities

- Like all software, debuggers contain vulnerabilities, and sometimes malware authors attack them in order to prevent debugging.

PE Header Vulnerabilities

PE Header Vulnerabilities

- The first technique modifies the Microsoft PE header of a binary executable, causing OllyDbg to crash when loading the executable.
- The result is an error of “Bad or Unknown 32-bit Executable File,” **yet the program usually runs fine outside the debugger**.

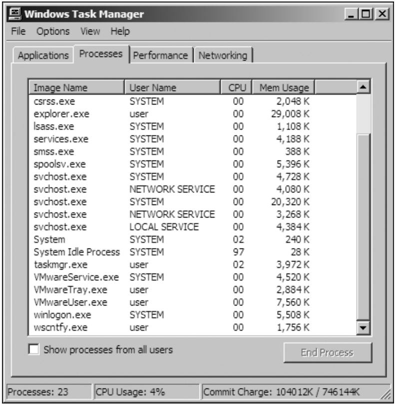
The OutputDebugString Vulnerability

- Malware often attempts to exploit a format string vulnerability in OllyDbg v1.1, by providing a string of %s as a parameter to *OutputDebugString*
- Beware of suspicious calls like *OutputDebugString* ("%s%s%s%s%s%s%s%s%s%s%s%s%s%s"). If this call executes, your debugger will crash.

Anti-Virtual Machine (anti-VM) Techniques I

VMware Artifacts

- The VMware environment leaves many artifacts on the system, especially **when VMware Tools is installed**.
- Notice that three VMware processes are running: *VMwareService.exe*, *VMwareTray.exe*, and *VMwareUser.exe*.
  - Any one of these can be found by malware as it searches the process listing for the VMware string.



Anti-Virtual Machine (anti-VM) Techniques II

VMwareService.exe

- VMwareService.exe runs the VMware Tools Service as a child of services.exe.
- It can be identified by searching the registry for services installed on a machine or by listing services using the following command:

```
C:\> net start | findstr VMware

     VMware Physical Disk Helper Service
     VMware Tools Service
```

Anti-Virtual Machine (anti-VM) Techniques III

File System and Registry

- The VMware installation directory `C:\Program Files\VMware\VMware Tools` may also contain artifacts
- A quick search for *VMware* in a virtual machine's registry might find keys like the following, which are entries that include information about the **virtual hard drive**, **adapters**, and **virtual mouse**.

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0]
"Identifier"="VMware Virtual IDE Hard Drive"
"Type"="DiskPeripheral"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Reinstall\0000]
"DeviceDesc"="VMware Accelerated AMD PCNet Adapter"
"DisplayName"="VMware Accelerated AMD PCNet Adapter"
"Mfg"="VMware, Inc."
"ProviderName"="VMware, Inc."

[HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E96F-E325-11CE-BFC1-08002BE10318}\0000]
"LocationInformationOverride"="plugged into PS/2 mouse port"
"InfPath"="oem13.inf"
"InfSection"="VMMouse"
"ProviderName"="VMware, Inc."
```

Anti-Virtual Machine (anti-VM) Techniques IV

Vulnerable Instructions

- Some instructions in x86 access hardware-based information but don't generate interrupts.
  - sidt
  - sgdt
  - sldt
  - cpuid

Anti-Virtual Machine (anti-VM) Techniques V

sidt

- The *sidt* instruction writes the 6-byte Interrupt Descriptor Table (IDT) register to a specified memory region.
  - There is only one Interrupt Descriptor Table Register (IDTR), one Global Descriptor Table Register (GDTR) and one Local Descriptor Table Register (LDTR) per processor.

IDTR Register

	IDT Base Address				IDT Limit	
VMware (hex)	FF	??	??	??	??	??
VMware (bin)	11111111	????????	????????	????????	????????	????????
Byte Offset	0x5	0x4	0x3	0x2	0x1	0x0

The IDT is at:

- `0x80ffffff` in Windows
- `0xe8XXXXXX` in Virtual PC
- `0xffXXXXXX` in VMware

Anti-Virtual Machine (anti-VM) Techniques VI

```
lea     eax, [ebp+Dst]
sidt    fword ptr [eax] ; Contents of IDTR saved to memory location
                        ; pointed to by EAX
mov     al, [eax+5]     ; Start of base memory address (5th byte offset)
                        ; saved to AL
cmp     al, 0FFh        ; Check whether it is 0xFF (VMware signature)
jnz     short loc_401E19
```

Countermeasure to sidt

- run on a multicore processor machine
- NOP-out the sidt instruction
- Modify the jump following the test

Anti-Virtual Machine (anti-VM) Techniques VII

Querying the I/O Communication Port

- The *in* instruction reads from a port (serial and printer ports, keyboard, mouse, temperature sensors, ...)
  `in dest, src`
- VMware actually monitored the use of the in instruction and capture the I/O destined for the communication channel port 0x5668 (VX)

```
mov eax, 'VMXh'
mov ecx, 0ah      ; get Vmware version
mov dx, 'VX'
in eax, dx
cmp ebx, 'VMXh'
je detected
```

The easiest way to overcome this technique is to NOP-out the in instruction or to patch the conditional jump to allow it regardless of the outcome of the comparison.

Anti-Virtual Machine (anti-VM) Techniques VIII

ScoopyNG

ScoopyNG (`http://www.trapkit.de/`) is a free VMware detection tool that implements seven different checks for a virtual machine, as follows:

- The first three checks look for the sidt, sgdt, and sldt (Red Pill and No Pill) instructions.
- The fourth check looks for str.
- The fifth and sixth use the backdoor I/O port 0xa and 0x14 options, respectively.
- The seventh check relies on a bug in older VMware versions running in emulation mode.

Anti-Virtual Machine (anti-VM) Techniques IX

Tweaking Settings

- There are also a number of undocumented features in VMware that can help mitigate anti-VMware techniques.

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

Table of Contents

Packers and Unpacking I

Packer Anatomy

- All packers take an executable file as input and produce an executable file as output.
- Most packers use a compression algorithm to compress the original executable.

Packers and Unpacking II

The Unpacking Stub

- Nonpacked executables are loaded by the OS.
- With packed programs, the unpacking stub is loaded by the OS, and then the unpacking stub loads the original program.
- The code entry point for the executable points to the unpacking stub rather than the original code.
- The unpacking stub performs three steps:
  - Unpacks the original executable into memory
  - Resolves all of the imports of the original executable
  - Transfers execution to the original entry point (OEP)

Packers and Unpacking III

1 - Loading the Executable

- When regular executables load, a loader reads the PE header on the disk, and allocates memory for each of the executable's sections based on that header.
- The loader then copies the sections into the allocated spaces in memory.
- Packed executables also format the PE header so that the loader will allocate space for the sections.
- The unpacking stub unpacks the code for each section and copies it into the space that was allocated.

Packers and Unpacking IV

2 - Resolving Imports

- The Windows loader cannot read import information that is packed.
- For a packed executable, the unpacking stub will resolve the imports.
  - After the unpacking stub unpacks the original executable, it reads the original import information.
  - It will call *LoadLibrary* for each library, in order to load the DLL into memory,
  - It will then use *GetProcAddress* to get the address for each function.

Packers and Unpacking V

3 - The Tail Jump

- Once the unpacking stub is complete, it must transfer execution to the OEP.
- A *jump* instruction is the simplest and most popular way to transfer execution.

Packers and Unpacking VI

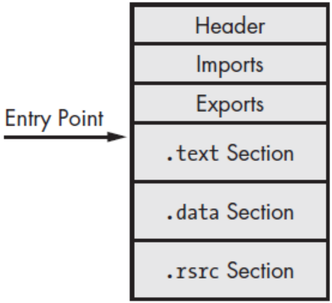


Figure 18-1: The original executable, prior to packing

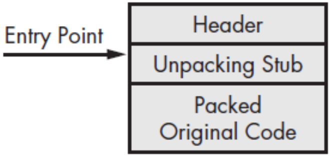


Figure 18-2: The packed executable, after the original code is packed and the unpacking stub is added

Packers and Unpacking VII

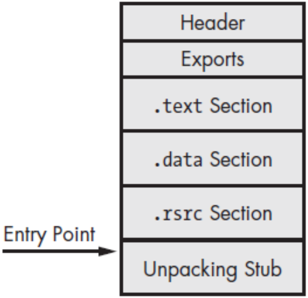


Figure 18-3: The program after being unpacked and loaded into memory. The unpacking stub unpacks everything necessary for the code to run. The program's starting point still points to the unpacking stub, and there are no imports.

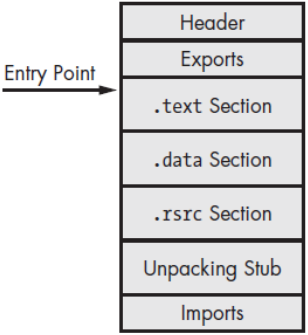


Figure 18-4: The fully unpacked program. The import table is reconstructed, and the starting point is back to the original entry point (OEP).

Identifying Packed Programs I

Indicators of a Packed Program

- The following list summarizes signs to look for when determining whether malware is packed.
  - The **program has few imports**, and particularly if the only imports are *LoadLibrary* and *GetProcAddress*.
  - When the program is opened in IDA Pro, only **a small amount of code is recognized by the automatic analysis**.
  - When the program is opened in OllyDbg, there is a warning that the program may be packed.
  - The program shows **section names that indicate a particular packer** (such as UPX0).
  - The program has abnormal section sizes, such as a .text section with a Size of Raw Data of 0 and Virtual Size of nonzero.

Identifying Packed Programs II

Unpacking Options

- There are three options for unpacking a packed executable:
  - automated static unpacking
  - automated dynamic unpacking
  - manual dynamic

Automated Unpacking

Automated Unpacking

- Automated static unpacking programs decompress and/or decrypt the executable.
- This is the fastest method, and when it works.
- **PE Explorer** comes with several static unpacking plug-ins as part of the default setup.
  - NSPack
  - UPack
  - UPX
- Automated dynamic unpackers run the executable and allow the unpacking stub to unpack the original executable code.
  - There are no good publicly available automated dynamic unpackers.

Manual Unpacking

Manual Unpacking

- There are two common approaches to manually unpacking a program
  - Discover the packing algorithm and write a program to run it in reverse. By running the algorithm in reverse, the program undoes each of the steps of the packing program.
  - Run the packed program so that the unpacking stub does the work for you, and then dump the process out of memory, and manually fix up the PE header so that the program is complete.