Difficulties of *malware analysis*

Nguyễn Chấn Việt

Đơn vị tổ chức:

Lạc Tiên

CTICT

VIA

Đơn vị tài trợ:

FPT

VNG

Malware author perspective

- Manual Analysis
- Automated Malware Analysis

Malware author perspective

- Manual Analysis
  - How to confuse Malware Analyst ?
- Automated Malware Analysis
  - How to defeat Automated Malware Analysis ?

Reverse Engineering approach

Analysis Prevention Techniques

- Anti-Disassembly
  - Protect malware from static analysis
- Anti-Debugging
  - Protect malware from dynamic analysis
- Anti-VM
  - Malware notice from VM environment
- Anti-Sandbox
  - Malware notice from sandbox environment

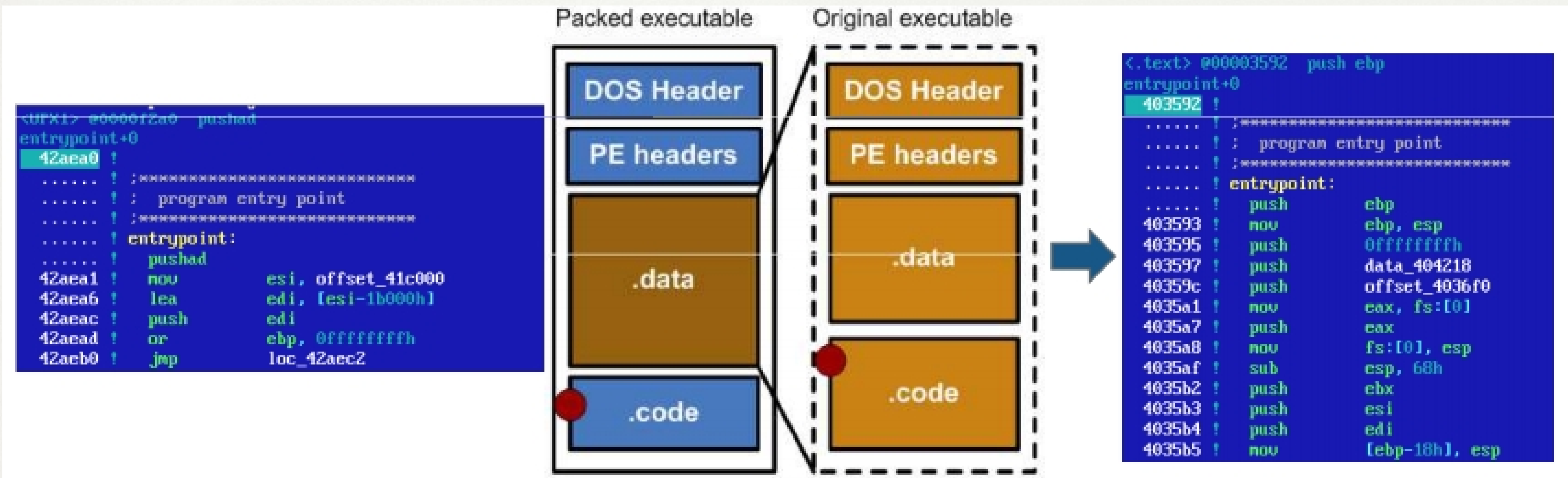
Anti-Disassembly

- Packing
- Confuse Disassembly Algorithms
- Fake Jumps
- Impossible disassembly
- …

Packing

- Packers are used to shrink the size of executable file
- Makes static analysis harder

Packing


entrypoint+0
42aea0 !
...... ! ; program entry point
...... ! entrypoint:
...... ! pushad
42aea1 ! mov esi, offset_41c000
42aea6 ! lea edi, [esi-1b000h]
42aeac ! push edi
42aead ! or ebp, 0ffffffffh
42aeb0 ! jmp loc_42aec2
Packed executable
DOS Header
PE headers
.data
.code
Original executable
DOS Header
PE headers
.data
.code
<.text> @00003592 push ebp
entrypoint+0
403592 !
...... ! ; program entry point
...... ! entrypoint:
...... ! push ebp
403593 ! mov ebp, esp
403595 ! push 0ffffffffh
403597 ! push data_404218
40359c ! push offset_4036f0
4035a1 ! mov eax, fs:[0]
4035a7 ! push eax
4035a8 ! mov fs:[0], esp
4035af ! sub esp, 68h
4035b2 ! push ebx
4035b3 ! push esi
4035b4 ! push edi
4035b5 ! mov [ebp-18h], esp

Packing

- Complex packers :
  - Multi-layer encryption
  - Advanced anti-debugging techniques
  - Code abstraction (metamorphic, virtual machines etc.)
  - Examples: Armadillo, Sdprotect, ExeCrypt, VMProtect

Packing

- Packer Analysis/Detection :
  - PEiD : it can detect more than 400 different signatures in PE files
  - RDG Packer Detector

Packing

- More :
  - The Art of Unpacking
  - Anti-Unpacking Tricks - Peter Ferrie

Anti-Disassembly

- Type of Disassembly :
  - Linear sweep :
    - Disassemble one instruction at a time
    - Do not look at type of instruction
  - Recursive traversal :
    - Look at instruction and disassemble based on program flow
    - Used by IDA Pro and other commercial products

Anti-Disassembly

- Confuse Linear Disassembly Algorithm
  - Insert Garbage byte :
    - Since data is mixed with code, data can disassemble to valid instructions

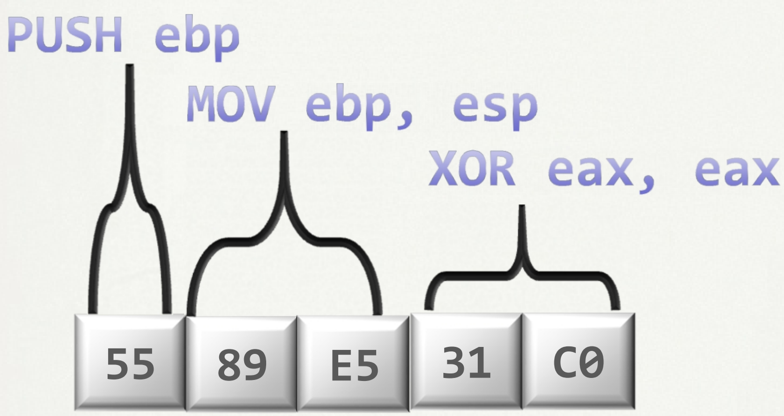
```
jmp .destination
db 0x6a ; garbage byte
.destination:
; rest of the code
pop eax
```

    - Result of disassembler :

```
eb 01     jmp 0x401003
6a 58     push 0x58
```

Anti-Disassembly

- Confuse Linear Disassembly Algorithm
  - Insert Garbage byte :



Anti-Disassembly

- Confuse Linear Disassembly Algorithm
  - Insert Garbage byte :

Anti-Disassembly

- Confuse Recursive traversal Disassembly Algorithm

  Conditional that is, say, always true

  – Jump Instructions with the Same Target

    - The most common anti-disassembly technique seen in the wild is two back-to-back conditional jump instructions that both point to the same target

    jz short near ptr loc_4011C4+1

    75 01 jnz short near ptr loc_4011C4+1

  – A Jump Instruction with a Constant Condition

    - XOR : XOR instruction immediately followed by JNZ or JZ instruction

      xor eax, eax

      jz short near ptr loc_4011C4+1

    - STC : STC instruction immediately followed by JNC or JAE instruction
    - CLC : CLC instruction immediately followed by JC or JB instruction

Anti-Disassembly

- Splicing Instructions - Called “impossible disassembly”
  - A problem of representation

```
jmp -1
;these are hidden
inc eax
dec eax
```

Anti-Disassembly

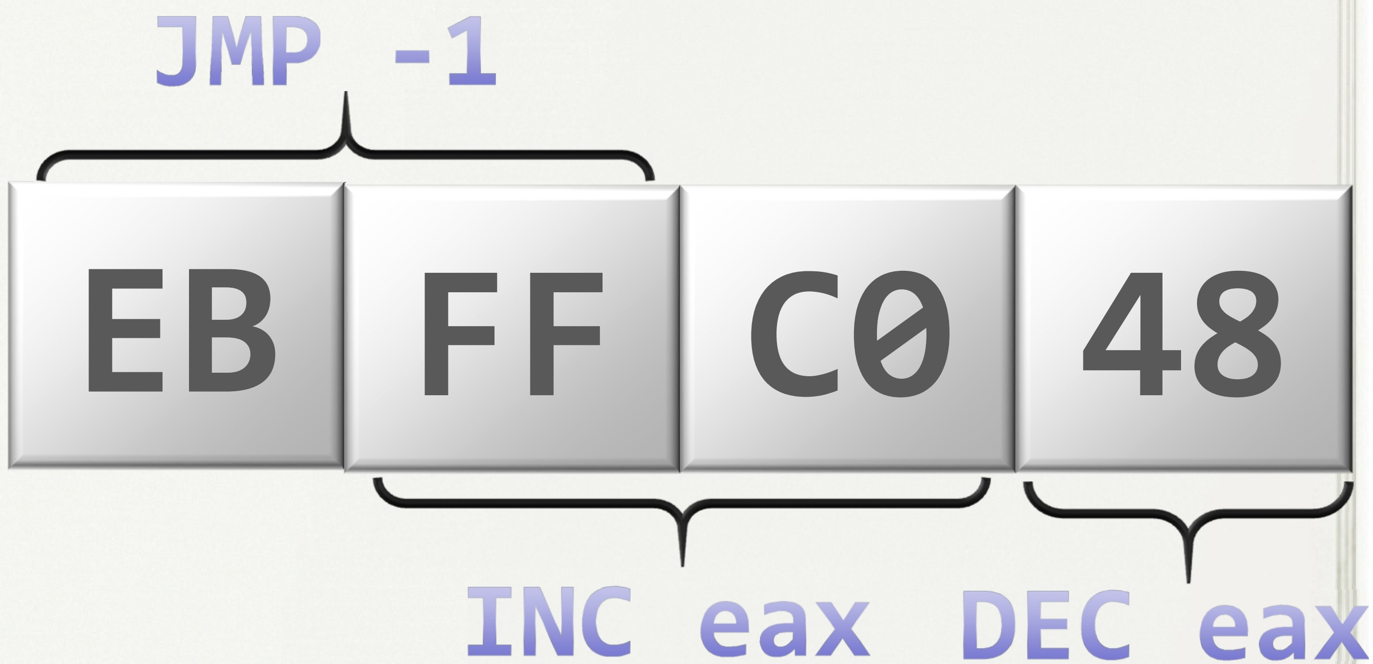
- Splicing Instructions - Called “impossible disassembly”
  - A problem of representation

jmp -1

;these are hidden

inc eax

dec eax



Anti-Disassembly

- Function pointer problems
  - It is easy to hide function calls made through pointers
- RET

004011C0	var_4	= byte ptr -4
004011C0	call	$+5
004011C5	add [esp+4+var_4], 5	
004011C9	ret	
004011C9	sp-analysis failed	
004011CA	Confused IDA Pro…..	

- Misusing Structured Exception Handlers

Anti-Disassembly

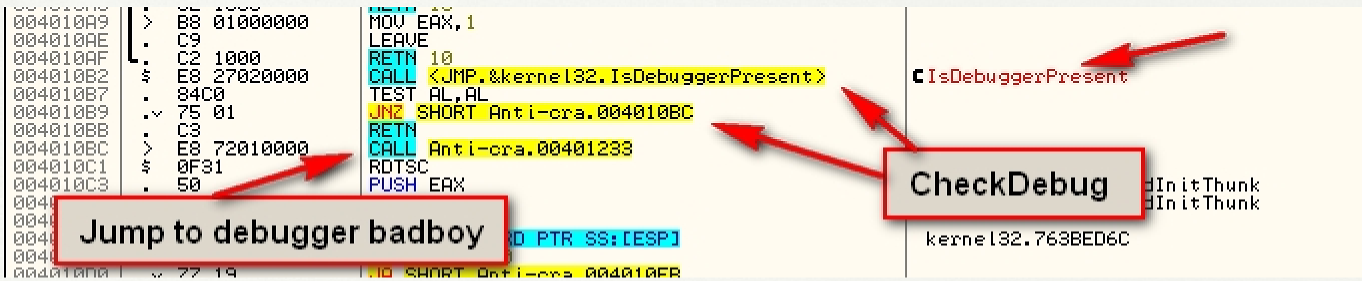
- More :
  - Practical Malware analysis – chapter 15
  - http://leetmatrix.blogspot.com/2013/02/an-anti-disassembly-trick.html

Anti-Anti-Disassembly

- IDA supports manually re-classifying code as well as code replacement to “fix” problem areas
- Deobfuscator : Deobfuscation plugin for IDA - http://code.google.com/p/optimice/
- Good malware analysts can recognize impossible assembly and run through the code to figure out what is going on

Anti-Debugging

- IsDebuggerPresent() Windows API



- if (IsDebuggerPresent()) {
- MessageBox(NULL, L"Debugger Detected Via IsDebuggerPresent",
- L"Debugger Detected", MB_OK);
- }

Anti-Debugging

- CheckRemoteDebuggerPresent() Windows API

- CheckRemoteDebuggerPresent(GetCurrentProcess(), &pbIsPresent);
- if (pbIsPresent) {
- MessageBox(NULL, L"Debugger Detected Via
- CheckRemoteDebuggerPresent", L"Debugger Detected", MB_OK);
- }

Anti-Debugging

- IsDebuggerPresent : check the PEB.BeingDebugged flag

- status = (_NtQueryInformationProcess) (hnd, ProcessBasicInformation, &pPIB, sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);
- if (status == 0) {
- if (pPIB.PebBaseAddress->BeingDebugged == 1) {
- MessageBox(NULL, L"Debugger Detected Using PEB!IsDebugged", L"Debugger Detected", MB_OK);
- } else {
- MessageBox(NULL, L"No Debugger Detected", L"No Debugger Detected", MB_OK)

Anti-Debugging

- PEB ProcessHeap Flag Debugger Detection

```
int main(int argc, char* argv[])
{
    unsigned int var;
    __asm
    {
        MOV EAX, FS:[0x30];
        MOV EAX, [EAX + 0x18];
        MOV EAX, [EAX + 0x0c];
        MOV var,EAX
    }

    if(var != 2)
    {
        printf("Debugger Detected");
    }
    return 0;
}
```

Anti-Debugging

```
0:000> dt _PEB
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void
   +0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
   +0x028 EnvironmentUpdateCount : Uint4B
   +0x02c KernelCallbackTable : Ptr32 Void
   +0x030 SystemReserved   : [1] Uint4B
   +0x034 AtlThunkSListPtr32 : Uint4B
   +0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
   +0x03c TlsExpansionCounter : Uint4B
   +0x040 TlsBitmap        : Ptr32 Void
   +0x044 TlsBitmapBits    : [2] Uint4B
   +0x04c ReadOnlySharedMemoryBase : Ptr32 Void
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
   +0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
   +0x058 AnsiCodePageData : Ptr32 Void
   +0x05c OemCodePageData  : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag     : Uint4B
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER
   +0x078 HeapSegmentReserve : Uint4B
   +0x07c HeapSegmentCommit : Uint4B
   +0x080 HeapDeCommitTotalFreeThreshold : Uint4B
   +0x084 HeapDeCommitFreeBlockThreshold : Uint4B
   +0x088 NumberOfHeaps    : Uint4B
   +0x08c MaximumNumberOfHeaps : Uint4B
   +0x090 ProcessHeaps     : Ptr32 Ptr32 Void
```

Anti-Debugging

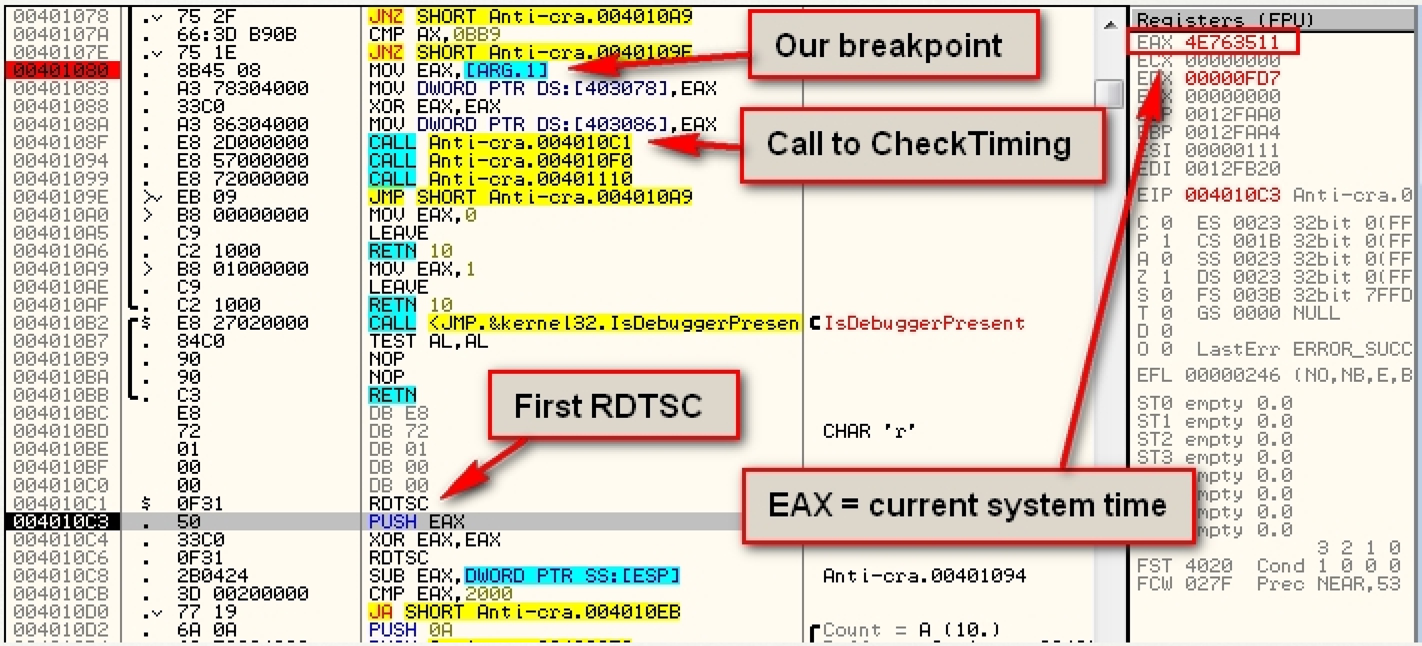
- PEB!NtGlobalFlag

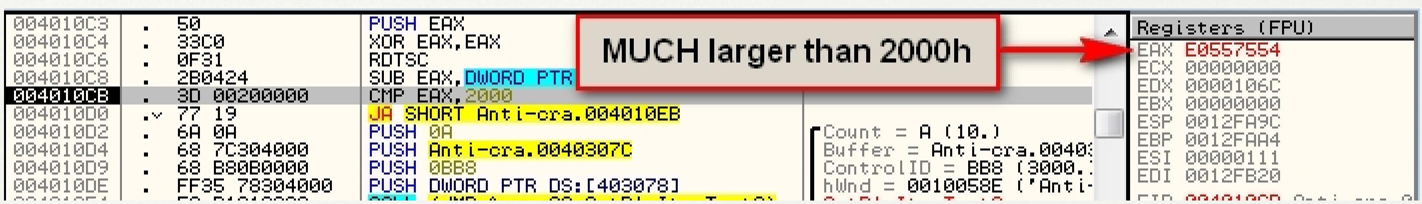
- status = (_NtQueryInformationProcess) (hnd, ProcessBasicInformation, &pPIB, sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);
- value = (pPIB.PebBaseAddress);
- value = value+0x68;
- if (*value == 0x70) {
- MessageBox(NULL, L"Debugger Detected Using PEB!NTGlobalFlag", MessageBox(NULL, L"Debugger Detected Using PEB!NTGlobalFlag", L"Debugger Detected", MB_OK);
- } else {
- MessageBox(NULL, L"No Debugger Detected", L"No Debugger Detected", MB_OK);
- }

Anti-Debugging

- RDTSC is used to retrieve the time stamp counter (number of clocks since boot-up) so this is a time-related trick. When you debug, the distance between those values that are returned in EAX will be higher than those when the program runs without being debugged. So, if there is really a difference you're debugging

Anti-Debugging





Anti-Debugging

```
int main(int argc, char* argv[])
{
    unsigned int time1 = 0;
    unsigned int time2 = 0;
    __asm
    {
        RDTSC
        MOV time1,EAX
        RDTSC
        MOV time2, EAX

    }
    if ((time2 - time1) > 100)
    {
        printf("%s", "VM Detected");
        return 0;
    }
    printf("%s", "VM not present");
    return 0;
}
```

Anti-Debugging

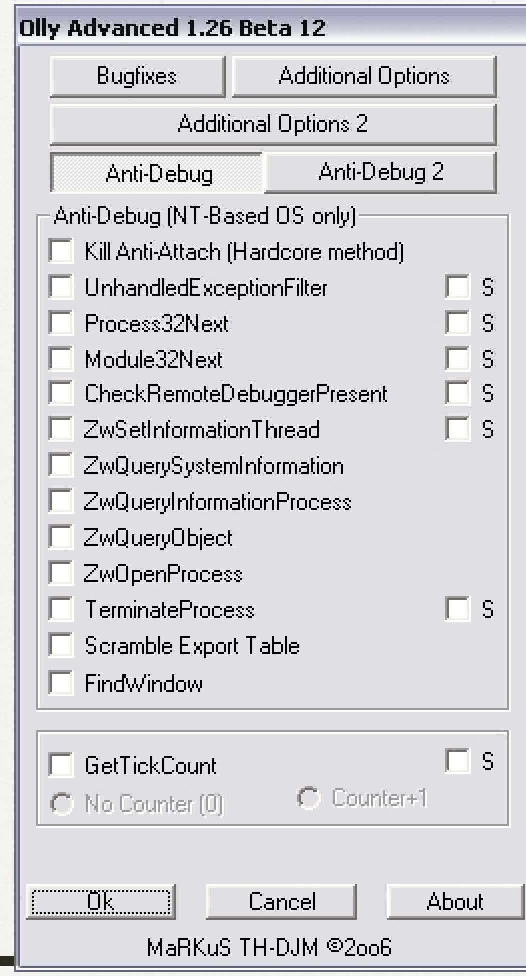
- Find evidence of debugger on system:
  - Registry entries
  - FindWindow API call :
    - HANDLE ollyHandle = NULL;
    - ollyHandle = FindWindow(L"OLLYDBG", 0);
    - if (ollyHandle == NULL) {
    - MessageBox(NULL, L"OllyDbg Not Detected", L"Not Detected", MB_OK);
    - } else {
    - MessageBox(NULL, L"Ollydbg Detected Via OllyDbg FindWindow()", MessageBox(NULL, L"Ollydbg Detected Via OllyDbg FindWindow()",
    - L"OllyDbg Detected", MB_OK);
    - }

Anti-Debugging

- More :
  - Anti-Debugging - A Developers Viewpoint
  - Windows Anti-Debug Reference
  - The “Ultimate”Anti-Debugging Reference

Anti-anti-debugging

- Olly Advanced
- StrongOD
- aadp : http://code.google.com/p/aadp/



Anti-VM

- VM Fingerprints
  - Descriptor Table addresses (IDT, LDT, etc.)
  - Running Processes (eg. VMWare Tools)
  - Registry entries that include "VMWare“
  - loaded modules name
  - Default virtual machine hardware
  - Common VM MAC addresses
  - VMWare specific I/O port
  - Basically, any difference between a VM and a real computer

Anti-VM

```
00428890
00428890 loc_428890:
00428890 push    dword ptr [eax]
00428892 call    hashString_0
00428897 pop     ecx
00428898 cmp     eax, 2FE483F3h  ; vmscsi.sys
0042889D jz      short loc_4288F2

0042889F cmp     eax, 2FD5F8F3h  ; vmhgfs.sys
004288A4 jz      short loc_4288F2

004288A6 cmp     eax, 0CFF129A8h ; vmx_svga.sys
004288AB jz      short loc_4288F2

004288AD cmp     eax, 2FDB60F3h  ; vmxnet.sys
004288B2 jz      short loc_4288F2

004288B4 cmp     eax, 2FFC2FB4h  ; vmmouse.sys
004288B9 jz      short loc_4288F2

004288BB cmp     eax, 2FF91C94h  ; vmdebug.sys
004288C0 jz      short loc_4288F2
```

Anti-VM

```
004288C8
004288C8 loc_4288C8:                ; vmwaretray
004288C8 push    9F408DC4h
004288CD call    enumCheckProcesses ; result in EAX...EAX=0 means passed
004288D2 pop     ecx
004288D3 test    eax, eax
004288D5 jnz     short End
```

```
004288D7 push    9F5784C4h          ; vmwareuser
004288DC call    enumCheckProcesses ; result in EAX...EAX=0 means passed
004288E1 pop     ecx
004288E2 test    eax, eax
004288E4 jnz     short End
```

Anti-VM

```
int main(int argc, char **argv)
{

char lszValue[100];
   HKEY hKey;
   int i=0;
    RegOpenKeyEx (HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Services\\Disk\\Enum", 0L,
KEY_READ , &hKey);

    RegQueryValue(hKey,"0",lszValue,sizeof(lszValue));

    printf("%s", lszValue);
    if (strstr(lszValue, "VMware"))
    {
        printf("Vmware Detected");
    }

     RegCloseKey(hKey);
    return 0;

```

Anti-VM

- Red Pill is an anti-VM technique that executes the sidt instruction to grab the value of the IDTR register. The virtual machine monitor must relocate the guest's IDTR to avoid conflict with the host's IDTR. Since the virtual machine monitor is not notified when the virtual machine runs the sidt instruction, the IDTR for the virtual machine is returned. The Red Pill tests for this discrepancy to detect the usage of VMware.

Anti-VM

- The sgdt and sldt instruction technique for VMware detection is commonly known as No Pill. Unlike Red Pill, No Pill relies on the fact that the LDT structure is assigned to a processor, not an operating system. And because Windows does not normally use the LDT structure, but VMware provides virtual support for it, the table will differ predictably : The LDT location on the host machine will be zero, and on the virtual machine, it will be nonzero. A simple check for zero against the result of the sldt instruction does the trick.

Anti-VM

- ScoopyNG - The VMware Detection Tool
  - ScoopyNG combines the detection tricks of Scoopy Doo and Jerry as well as some new techniques to determine if a current OS is running inside a VMware Virtual Machine (VM) or on a native system.
  - 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-VM

C:\WINDOWS\system32\cmd.exe

```
####################################################
::       ScoopyNG - The VMware Detection Tool      ::
::              Windows version v1.0               ::

[+] Test 1: IDT
IDT base: 0x82c89400
Result  : Native OS


[+] Test 2: LDT
LDT base: 0xdead0000
Result  : Native OS


[+] Test 3: GDT
GDT base: 0x82c89000
Result  : Native OS


[+] Test 4: STR
STR base: 0x28000000
Result  : Native OS


[+] Test 5: VMware "get version" command
Result  : Native OS


[+] Test 6: VMware "get memory size" command
Result  : Native OS


[+] Test 7: VMware emulation mode
Result  : Native OS or VMware without emulation mode
          (enabled acceleration)

::                  tk,  2008                      ::
::               [ www.trapkit.de ]                ::
####################################################
```

```
####################################################
::       ScoopyNG - The VMware Detection Tool      ::
::              Windows version v1.0               ::

[+] Test 1: IDT
IDT base: 0xffc18000
Result  : VMware detected


[+] Test 2: LDT
LDT base: 0xdead4060
Result  : VMware detected


[+] Test 3: GDT
GDT base: 0xffc07000
Result  : VMware detected


[+] Test 4: STR
STR base: 0x00400000
Result  : VMware detected


[+] Test 5: VMware "get version" command
Result  : VMware detected
Version : Workstation


[+] Test 6: VMware "get memory size" command
Result  : VMware detected


[+] Test 7: VMware emulation mode
Result  : Native OS or VMware without emulation mode
          (enabled acceleration)

::                  tk,  2008                      ::
::               [ www.trapkit.de ]                ::
####################################################
```

Anti-VM

- The same with VirtualBox
  - 9 method to detect VirtualBox by waleedassar : http://pastebin.com/RU6A2UuB

Anti-VM

- More :
  - Thwarting Virtual Machine Detection
  - Detecting the Presence of Virtual Machines Using the Local Data Table

Anti-anti-vm

- Hardening your VM
  - Don’t install the VMware tool
  - Change the configuration of your virtual machine by adding the following options to your .vmx file :
    - 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"

Anti-anti-vm

- Patching the code : If you debug the malware and identify some of the specific instructions (e.g. *sidt*, *sgdt*, *sldt*) you can replace the code with NOPs to prevent it.

- More :
  - http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf
  - http://radlab.cs.berkeley.edu/w/upload/3/3d/Detecting_VM_Aware_Malware.pdf
  - http://vrt-blog.snort.org/2009/10/how-does-malware-know-difference.html

Anti-Sandbox

- Logic bombs are particular checks in the program which require certain events to be true in order to execute the malicious payload
  - Checking if something changes on the user desktop
  - checking if the mouse pointer is not moving for a particular time
- Sandbox Overloading
  - Malware flood the sandbox by generating too much worthless behavior data (e.g the sleep call) before executing the real payload. Logging the generated behavior data introduces additional delays and therefore the execution does not reach the real payload
  - Solution : only analysis network traffic, does not capture any system level behavior

Anti-Sandbox

```
#include <windows.h>
#include <stdio.h>

void overloadSandbox()
{
    char Path[20];

    for(int i = 0; i < 100000; ++i)
    {
        sprintf_s(Path, 20, "C:\\test");
        DeleteFile(Path);
    }
}

int main()
{
    overloadSandbox();
    //Real payload
    CreateFile("C:\\payload" , GENERIC_WRITE, FILE_SHARE_WRITE, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    return 0;
}
```

Anti-Sandbox

- Pafish is a demo tool that performs some anti(debugger/VM/sandbox) tricks :
  - https://github.com/a0rtega/pafish

Anti-Anti-Sandbox

Logic bombs :

- By understanding the behavior of the logic bomb code in the analysis report human analysts, we can improve your sandbox

Sandbox Overload :

- In some case, we should only analyses only network traffic and does not capture any system level behavior
- In addition we could also write a signature to detect and blacklist the massive worthless behavior data (e.g the sleep call)

Some tools can help you

- Crowd Detox : plugin for Hex-Rays automatically removes junk code and variables from Hex-Rays function decompilations

- CrowdRE : aims to make it easier for developers to reverse engineer complex applications by working collaboratively with other users

- http://www.crowdstrike.com/community-tools/index.html

Conclusion

- Automated Malware Analysis is good but it can be defeat by new anti-* techniques => we still need manual analysis for advance malwares and update back to AMAs

Thank you !