

MODERN TECHNIQUES TO DEOBFUSCATE AND UEFI/BIOS MALWARE

HITBSecConf2019 - Amsterdam



by Alexandre Borges



Agenda:

- ❖ Few words about anti-reversing
- ❖ METASM
- ❖ Keystone + uEmu
- ❖ MIASM
- ❖ BIOS/UEFI rootkits:
 - ❖ Windows Boot Process
 - ❖ MBR / VBR / IPL / KCS
 - ❖ ELAM / Control Integrity
 - ❖ Secure Boot
 - ❖ BIOS Guard / Boot Guard
 - ❖ UEFI Protections + chipsec

- ✓ **Malware and Security Researcher.**
- ✓ **Speaker at DEFCON USA 2018**
- ✓ **Speaker at DEFCON CHINA 2019**
- ✓ **Speaker at CONFidence Conf. 2019**
- ✓ **Speaker at BSIDES 2018/2017/2016**
- ✓ **Speaker at H2HC 2016/2015**
- ✓ **Speaker at BHACK 2018**
- ✓ **Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics and Rootkits.**
- ✓ **Reviewer member of the The Journal of Digital Forensics, Security and Law.**
- ✓ **Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response**

ANTI-REVERSING

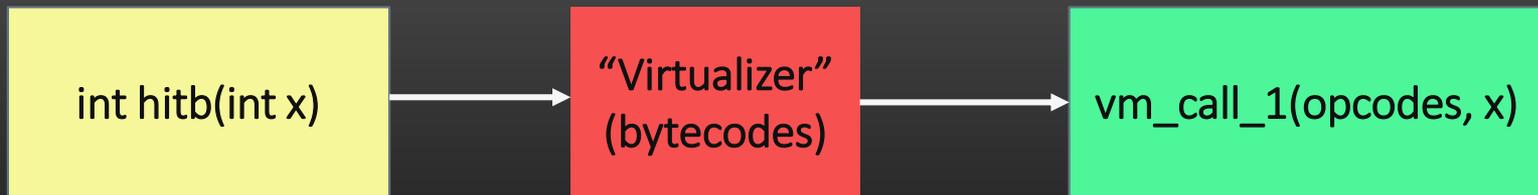
(few words)

- ✓ **Obfuscation** aims to protect software of being reversed, protect intellectual property and, in our case, malicious code too. 😊
- ✓ Honestly, obfuscation **does not really protect the program**, but it is able to make the reverser's life **harder** than usual.
- ✓ Thus, at end, obfuscation **buys time by enforcing reversers to spend resources and time to break a code.**
- ✓ We see **obfuscated code** every single day when analyzing droppers in VBA and PowerShell, so it might seem not to be a big deal.
- ✓ However, **they use very basic obfuscated techniques when they are compared to sophisticated threats.**

- ✓ We can use IDA Pro SDK to write plugins for:
 - ✓ extending the IDA Pro functionalities:
 - ✓ analyzing some code and data flow
 - ✓ automatizing the unpacking process of strange malicious files.
 - ✓ writing a loader to modified MBR structure. 😊
- ✓ Unfortunately, there are packers and protectors such as VMprotect, Themida, Arxan and Agile .NET that use modern obfuscation techniques, so making code reversing very complicated.

- ✓ Most protectors have been used with **64-bit code** (and malware).
- ✓ **Original IAT is removed** from the original code (as usually applied by any packer). However, **IAT from packers like Themida keeps only one function (TlsSetValue())**.
- ✓ Almost all of them provides **string encryption**.
- ✓ They **check the memory integrity**.
- ✓ Thus, it is not possible to dump a clean executable from the memory (using **Volatility**, for example) because **original instructions are not decoded in the memory**.
- ✓ Instructions (x86/x64 code) are **virtualized** and transformed into **virtual machine instructions (RISC instructions)**.
- ✓ **.NET protectors rename classes, methods, fields and external references**.

- ✓ Some packers can use **instruction encryption** on memory as additional memory layer.
- ✓ Obfuscation used by these protectors is **stack based**, so it makes **hard to handle virtualized code statically**.
- ✓ **Virtualized code is polymorphic**, so there **are many virtual machine representations** referring to the same CPU instruction.
- ✓ There are also lots of **fake push instructions**.
- ✓ There are many **dead and useless codes**.
- ✓ There is some **code reordering using many unconditional jumps**.
- ✓ All obfuscators use **code flattening**.



Fetching bytes, decoding them to instructions and dispatching to handlers

❖ Protectors using virtual machines introduces into the obfuscated code:

- ✓ A context switch component, which “transfers” registry and flag information into VM context (virtual machine). The opposite movement is done later from VM machine and native (x86/x64) context (suitable to keep within C structures during unpacking process 😊)
- ✓ This “transformation” from native register to virtualized registers can be one to one, but not always.

✓ Inside of the virtual machine, the cycle is:

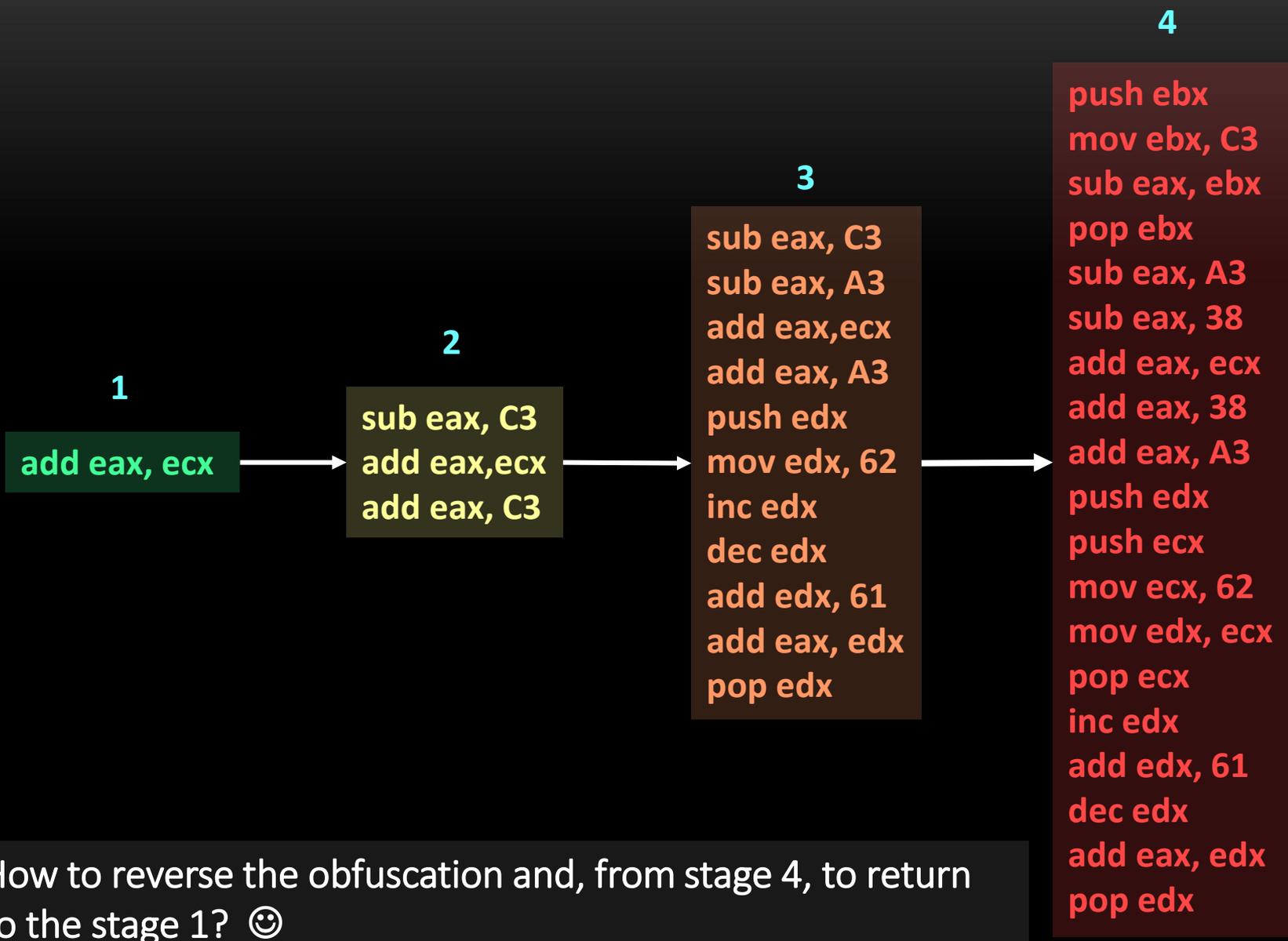
- ✓ fetch instruction
- ✓ decode it
- ✓ find the pointer to instruction and lookup the associate opcode in a handler table
- ✓ call the target handler

- ✓ **Constant unfolding:** technique used by obfuscators to **replace a constant** by a bunch of code that produces the same resulting constant's value.
- ✓ **Pattern-based obfuscation:** exchanging of one instruction by a set of equivalent instructions.
- ✓ Abusing **inline functions**.
- ✓ **Anti-VM techniques:** prevents the malware sample to run inside a VM.
- ✓ **Dead (garbage) code:** this technique is implemented by **inserting codes** whose results will be overwritten in next lines of code or, worse, they won't be used anymore.
- ✓ **Code duplication:** different paths coming into the same destination (**used by virtualization obfuscators**).

- ✓ **Control indirection 1:** call instruction → update the stack pointer → return skipping some junk code after the call instruction (RET x).
- ✓ **Control indirection 2:** malware trigger an exception → registered exception is called → new branch of instructions.
- ✓ **Opaque predicate:** Apparently there is an evaluation (jz / jnz) to take a branch or another one, but the result is always evaluated to true (or false), which means an unconditional jump. Thus, there is a dead branch. Usually, a series of arithmetic / logic tricks are used.
- ✓ **Anti-debugging:** its used as an irritating technique to slow the process analysis.
- ✓ **Polymorphism:** it is produced by using self-modification code (like shellcodes) and by using encrypting resources (similar most malware samples).

- ✓ **Call stack manipulation:** Changes the stack flow by using instruction tricks composed with the ret instruction, making the **real return point hidden**.
- ✓ Is it possible to **deobfuscate virtualized instructions**? Yes, it is possible by:
 - ✓ using **reverse recursive substitution** (similar -- not equal -- to **backtracking feature from Metasm**).
 - ✓ using **symbolic equation system** is another good approach (again.... **Metasm and MIASM!**).
- ✓ There are many good IDA Pro plugins such as **Code Unvirtualizer**, **VMAttack**, **VMSweeper**, and so on, which could be used to handle simple virtualization problems.

METASM + MIASM



How to reverse the obfuscation and, from stage 4, to return to the stage 1? 😊

✓ **METASM** works as disassembler, assembler, debugger, compiler and linker.

✓ Key features:

✓ Written in **Ruby**

✓ C compiler and decompiler

✓ Automatic **backtracking**

✓ Live process manipulation

✓ Supports the following architecture:

✓ **Intel IA32 (16/32/64 bits)**

✓ **PPC**

✓ **MIPS**

✓ Supports the following file format:

✓ **MZ and PE/COFF**

✓ **ELF**

✓ **Mach-O**

✓ **Raw (shellcode)**

✓ `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`

❖ based on metasm.rb file and Bruce Dang code.

```
1 #!/usr/bin/env ruby
2 #
3
4 require "metasm"
5 include Metasm
6
7 coderef = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOB)
8 item:
9     push ebx
10    mov ebx, 0xc3
11    sub eax, ebx
12    pop ebx
13    sub eax, 0xa3
14    sub eax, 0x38
15    add eax, ecx
16    add eax, 0x38
17    add eax, 0xa3
18    push edx
19    push ecx
20    mov ecx, 0x62
21    mov edx, ecx
22    pop ecx
23    inc edx
24    add edx, 0x61
25    dec edx
26    add eax, edx
27    pop edx
28    jmp eax
29 EOB
30
31 addrstart = 0
32 textcode = coderef.init_disassembler
33 textcode.disassemble(addrstart)
34 hitb_di = textcode.di_at(addrstart)
35 hitb = hitb_di.block
36 puts "\n[$] Our HITB 2019 Amsterdam test code:\n "
37 puts hitb.list
```

This is our code from previous slide to be deobfuscated and backtracked. 😊

Starts the disassembler engine and disassemble from the first address (zero).

```

39 hitb.list.each{|aborges|
40   puts "\n[##] #{aborges.instruction}"
41   back = aborges.backtrace_binding()
42   a = back.keys
43   b = back.values
44   c = a.zip(b)
45   puts "HITB 2019 Amsterdam data flow follows below:\n"
46   c.each do |mykey, myvalue|
47     puts " Result:  #{mykey} => #{myvalue}"
48     if aborges.opcode.props[:setip]
49       puts "\nHITB 2019 Amsterdam control flow follows below:\n"
50       puts " * #{textcode.get_xrefs_x(aborges)}"
51     end
52   end
53 end
54 }
55 }
56
57 addrstart2 = 0
58 textcode2 = coderef.init_disassembler
59 textcode2.disassemble(addrstart2)
60
61 dd = textcode2.block_at(addrstart2)
62 final = textcode2.get_xrefs_x(dd.list.last).first
63 puts "\n[+] final output: #{final}"

```

Starts the backtrace engine and walk back in the code.

Determines which is the final instruction to walk back from there. 😊

```

65 values = textcode2.backtrace(final, dd.list.last.address, {:log => bac
klog = [] , :include_start => true})
66 backlog.each{|item|
67     case type = item.first
68     when :start
69         item, expression, address = item
70         puts "[start] Here is the sequence of expression evalu
ations   #{expression} from 0x#{address.to_s(16)}\n"
71
72     when :di
73         item, new, old, instruction = item
74         puts "[new update] instruction #{instruction},\n --> u
pdating expression once again old #{old} new #{new}\n"
75     end
76 }
77
78 effective = backlog.select{|y| y.first==:di}.map{|y| y[3]}.reverse
79 puts "\nThe effective instructions are:\n\n"
80 puts effective
81
82

```

Logs all the "backtracked" instructions.

Shows the effective instructions, which might alter the final result.

```
root@kali:~/programs/metasm# ./hitb2019.rb
```

```
[$] Our HITB 2019 Amsterdam 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
```

```
##] push ebx
```

```
HITB 2019 Amsterdam data flow follows below:
```

```
Result: esp => esp-4
```

```
Result: dword ptr [esp] => ebx
```

```
##] mov ebx, 0c3h
```

```
HITB 2019 Amsterdam data flow follows below:
```

```
Result: ebx => 0c3h
```

```
[##] push ebx
```

```
HITB 2019 Amsterdam data flow follows below:
```

```
Result: esp => esp-4
```

```
Result: dword ptr [esp] => ebx
```

```
[##] mov ebx, 0c3h
```

```
HITB 2019 Amsterdam data flow follows below:
```

```
Result: ebx => 0c3h
```

```
[##] sub eax, ebx
```

```
HITB 2019 Amsterdam data flow follows below:
```

```
Result: eax => eax-ebx
```

```
Result: eflag_z => (((eax&0xffffffff)-(ebx&0xffffffff))&0xffffffff)==0
```

```
Result: eflag_s => (((eax&0xffffffff)-(ebx&0xffffffff))&0xffffffff)>>1fh)!=0
```

```
Result: eflag_c => (eax&0xffffffff)<(ebx&0xffffffff)
```

```
Result: eflag_o => (((eax&0xffffffff)>>1fh)!=0)==(!(((ebx&0xffffffff)>>1fh)!=0))&&(((eax&0xffffffff)>>1fh)!=0)!=(((eax&0xffffffff)-(ebx&0xffffffff))&0xffffffff)>>1fh)!=0))
```

```
[##] pop ebx
```

```
HITB 2019 Amsterdam data flow follows below:
```

```
Result: esp => esp+4
```

```
Result: ebx => dword ptr [esp]
```

```
[##] sub eax, 0a3h
```

```
HITB 2019 Amsterdam data flow follows below:
```

```
Result: eax => eax-0a3h
```

```
Result: eflag_z => (((eax&0xffffffff)-((0a3h)&0xffffffff))&0xffffffff)==0
```

```
Result: eflag_s => (((eax&0xffffffff)-((0a3h)&0xffffffff))&0xffffffff)>>1fh)!=0
```

```
Result: eflag_c => (eax&0xffffffff)<((0a3h)&0xffffffff)
```

```
Result: eflag_o => (((eax&0xffffffff)>>1fh)!=0)==(!(((0a3h)&0xffffffff)>>1fh)!=0))&&(((eax&0xffffffff)>>1fh)!=0)!=(((eax&0xffffffff)-((0a3h)&0xffffffff))&0xffffffff)>>1fh)!=0))
```

[+] final output: eax

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

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
```

These are the effective instructions. 😊

```
root@kali:~/programs/metasm#
```

- ✓ **Emulation** is always an excellent method to solve practical reverse engineering problems and , fortunately, we have the **uEmu** and also could use the **Keystone Engine** assembler and **Capstone Engine** disassembler. 😊
- ✓ **Keystone Engine** acts an **assembler** engine and:
 - ✓ Supports **x86, Mips, Arm** and many other architectures.
 - ✓ It is **implemented in C/C++** and has bindings to **Python, Ruby, Powershell and C#** (among other languages).
- ✓ Installing **Keystone**:
 - ✓ `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`
 - ✓ `root@kali:~/programs/keystone-0.9.1# mkdir build ; 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`

```
#include <stdio.h>
#include <keystone/keystone.h>

#define HITB2019AMS "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;
    }

    if (ks_asm(keyeng, HITB2019AMS, 0, &encode, &size, &count)) {
        printf("ERROR: A fail has occurred 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]);
        }
    }

    ks_free(encode);
    ks_close(keyeng);

    return 0;
}
```

instructions from the original obfuscated code

Creating a keystone engine

Assembling our instructions using keystone engine.

Freeing memory and closing engine.

```
root@kali:~/programs/hitb2019ams# more Makefile
```

```
.PHONY: all clean
```

```
KEYSTONE_LDFLAGS = -lkeystone -lstdc++ -lm
```

```
all:
```

```
    ${CC} -o hitb2019ams hitb2019ams.c ${KEYSTONE_LDFLAGS}
```

```
clean:
```

```
    rm -rf *.o hitb2019ams
```

```
root@kali:~/programs/hitb2019ams#
```

```
root@kali:~/programs/hitb2019ams# make
```

```
cc -o hitb2019ams hitb2019ams.c -lkeystone -lstdc++ -lm
```

```
root@kali:~/programs/hitb2019ams#
```

```
root@kali:~/programs/hitb2019ams# ./hitb2019ams
```

```
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/hitb2019ams#
```

```
root@kali:~/programs/hitb2019ams#
```

```
root@kali:~/programs/hitb2019ams# ./hitb2019ams | xxd -r -p - > hitb2019ams.bin
```

```
root@kali:~/programs/hitb2019ams#
```

```
root@kali:~/programs/hitb2019ams# hexdump -C hitb2019ams.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
```

```
root@kali:~/programs/hitb2019ams#
```

```
seg000:00000000 ; File Name : C:\UMs\hitb2019ams.bin
seg000:00000000 ; Format : Binary file
seg000:00000000 ; Base Address: 0000h Range: 0000h - 002Dh Loaded length: 002Dh
```

```
seg000:00000000 .686p
seg000:00000000 .mmx
seg000:00000000 .model flat
```

```
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,
```

- 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
- seg000:00000018 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

IDA Pro confirms our disassembly task. 😊

```

#include <stdio.h>
#include <inttypes.h>
#include <capstone/capstone.h>

#define CODE "\x53\xbb\xb9\x00\x00\x00\x29\xd8\x5b\x83\xe8\x55\x83\xe8\x32\x01\xc8\x83\x
xc0\x50\x83\xc0\x37\x52\x51\xb9\x49\x00\x00\x00\x89\xca\x59\x42\x83\xc2\x70\x4a\x01\xd0
\x5a"

int main(void)
{
    csh cs_handle;
    cs_insn *instruction;
    size_t count;

    if (cs_open(CS_ARCH_X86, CS_MODE_32, &cs_handle) != CS_ERR_OK)
        return -1;
    count = cs_disasm(cs_handle, CODE, sizeof(CODE)-1, 0x0001, 0, &instruction);
    if (count > 0) {
        size_t j;
        for (j = 0; j < count; j++) {
            printf("0x%"PRIx32":\t%s\t\t%s\n", instruction[j].address, inst
ruction[j].mnemonic, instruction[j].op_str);
        }
        cs_free(instruction, count);
    } else
        printf("Error: It's happened an error during the disassembling!\n");

    cs_close(&cs_handle);

    return 0;
}

```

To install Capstone: `apt-get install libcapstone3 libcapstone-dev` 😊

```
root@kali:~/programs/hitb2019ams/capstone# more Makefile
.PHONY: all clean
```

```
CAPSTONE_LDFLAGS = -lcapstone -lstdc++ -lm
```

```
all:
    ${CC} -o hitb2019ams_rev hitb2019ams_rev.c ${CAPSTONE_LDFLAGS}
```

```
clean:
    rm -rf *.o hitb2019ams_rev
```

```
root@kali:~/programs/hitb2019ams/capstone#
```

```
root@kali:~/programs/hitb2019ams/capstone# make
```

```
cc -o hitb2019ams_rev hitb2019ams_rev.c -lcapstone -lstdc++ -lm
```

```
root@kali:~/programs/hitb2019ams/capstone#
```

```
root@kali:~/programs/hitb2019ams/capstone# ./hitb2019ams_rev
```

```
0x1:  push      ebx
0x2:  mov       ebx, 0xb9
0x7:  sub      eax, ebx
0x9:  pop      ebx
0xa:  sub      eax, 0x55
0xd:  sub      eax, 0x32
0x10: add      eax, ecx
0x12: add      eax, 0x50
0x15: add      eax, 0x37
0x18: push     edx
0x19: push     ecx
0x1a: mov     ecx, 0x49
0x1f: mov     edx, ecx
0x21: pop     ecx
0x22: inc     edx
0x23: add     edx, 0x70
0x26: dec     edx
0x27: add     eax, edx
0x29: pop     edx
```

Original code disassembled
by Capstone. 😊

```
root@kali:~/programs/hitb2019ams/capstone#
```

uEmu CPU Context

Register	Value
eax	0x1
ebx	0x0
ecx	0x6
edx	0x0
esi	0x0
edi	0x0
ebp	0x0
esp	0x0
eip	0x0
sp	0x0

set up before running uEmu

OK Cancel Search Help

Line 6 of 10

- ✓ Download uEmu from <https://github.com/alexhude/uEmu>
- ✓ Install Unicorn: `pip install unicorn`.
- ✓ Load uEmu in IDA using **ALT+F7** hot key.
- ✓ **Right click** the code and choose the uEmu sub-menu.

uEmu CPU Context

CPU context at [0x2C: pop edx]

eax: 0x00000007	edi: 0x00000000
ebx: 0x00000000	ebp: 0x00000000
ecx: 0x00000006	esp: 0xFFFFF0FC
edx: 0x000000C3	eip: 0x0000002C
esi: 0x00000000	sp: 0x0000FFFC

This result confirms our previous conclusion.

Output window

```

-----
Python 2.7.15 (v2.7.15:ca079a3ea3, Apr 30 2018, 16:22:17) [MSC v.1500 32 bit (Intel)]
IDAPython v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
-----

The initial autoanalysis has been finished.
[uEmu]: Init plugin
[uEmu]: Run plugin
[uEmu]: CPU arch set to [ x86 ]
[uEmu]: Emulation started
[uEmu]: Mapping segments...
[uEmu]: * seg [0:2D]
[uEmu]:   map [0:FFF] -> [0:FFF]
[uEmu]:   cpy [0:2C]
[uEmu]: † <M> Missing memory at 0xfffffff0c, data size = 4, data value = 0x0
[uEmu]:   map [FFFFFFF0C:FFFFFFF0C] -> [FFFFFF000:FFFFFFF0C]
[uEmu]: Breakpoint reached at 0x2C : pop edx
  
```

Python

- ✓ **MIASM** is one of most impressive framework for reverse engineering, which is able to **analyze, generate and modify** several different types of programs.
- ✓ **MIASM** supports **assembling and disassembling** programs from different platforms such as **ARM, x86, MIPS** and so on, and it is also able to **emulate code by using JIT**.
- ✓ Therefore, **MIASM** is excellent to de-obfuscation.
- ✓ **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`
 - ✓ `./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
- ✓ (testing MIASM) root@kali:~/programs# python /root/programs/miasm/example/disasm/full.py -m x86_32 /root/programs/shellcode

INFO : Load binary

INFO : ok

INFO : import machine...

INFO : ok

INFO : func ok 0000000000001070 (0)

INFO : generate graph file

INFO : generate intervals

[0x1070 0x10A2]

INFO : total lines 0

- ✓ (testing MIASM) xdot graph_execflow.dot



```
graph TD
    start["_start"] --> loc1080["loc_1080"]
    loc1080 --> loc10a1["loc_10a1"]
```

```
00001070 XOR     EBP, EBP
00001072 POP     ESI
00001073 MOV     ECX, ESP
00001075 AND     ESP, 0xFFFFFFFF0
00001078 PUSH    EAX
00001079 PUSH    ESP
0000107A PUSH    EDX
0000107B CALL   loc_10a2
```

```
00001080 ADD     EBX, 0x2F80
00001086 LEA    EAX, DWORD PTR [EBX + 0xFFFFD280]
0000108C PUSH    EAX
0000108D LEA    EAX, DWORD PTR [EBX + 0xFFFFD220]
00001093 PUSH    EAX
00001094 PUSH    ECX
00001095 PUSH    ESI
00001096 PUSH    DWORD PTR [EBX + 0xFFFFFFFF8]
0000109C CALL   loc_1050
```

```
000010A1 HLT
```

```
1 from miasm2.analysis.binary import Container
2 from miasm2.analysis.machine import Machine
3 from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE
4
5 with open("hitb2019ams.bin") as fdesc:
6     cont=Container.from_stream(fdesc)
7
8 machine=Machine('x86_32')
9 mdis=machine.dis_engine(cont.bin_stream)
10 ourblocks = mdis.dis_multiblock(0)
11 for block in ourblocks:
12     print block
13 jitter = machine.jitter("llvm")
14 jitter.init_stack()
15 s = open("hitb2019ams.bin").read()
16 run_addr = 0x40000000
17 jitter.cpu.EAX=1
18 jitter.cpu.ECX=6
19 jitter.vm.add_memory_page(run_addr, PAGE_READ | PAGE_WRITE, s)
20 def code_sentinelle(jitter):
21     jitter.run = False
22     jitter.pc = 0
23     return True
24 jitter.add_breakpoint(0x4000002c, code_sentinelle)
25 jitter.push_uint32_t(0x4000002c)
26 jitter.jit.log_regs = True
27 jitter.jit.log_mn = True
28 jitter.init_run(run_addr)
29 jitter.continue_run()
30
31 open('hitb2019ams_cfg.dot', 'w').write(ourblocks.dot())
```

Opens our file. The Container provides the byte source to the disasm engine.

Instantiates the assemble engine using the x86 32-bits architecture.

Runs the recursive transversal disassembling since beginning.

Set "llvm" as JIT engine to emulate and initialize the stack.

Set the virtual start address, register values and memory protection.

Adds a breakpoint at the last line of code.

Run the emulation.

Generates a dot graph.

```
root@kali:~/programs/hitb2019ams# 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
```

```
->      c_next:loc_000000000000002D:0x0000002d
```

```
loc_000000000000002D:0x0000002d
```

```

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 FFFFFFF3E 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 FFFFFFF3E 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 FFFFFFFE9B 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

ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000025 INC EDX
EAX FFFFFFF44 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 FFFFFFF44 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 FFFFFFF44 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

```

```
loc_0000000000000000
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
```

Our proposed code. 😊

```
loc_000000000000002D
IOError
```

```
root@kali:~/programs/hitb2019ams# python
Python 2.7.16 (default, Apr 6 2019, 01:42:57)
[GCC 8.3.0] 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
>>> with open("hitb2019ams.bin") as fdesc:
...     cont=Container.from_stream(fdesc)
...
>>> hitbmach=Machine('x86_32')
>>> hitbdis=hitbmach.dis_engine(cont.bin_stream)
>>> myblocks = hitbdis.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
>>> sym = hitbmach.ira( )
>>> for block in myblocks:
...     sym.add_block(block)
...
[<miasm2.ir.ir.IRBlock object at 0x7f25bb863820>]
[]
>>> from miasm2.ir.symbexec import SymbolicExecutionEngine
>>> symb = SymbolicExecutionEngine(sym, hitbmach.mn.regs.regs_init)
>>> symbolic_pc = symb.run_at(0, step=True)
```

Get the IRA converter.

Initialize and run the Symbolic Execution Engine.

Instr **PUSH** **EBX**

Assignblk:

ESP = ESP + -0x4

@32[ESP + -0x4] = EBX

ESP = ESP_init + 0xFFFFFFFFFC

@32[ESP_init + 0xFFFFFFFFFC] = EBX_init

Instr **MOV** **EBX, 0xC3**

Assignblk:

EBX = 0xC3

ESP = ESP_init + 0xFFFFFFFFFC

EBX = 0xC3

@32[ESP_init + 0xFFFFFFFFFC] = 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

```

EAX          = EAX_init + ECX_init
cf           = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFF
FFF3D)) & ((EAX_init + ECX_init + 0xFFFFFFFF3D) ^ 0xFFFFFFFF3C)) ^ (EAX_init + E
CX_init) ^ (EAX_init + ECX_init + 0xFFFFFFFF3D) ^ 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 + 0xFFFF
F3D) ^ 0xC3)[4:5]
of          = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFF
FF3D)) & ((EAX_init + ECX_init + 0xFFFFFFFF3D) ^ 0xFFFFFFFF3C))[31:32]
nf          = (EAX_init + ECX_init)[31:32]
@32[ESP_init + 0xFFFFFFFF8] = ECX_init
@32[ESP_init + 0xFFFFFFFFC] = EDX_init

```

```

Instr POP      EDX
Assignblk:
IRDst = loc_000000000000002D:0x0000002d

```

```

EAX          = EAX_init + ECX_init
cf           = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFF
FFF3D)) & ((EAX_init + ECX_init + 0xFFFFFFFF3D) ^ 0xFFFFFFFF3C)) ^ (EAX_init + E
CX_init) ^ (EAX_init + ECX_init + 0xFFFFFFFF3D) ^ 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 + 0xFFFF
F3D) ^ 0xC3)[4:5]
IRDst       = 0x2D
of          = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFF
FF3D)) & ((EAX_init + ECX_init + 0xFFFFFFFF3D) ^ 0xFFFFFFFF3C))[31:32]
nf          = (EAX_init + ECX_init)[31:32]
@32[ESP_init + 0xFFFFFFFF8] = ECX_init
@32[ESP_init + 0xFFFFFFFFC] = EDX_init

```

BIOS/UEFI THREATS

- ✓ Since Windows Vista, the **main protection against rootkits** have been the **KCS (Kernel-mode Code Signing Policy)** that **prevents any unsigned kernel module** to be loaded.
- ✓ KCS forces **all integrity checks on drivers started on boot (PnP/non-PnP) in x64 systems.**
- ✓ In a general way, **rootkits** could try to bypass the KCS:
 - ✓ **disabling it by changing BCD variables** to put the system in **testsigning mode.**
 - ✓ **exploiting some vulnerability** to modify the boot process.
 - ✓ **using a driver with valid certificate from third party companies** to attack the system.
 - ✓ **disabling the Secure Boot**, which forces **BIOS to verify UEFI and system boot files.**

- ✓ **Code Integrity** can be only disabled by changing BCD variables whether the Secure Boot is disabled.
- ✓ Unfortunately, BIOS/UEFI threats attacks earlier boot stages, before KCS starts running, so KCS is not effective against bootkits.
- ✓ Therefore, the malware's goal is to attack any point before common defenses start to compromise the boot process.
- ✓ Fortunately, the Code Integrity in Windows 8 and later are not controlled by only one variable (`nt!g_CiEnabled`) and there are several other "control variables".
- ✓ As the KCS is not able to fight against bootkits, which load before any system/kernel protection starts, so Secure Boot could help us because:
 - ✓ it checks the integrity of Windows boot files and UEFI components.
 - ✓ it checks the bootloader's integrity.

- ✓ On Windows 10, the protection against the kernel and Windows boot components were improved through the Virtual Secure Mode.
- ✓ VSM (Virtual Secure Mode) provides an isolation for the Windows kernel and critical system modules from other components by using virtualization extensions of the CPU.
- ✓ Therefore, non-critical drivers are not able to disable code integrity because they run in separated containers. 😊
- ✓ As we already know, VSM is composed by:
 - ✓ Local Security Authority (to keep processes based on LSASS working)
 - ✓ Kernel Mode Code Integrity (KMCI)
 - ✓ Hypervisor Code Integrity

- ✓ Finally, we can talk about the **Device Guard** that is composed by:
 - ✓ **Configurable Code Integrity (CCI)**: assure that only **trusted code** runs from the **boot loader**.
 - ✓ **VSM Protected Code Integrity**: represents the **KMCI (Kernel Mode Code Integrity)** and **Hypervisor Code Integrity (HVCI)** in the VSM.
 - ✓ **Platform and UEFI Secure Boot**: protects the **UEFI and boot components** by using **digital signature**.
- ✓ To use the advantages and run on systems that **Device Guard** is active:
 - ✓ driver **can't load data as executable code**.
 - ✓ driver **can't alter anything on the system memory**.
 - ✓ **allocated pages can't be executable and writable at same time**.
- ✓ Likely, **many malware samples don't follow these recommendation**, so they **don't work on systems that Device Guard is on**.

- ✓ Windows offers other protection options to protect against malware such as **ELAM (Early Launch Anti-Malware)** to prevent malicious and unauthorized code to execute in the kernel land.
- ✓ There are many interesting aspects about **ELAM**:
 - ✓ It is based on **callback methods**, which monitors drivers and registries.
 - ✓ ELAM classifies **drivers in good, bad and unknown**.
 - ✓ Its decisions are based on **image's name, hash, registry location and certificate issuer/publisher**.
 - ✓ There are some possible values used in the **ELAM policy**, but the default one (**PNP_INITIALIZE_BAD_CRITICAL_DRIVERS**) is suitable for most sceneries because it allows to load bad critical drivers, but not bad non-critical drivers. 😊

- ✓ At **winload.exe execution**, ELAM can not access any binary on disk because the drivers to access the disk is not ready yet.
- ✓ ELAM is excellent against rootkit, but it is not appropriate against **bootkits**, which usually **load before winload.exe** executing (Windows executable that loads ELAM).

- ✓ In legacy systems, **bootkits** could attack:
 - ✓ **MBR**: they compromising either **MBR boot code or partition table** (located at 0x1be).
 - ✓ **VBR**: as VBR (Volume Boot Record) **holds the information about the filesystem's type**, so the idea is to **compromise BIOS parameter block (BPB)** to **change the IPL loading process (next stage) or even executing a malicious code**.
 - ✓ The target field to attack is the **"Hidden sectors" field, which provides the IPL location, in the BPB**.
 - ✓ The **malicious code** could be loaded from a **hidden and encrypted file system**. 😊
 - ✓ After the malicious code being execute, so the **"real IPL"** is loaded.

- ✓ **IPL**: the IPL (Initial Program Loader) holds necessary bootstrap code to locate the OS loader.
- ✓ Thus:
 - ✓ Compromising the IPL might cause the execution of a malicious code instead of loading the bootmgr module.
 - ✓ A malicious IPL could load a malicious kernel driver during the booting process.
 - ✓ Some malicious IPL codes are polymorphic. Take care. 😊

Offset	Title	Value
0	JMP instruction	EB 52 90
3	File system ID	NTFS
B	Bytes per sector	512
D	Sectors per cluster	8
E	Reserved sectors	0
10	(always zero)	00 00 00
13	(unused)	00 00
15	Media descriptor	F8
16	(unused)	00 00
18	Sectors per track	63
1A	Heads	255
1C	Hidden sectors	206,848

BIOS_PARAMETER_
_BLOCK_NTFS

- ✓ Changing the “Hidden sectors” field could cause loading a malicious code instead of executing the IPL.

- ✓ Once the **bootmgr module** is loaded, so the malware's goal is to **circumvent the code integrity verification**. 😊
- ✓ Furthermore, the **bootmgr** and **windload.exe** have a central responsibility in **perform the transition between real mode to protect mode**.
- ✓ Both executables are critical for bootkits because they need to **keep the control of the boot process during this transition**.
- ✓ Once the the **code integrity checking has been disabled**, it is possible to **replace important boot components such as kdcom.dll** by a malicious one.

✓ BIOS disk services provides several operations:

✓ extended read (0x42)

✓ extended write (0x43)

✓ extended get driver parameters (0x48)

✓ Subverting (hooking) INT 13h handle (including reading and writing operation from disk) it is one of the best way to compromise:

✓ the bootmgr

✓ winload.exe

✓ the kernel.

✓ Of course, it is pretty easy to disassemble a MBR in IDA Pro:

✓ `dd.exe -v if=\\.\PHYSICALDRIVE0 of=mbr.bin bs=512 count=1`

✓ Set the offset to `0x7c00` and disassemble it as `16-bit code`.

```
seg000:7C00 loc_7C00: ; CODE XREF: seg000:7D2A↓J
seg000:7C00 xor ax, ax
seg000:7C02 mov ss, ax
seg000:7C04 mov sp, 7C00h
seg000:7C07 mov es, ax
seg000:7C09 mov ds, ax
seg000:7C0B mov si, 7C00h
seg000:7C0E mov di, 600h
seg000:7C11 mov cx, 200h
seg000:7C14 cld
seg000:7C15 rep movsb
seg000:7C17 push ax
seg000:7C18 push 61Ch
seg000:7C1B retf
-----
seg000:7C1C sti
seg000:7C1D mov cx, 4
seg000:7C20 mov bp, 7BEh
seg000:7C23 loc_7C23: ; CODE XREF: seg000:7C30↓j
seg000:7C23 cmp byte ptr [bp+0], 0
seg000:7C27 jl short loc_7C34
seg000:7C29 jnz loc_7D3B
seg000:7C2D add bp, 10h
seg000:7C30 loop loc_7C23
seg000:7C32 int 18h ; TRANSFER TO ROM BASIC
; causes transfer to ROM-based BASIC (IBM-PC)
; often reboots a compatible; often has no effect at all
seg000:7C34 loc_7C34: ; CODE XREF: seg000:7C27↑j
; seg000:7CAE↓j
seg000:7C34 mov [bp+0], dl
seg000:7C37 push bp
seg000:7C38 mov byte ptr [bp+11h], 5
seg000:7C3C mov byte ptr [bp+10h], 0
seg000:7C40 mov ah, 41h
```

Clean MBR. 😊

```

seg000:7C00 ; File Name      : C:\UMs\mbr_infected.bin
seg000:7C00 ; Format          : Binary file
seg000:7C00 ; Base Address: 0000h Range: 7C00h - 7E00h Loaded length: 0200h
seg000:7C00
seg000:7C00      .686p
seg000:7C00      .mmx
seg000:7C00      .model flat

```

```

seg000:7C00 ; =====
seg000:7C00 ; Segment type: Pure code
seg000:7C00 seg000      segment byte public 'CODE' use16
seg000:7C00      assume cs:seg000
seg000:7C00      ;org 7C00h
seg000:7C00      assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing

```

- seg000:7C00
- seg000:7C01
- seg000:7C04
- seg000:7C06
- seg000:7C08
- seg000:7C0A
- seg000:7C0D
- seg000:7C0E
- seg000:7C12
- seg000:7C18
- seg000:7C1E
- seg000:7C21
- seg000:7C21
- seg000:7C24
- seg000:7C26
- seg000:7C2A
- seg000:7C2C
- seg000:7C30
- seg000:7C35
- seg000:7C35
- seg000:7C35
- seg000:7C36
- seg000:7C36

```

cli
xor     eax, eax
mov     ss, ax
mov     es, ax
mov     ds, ax
mov     sp, 7C00h
sti
mov     ds:byte_7C93, d1
mov     eax, 20h
mov     ebx, 22h
mov     cx, 8000h
loc_7C21:
call    sub_7C38 ; CODE XREF: seg000:7C2A↓j
dec     eax
cmp     eax, 0
jnz     short loc_7C21
mov     eax, ds:8000h
jmp     far ptr 0:8000h
loc_7C35:
hlt
jmp     short loc_7C35 ; CODE XREF: seg000:7C36↓j

```

Infected MBR. 😊

- ✓ C:\> "C:\Program Files (x86)\VMware\VMware Workstation\vmware-vdiskmanager.exe" -r Windows_7_x86-cl2-000002.vmdk -t 0 infected.vmdk
- ✓ root@kali:~# qemu-img convert -f vmdk -O raw infected.vmdk infected.raw
- ✓ root@kali:~# dd if=infected.raw of=mbr_infected.bin bs=512 count=1
- ✓ root@kali:~# file mbr_infected.bin
br_infected.bin: DOS/MBR boot sector
- ✓ Install Bochs and create a bochsrc file pointing to the converted image above:

```
romimage: file= "C:\Program Files (x86)\Bochs-2.6.9\BIOS-bochs-latest"  
vgaromimage: file= "C:\Program Files (x86)\Bochs-2.6.9\VGABIOS-lgpl-latest"  
megs: 32  
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14  
ata0-master: type=disk, path="C:\VMs\infected.raw", mode=flat, cylinders=1024, heads=16, spt=63  
boot: disk  
vga: extension=vbe  
mouse: enabled=0  
log: nul  
logprefix: %t%e%d  
panic: action=fatal  
error: action=report  
info: action=report  
debug: action=ignore  
# display_library: win32, options="gui_debug"
```

```
C:\Program Files (x86)\Bochs-2.6.9>bochsdbg.exe -q -f bochsrc
```

```
=====
Bochs x86 Emulator 2.6.9
Built from SVN snapshot on April 9, 2017
Compiled on Apr 9 2017 at 09:49:25
=====
```

```
000000000000i[      ] reading configuration from bochsrc
000000000000i[      ] installing win32 module as the Bochs GUI
000000000000i[      ] using log file nul
```

```
Next at t=0
```

```
(0) [0x0000fffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b          ; ea5be000f0
```

```
<bochs:1> lb 0x7c00
```

```
<bochs:2> info break
```

```
Num Type          Disp Enb Address
  1 lbreakpoint   keep y  0x0000000000007c00
```

```
<bochs:3> c
```

```
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
```

```
Next at t=17404827
```

```
(0) [0x000000007c00] 0000:7c00 (unk. ctxt): cli                    ; fa
```

```
<bochs:4> u /8
```

```
00007c00: (      ): cli                    ; fa
00007c01: (      ): xor eax, eax             ; 6631c0
00007c04: (      ): mov ss, ax              ; 8ed0
00007c06: (      ): mov es, ax              ; 8ec0
00007c08: (      ): mov ds, ax              ; 8ed8
00007c0a: (      ): mov sp, 0x7c00          ; bc007c
00007c0d: (      ): sti                     ; fb
00007c0e: (      ): mov byte ptr ds:0x7c93, dl ; 8816937c
```

```
<bochs:5> s
```

```
Next at t=17404828
```

```
(0) [0x000000007c01] 0000:7c01 (unk. ctxt): xor eax, eax                ; 6631c0
```

```
<bochs:6> s
```

```
Next at t=17404829
```

Debug View Structures Enums

IDA View-EIP

```

BOOT_SECTOR:7C00 public start
BOOT_SECTOR:7C00 start proc near
EIP> BOOT_SECTOR:7C00 cli
    BOOT_SECTOR:7C01 xor     eax, eax
    BOOT_SECTOR:7C04 mov     ss, ax
    BOOT_SECTOR:7C06 assume  ss:MEMORY
    BOOT_SECTOR:7C06 mov     es, ax
    BOOT_SECTOR:7C08 assume  es:MEMORY
    BOOT_SECTOR:7C08 mov     ds, ax
    BOOT_SECTOR:7C0A assume  ds:MEMORY
    BOOT_SECTOR:7C0A mov     sp, 7C00h
    BOOT_SECTOR:7C0D sti
    BOOT_SECTOR:7C0E mov     byte_7C93, dl
    BOOT_SECTOR:7C12 mov     eax, 20h
    BOOT_SECTOR:7C18 mov     ebx, 22h
    BOOT_SECTOR:7C1E mov     cx, 8000h
    BOOT_SECTOR:7C21
    BOOT_SECTOR:7C21 loc_7C21: ; CODE XREF: sta
    BOOT_SECTOR:7C21 call    sub_7C38
    BOOT_SECTOR:7C24 dec     eax
    BOOT_SECTOR:7C26 cmp     eax, 0
    BOOT_SECTOR:7C2A jnz     short loc_7C21
    BOOT_SECTOR:7C2C mov     eax, dword_8000
    BOOT_SECTOR:7C30 jmp     far ptr dword_8000
    BOOT_SECTOR:7C30 start endp
    BOOT_SECTOR:7C30 ; -----
    BOOT_SECTOR:7C35 db     0F4h, 0EBh, 0FDh
    
```

00000000 | 00007C00: start | (Synchronized with EIP)

General registers

```

EAX 0000AA55 ↪ BOOT_SECTOR:AA55
EBX 00000000 ↪ IUTABLE:0000
ECX 00090000 ↪ debug002:90000
EDX 00000080 ↪ IUTABLE:0080
ESI 000E0000 ↪ ROMEXT:10000
EDI 0000FFAC ↪ debug002:FFAC
EBP 00000000 ↪ IUTABLE:0000
ESP 0000FFD6 ↪ debug002:FFD6
EIP 00007C00 ↪ start
EFL 00000082
    
```

Modules

```

Path
BOCHS_DISKIMAGE_LDR
    
```

Threads

Decimal	Hex	State
1	1	Ready

Infected MBR
being debugged 😊

Hex View-1 Stack view

```

7C00 FA 66 31 C0 8E D0 8E C0 8E D8 BC 00 7C FB 88 16 *f1+ÄdÄ+ÄI+.|'ê.
7C10 93 7C 66 B8 20 00 00 00 66 BB 22 00 00 00 B9 00 ô|f@*...f+''...|
7C20 80 E8 14 00 66 48 66 83 F8 00 75 F5 66 A1 00 80 Çp..fHfâ°.u$fi.Ç
    
```

0000 F000FF53 MEMORY:F000FF53

0004 F000FF53 MEMORY:F000FF53

0008 F000FF53 MEMORY:F000FF53

UNKNOWN | 00000000: IVTABLE:0000 | (Synchronized with ESP)

Output window

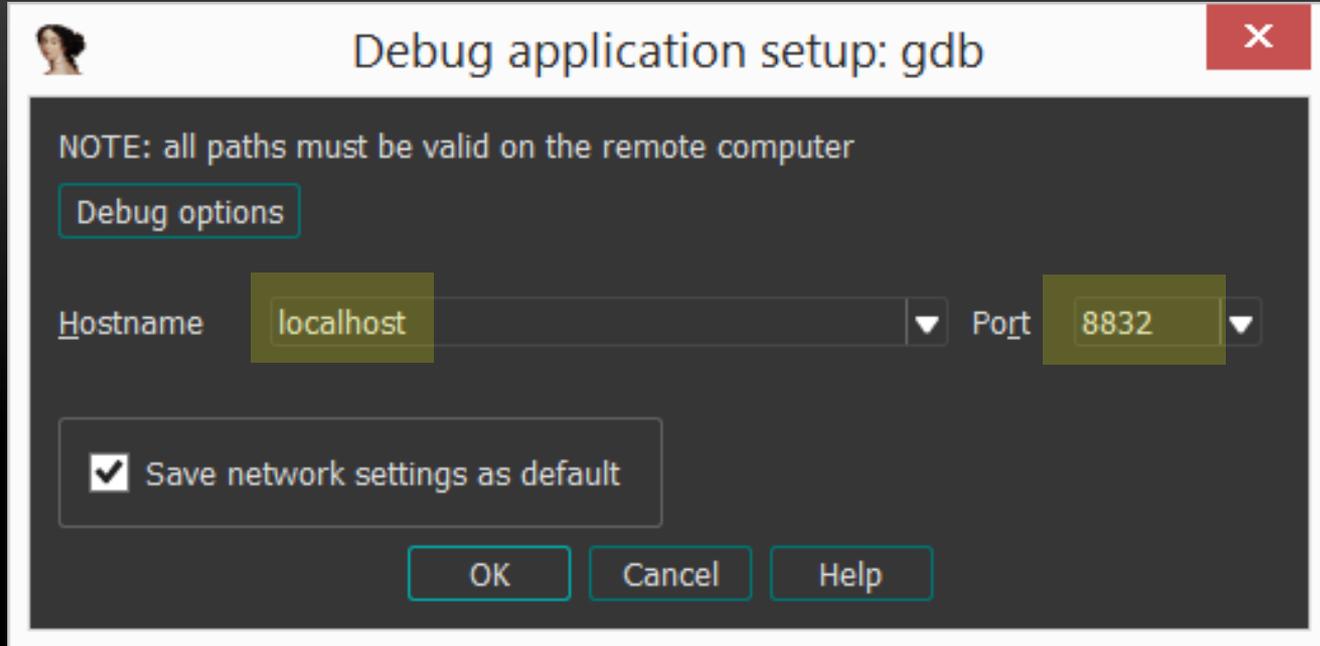
```

WARNING: Bochs version is higher than supported (2.6.8), please downgrade in case of problems.
FFFFFFFF: process BOCHS_DISKIMAGE_LDR has started (pid=1)
    
```

- ✓ Another way to **debug and analyze a MBR using IDA Pro** is also simple:
 - ✓ Download the `ida.py` from http://hexblog.com/ida_pro/files/mbr_bochs.zip
 - ✓ Copy the `ida.py` to your preferred folder (I've copied it to Bochs installation folder), edit the first lines to adapt it to your case:

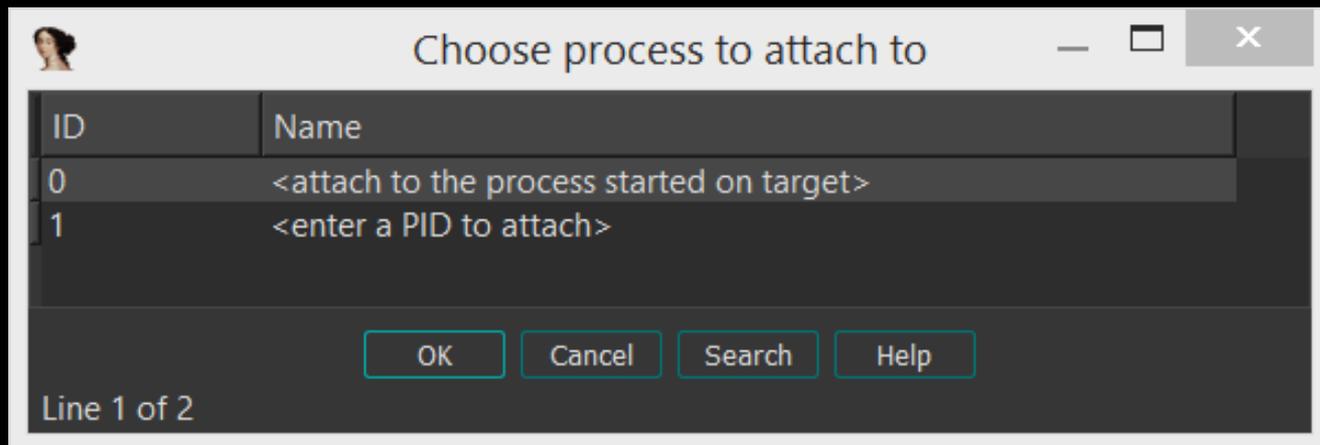
```
# Some constants
SECTOR_SIZE = 512
BOOT_START  = 0x7C00
BOOT_SIZE   = 0x7C00 + SECTOR_SIZE * 2
BOOT_END    = BOOT_START + BOOT_SIZE
SECTOR2     = BOOT_START + SECTOR_SIZE
MBRNAME     = "C:\VMs\mbr_infected.bin"
IMGNAME     = "C:\VMs\infected.raw"
```

- ✓ A better approach is to use a debugger instead of using an emulator.
- ✓ If you are using VMware Workstation, change the .vmx configuration file from the target machine to include the following lines:
 - ✓ `monitor.debugOnStartGuest32 = "TRUE" / monitor.debugOnStartGuest64 = "TRUE"`
 - ✓ Breaks on the first instruction since the power on.
 - ✓ `debugStub.listen.guest32 = "TRUE" / debugStub.listen.guest64 = "TRUE"`
 - ✓ Enables guest debugging.
 - ✓ `debugStub.hideBreakpoints = "TRUE"`
 - ✓ Use hardware breakpoint instead of using software breakpoints.
- ✓ Power on the virtual machine.
- ✓ Launch the IDA Pro, go to Debugger → Attach → Remote GDB debugger

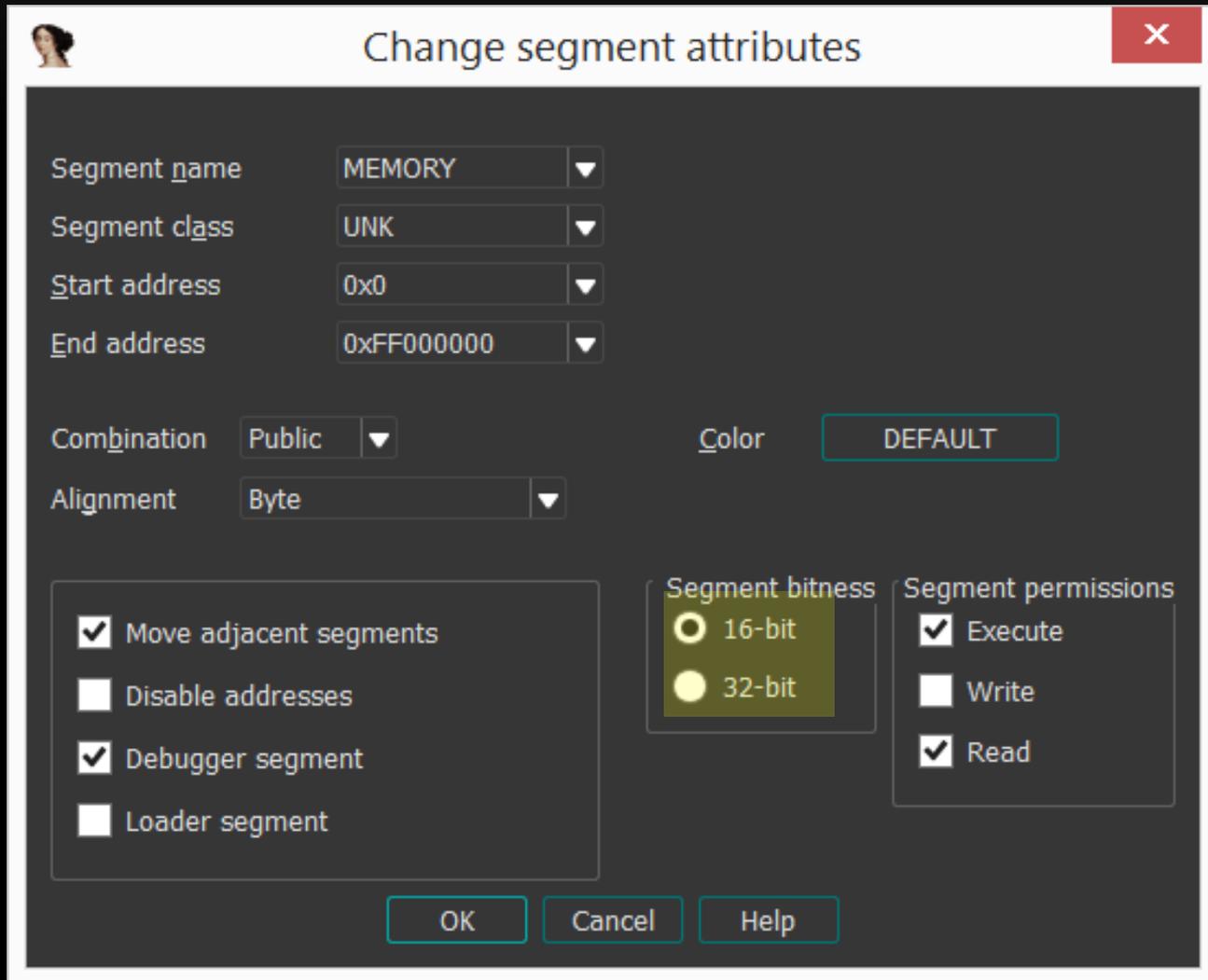


- ✓ We've set **Hostname** as "localhost" because we start the debugger in the same host of the VM.
- ✓ The **debugging port** must be 8832.

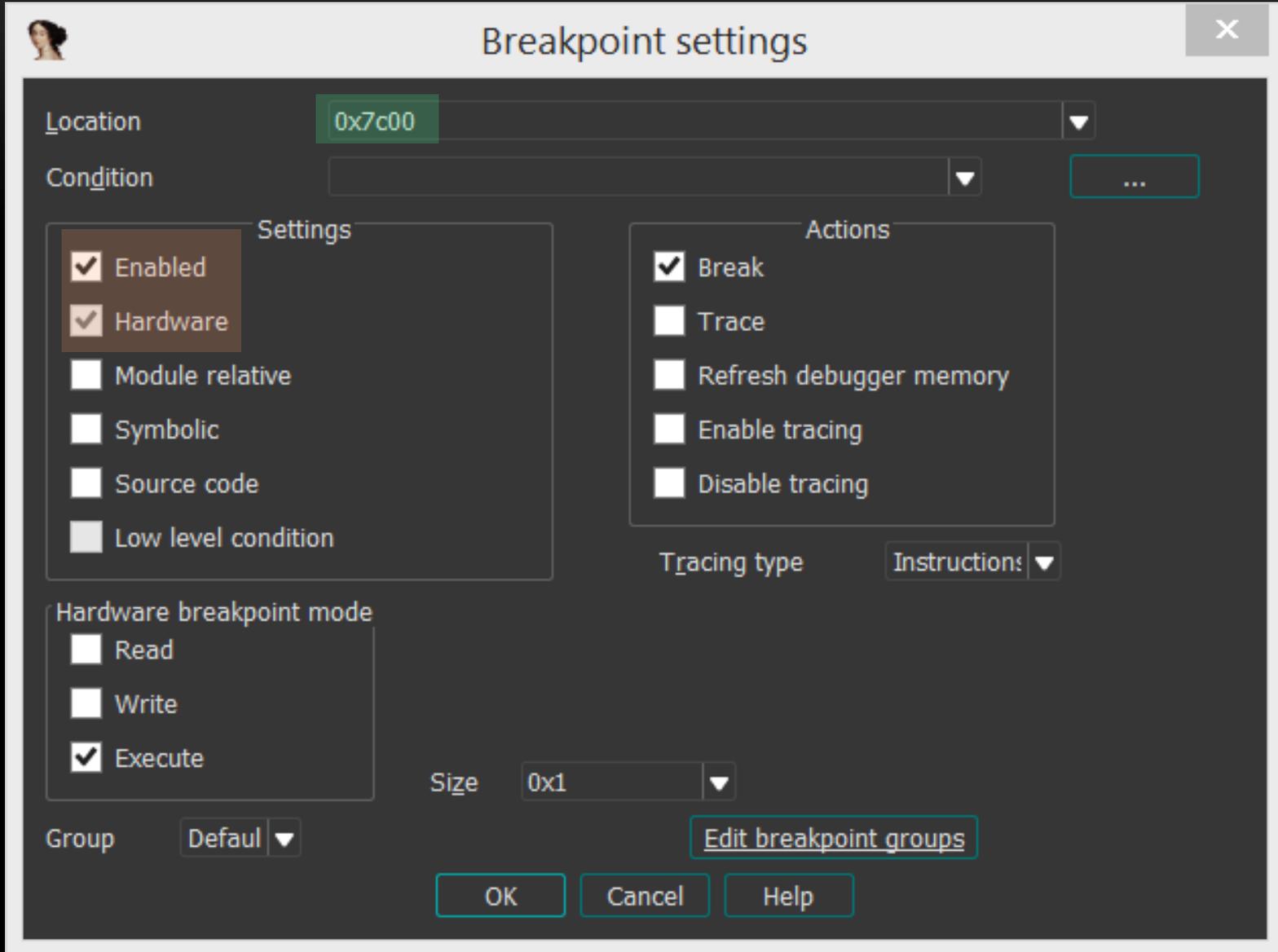
- ✓ After configuring the Debug application setup, click on **OK button** and choose "attach to the process started on target" as shown below.



- ✓ After debugger starting, go to **Views** → **Open subviews** → **Segments** (or hit SHIFT+F7), right click and go to **“Edit Segments”**.
- ✓ Change the **“Segment bitness”** option to **16-bit** (remember: MBR run in real mode, which is 16-bit):



- ✓ Go to **Debugger** → **Breakpoints** → **Add breakpoint**
- ✓ Set the **breakpoint at 0x7c00** (start of the MBR code).



✓ Continue the process (F9) and discard eventual exceptions. 😊

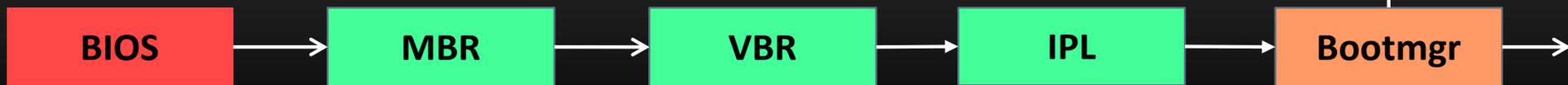
The screenshot displays the IDA Pro interface with several panels:

- Assembly View:** Shows assembly instructions with memory addresses. The instruction at `MEMORY:7C00` is highlighted: `xor ax, ax`. Other instructions include `mov ss, ax`, `mov sp, 7C00h`, `mov es, ax`, `mov ds, ax`, `mov si, 7C00h`, `mov di, 600h`, `mov cx, 200h`, `cld`, `rep mousb`, `push ax`, `push 61Ch`, `retf`, `sti`, `mov cx, 4`, `mov bp, 7BEh`, `cmp byte ptr [bp+0], 0`, and `j1 short loc_7C34`.
- General Registers:** Lists registers and their values. EAX is 00000000, ECX is 00000000, EDX is 00000080, EBX is 00000000, ESP is 000003EC, EBP is 00000000, ESI is 00000000, EDI is 00000000, EIP is 00007C00, EFL is 00000206, CS is 00000000, SS is 00000000, DS is 00000040, ES is 00000000, FS is 0000EA54, GS is 0000F000, and ST0 is 0 0.
- Modules:** Shows a single module: `<GDB remote process>`.
- Threads:** Shows two threads: Thread 1 (Decimal 1, Hex 1, State Ready) and Thread 4 (Decimal 4, Hex 4, State Ready).
- Stack View:** Shows memory addresses and values: `03EC EA543429 MEMORY:EA543429`, `03F0 D37A0001 MEMORY:D37A0001`, `03F4 04007BFE MEMORY:4007BFE`, `03F8 6B010001 MEMORY:6B010001`, and `03FC D37A0004 MEMORY:D37A0004`.

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

Subverting INT 13h would be lethal because winload.exe use it to load its modules.

The bootmgr uses the INT 13h disk service (from real mode) to access the disk service in protected mode.



UEFI is supported since Windows 7 SP1 x64

It holds the boot configuration information



Bootkits could attack it before loading the kernel and ELAM. 😊



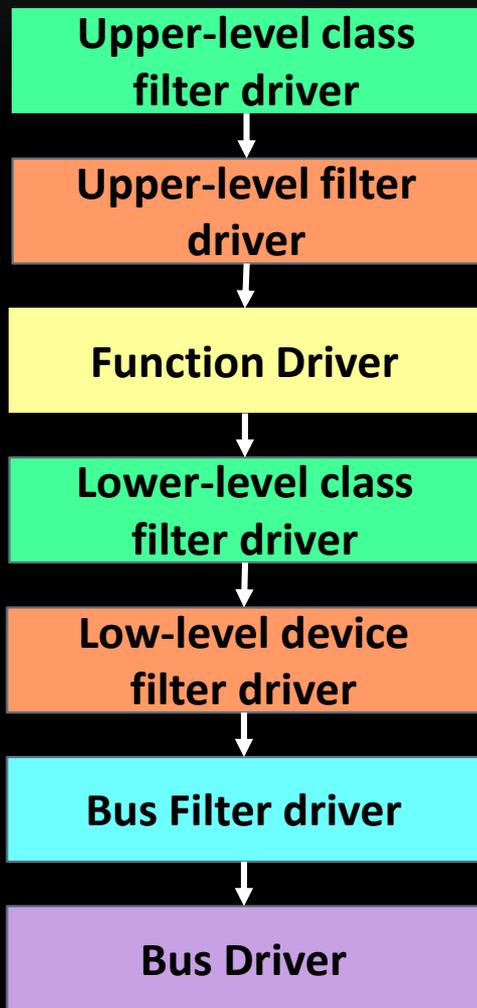
Code integrity is shared between kernel and ci.dll, but nt!CiEnable variable controls everything (Win 7 only).



- ✓ Classifies modules as good, bad and unknown.
- ✓ Additionally, it decides whether load a module or not according to the policy.

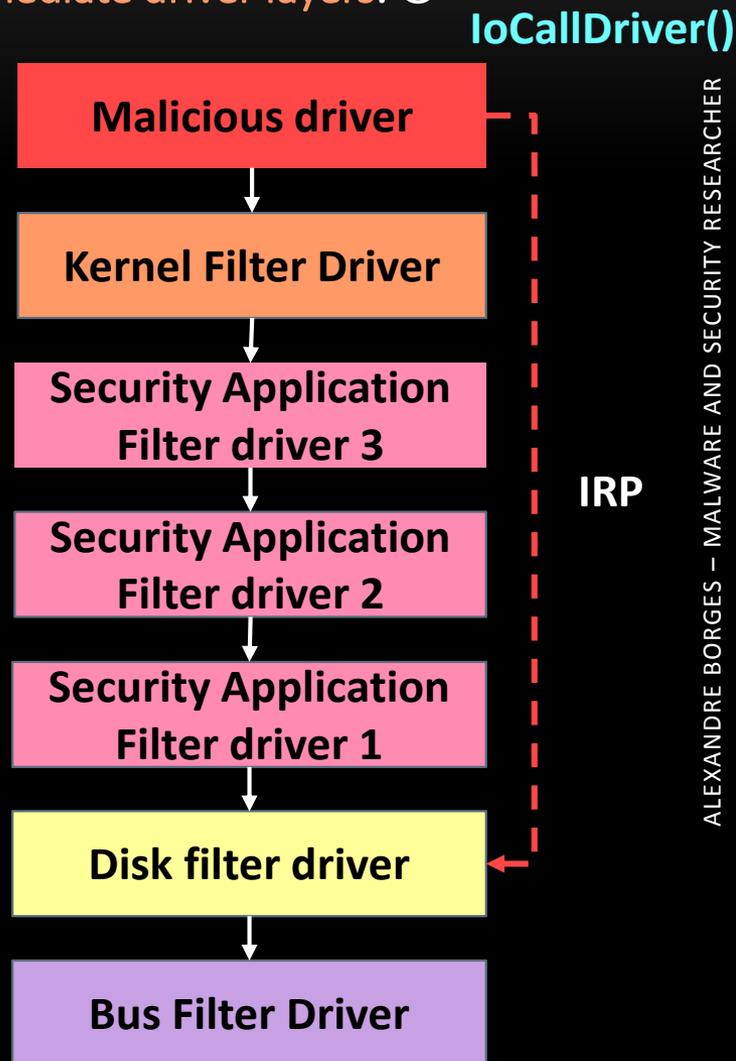
✓ Compromising the MBR code makes possible to load any malicious code from anywhere (even encrypted and from a hidden storage) and compromise the kernel by disabling the code integrity module, so making possible to load malicious kernel drivers (rootkits).

✓ Remember that any malicious driver can “bypass” intermediate driver layers. 😊



Driver development is usually done in pair, where the class driver handle general tasks, while the miniport-driver implement specific routines to the individual device.

Using the right I/O control code (IOCTL_SCSI_PASS_THROUGH_DIRECT), the malicious driver is able to “bypass” protections provide by programs.



✓ Additional bootkit/rootkit techniques:

- ✓ Hooking the `IRP_MJ_INTERNAL_CONTROL` handler from the mini-port disk driver object (`DRIVER_OBJECT`) to monitor/modify the data flow to disk.
- ✓ Bootkits/rootkits use **callback methods** to be notified about important events:
 - ✓ `PsSetLoadImageNotifyRoutine`: provides notification when a process, library or kernel memory is mapped into memory.
 - ✓ `PsSetCreateThreadNotifyRoutine`: points to a routine that is called when a thread starts or ends.
 - ✓ `IoRegisterFsRegistrationChange`: provides notification when a filesystem becomes available.

- ✓ **IoRegisterShutdownNotification**: the driver handler (IRP_MJ_SHUTDOWN) acts when the system is about going to down.
- ✓ **KeRegisterBugCheckCallback**: helps drivers to receive a notification to clean up before the shutdown.
- ✓ **PsSetCreateProcessNotifyRoutine**: this callback is invoked when a process starts or finishes. Usually, it is used by AVs and security programs.
- ✓ **CmRegisterCallback()** or **CmRegisterCallbackEx()** functions are called by drivers to register a **RegistryCallback** routine. This kind of callback is invoked when threads performs operations on the registry.
- ✓ Malware has been using **RegistryCallback routines** to **check whether their persistence entries are kept** and, just in case they have been removed, so the malware is able to add them back.

- ✓ Compromising INT 1 interruption, which is responsible for handling debugging events.
- ✓ Hiding partitions/filesystems at end of the disk. Additionally, encrypting them.
- ✓ Hooking key kernel modules routines such as `DriverUnload()` to prevent anyone to unload the malicious module.
- ✓ Some rootkits call `NtRaiseHardError()` to force a crash and, afterwards, loading a malicious driver.
- ✓ To force the BIOS reload the MBR to the memory (once it is infected or MFT is encrypted), the INT 19h is used.

- ✓ UEFI has changed the bootkit's attack profile: previously, BIOS' industry didn't have any standard, but UEFI established an unique one. Thus, a malware could be use to attack any platform (Write once, reuse always) 😊
- ✓ MBR + VBR + IPL are completely removed by UEFI. Additionally, UEFI support GPT format, whose signature is 0x200.
- ✓ UEFI is stored in the SPI flash and most part of the UEFI code is run in protected mode.
- ✓ The new bootmgfw.efi locates the winload.efi kernel loader (small changes.... 😊)

- ✓ Windows 8 has introduced the necessary support to UEFI Secure Boot.
- ✓ UEFI Secure Boot offers protection to boot components (OS bootloaders, UEFI DXE drivers and so on) against modification, but it doesn't offer protection against malware infecting the firmware.
- ✓ UEFI Secure Boot uses PKI to validate UEFI modules loaded from SPI.
- ✓ Unfortunately, this approach doesn't work with Terse Executable (TE) format, which doesn't have embedded digital signature. 😞

- ✓ **UEFI Secure Boot** is composed by:
 - ✓ **Platform key (PK)**, which establishes a trust relationship between the platform owner and the platform firmware. This platform key **verifies** the **KEK (Key Exchange Key)**.
 - ✓ **KEK** establishes a trust relationship between the platform firmware and OS.
 - ✓ Additionally, the **KEK verifies db and dbx** (both in NVRAM):
 - ✓ **Authorized Database (db)**: contains authorized signing certificates and digital signatures.
 - ✓ **Forbidden Database (dbx)**: contains forbidden certificates and digital signatures.
- ✓ Of course, if the **Platform Key is corrupted**, so everything is not valid anymore because the **Secure Boot turns out disabled**. ☹️

- ✓ **Another two databases** that are also used by Secure Boot:
 - ✓ **dbr**: contains **public key (certificates)** used to **validate OS recovery loader's signatures**.
 - ✓ **dbt**: contains **timestamping certificates** used to check an digital signature's timestamp of UEFI executable, which **prevents the usage of an expired signature in an executable**.
- ✓ **Pay attention**: the **security of all components** are based on the integrity of the SPI Flash.
- ✓ Of course, once we could **modify the SPI Flash content**, so the **UEFI Secure Boot could be disabled**.
- ✓ To help us to **detect any compromise of platform firmware**:
 - ✓ **Verified Boot**: checks whether the **platform firmware was modified**.
 - ✓ **Measured Boot**: **get hashes from boot components and stores them into the TPM** (Trusted Platform Module) configuration registers.

✓ UEFI components: SEC → PEI → DXE → BDS → TSL → RT → AL

- ✓ SEC → Security (Caches, TPM and MTRR initialization)
- ✓ PEI → Pre EFI Initialization (chipset initialization + memory controller)
- ✓ DXE → Driver Execution Environment (SMM initialization + devices initialization , Dispatch Drivers, FV enumeration)
- ✓ BDS → Boot Device Select (Hardware discovery + physical device enumeration)
- ✓ TSL → Transient System Load
- ✓ RT → Run Time
- ✓ BDS + DXE are responsible for finding the OS loader (path indicated by UEFI variable)

- ✓ Before proceeding, remember about **SMM basics (ring -1)**:
 - ✓ Interesting place to **hide malware** because is **protected from OS and hypervisors**.
 - ✓ The **SMM executable code** is **copied into SMRAM** and **locked there** during the initialization.
 - ✓ To **switch to SMM**, it is necessary to **trigger a SMI (System Management Interrupt)** and save the current content into SMRAM, so the **SMI handler is being executed**.
 - ✓ **SMI handlers** works as **interfaces between the OS and hardware**.
 - ✓ **Compromising a SMM driver**, for example, makes possible to **gain SMM privilege** and, from this point, to **disable the SPI Flash protection** and **modify a DXE driver**. Game over. 😊

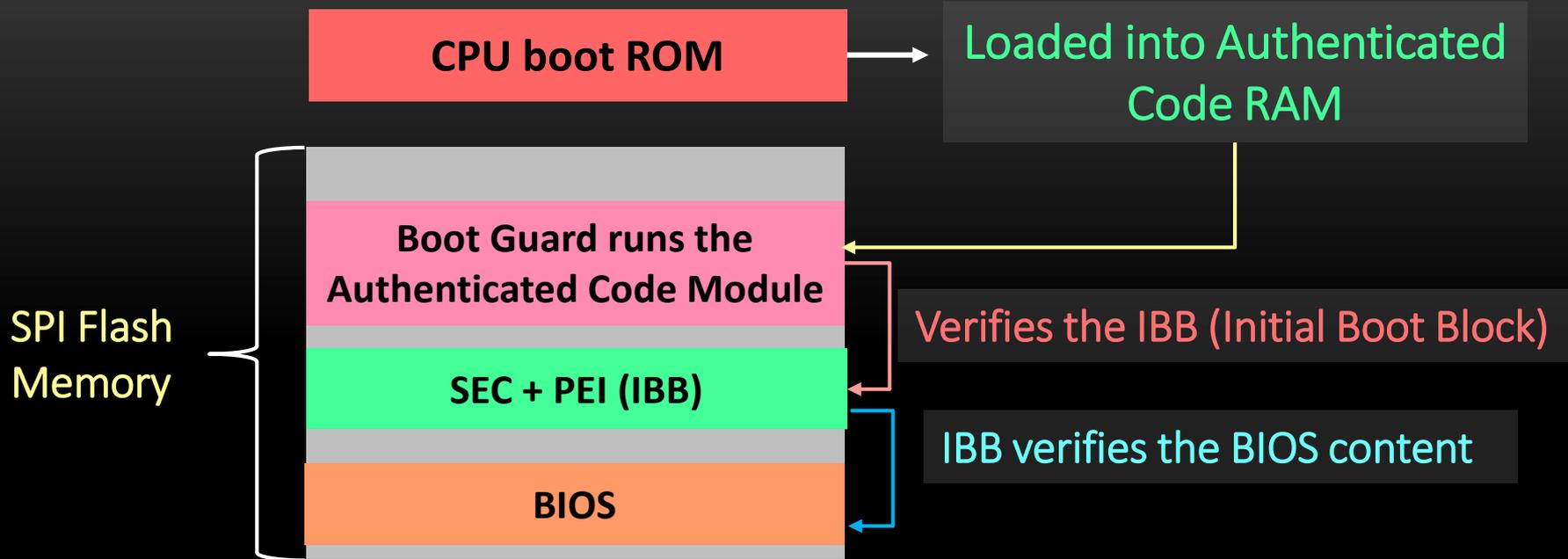
✓ user mode malware → rootkit → SMM → SPI flash / BIOS

- ✓ If the **OS Secure Boot is disabled**, the boot process can be compromised because the Patch Guard is only “running” after the boot process.
- ✓ Check the **KPP (Kernel Patch Protection) protected areas** that are covered:

✓ `!analyze -show 109` (on WinDbg)

- ✓ The **UEFI Secure Boot** protects and prevents attacks to modify any component component before the OS boot stage.
- ✓ Who does protect the system before **UEFI Secure Boot** being active?
 - ✓ **Boot Guard**, which is based on cryptographic keys. 😊
- ✓ Who does protect the platform against attacks trying to compromise the flash and the entire platform?
 - ✓ **BIOS Guard**, which protect and guarantee the integrity of the BIOS. 😊

- ✓ **Boot Guard** is used to validate the boot process by flashing a public key associated to the BIOS signature into the FPFs (Field Programmable Fuses) within the Intel ME.
- ✓ Is it a perfect solution? Unfortunately, few vendors have left these fuse unset. It could be lethal. 😊
- ✓ Additionally, a malware could alter the **flash write protection** and change the SPI flash.
- ✓ Even using the **Boot Guard** to protect the boot process, we have to protect the SPI flash using the BIOS Guard to protect against a **SMM driver rootkit**, for example.
- ✓ **BIOS Guard** is essential because, in the past, some malware threats already attacked the system by modifying the SMI routine of BIOS to compromise the update process.



- ✓ The **ACM** implements **Verified and Measured Boot**. 😊
- ✓ **Public key's hash**, which is used to verify the signature of the code with the ACM, is **hard-coded within the CPU**.
- ✓ **Boot Guard** protection makes modifying BIOS very hard if the attacker doesn't know the private key.
- ✓ At end, it works as a **certificate chain checking**. 😊

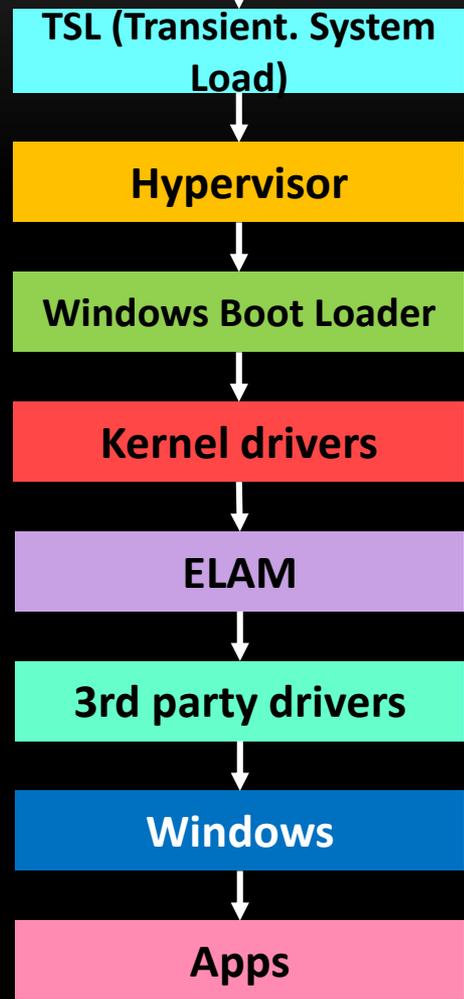
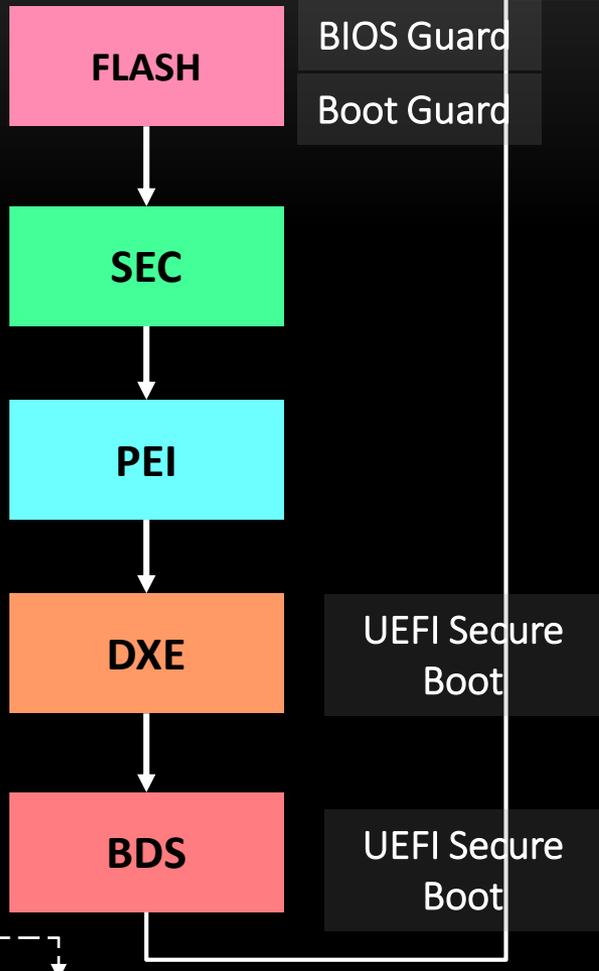
- ✓ In this case, we have the **BIOS Guard**, which protects the entire platform against attacks:
 - ✓ **SPI Flash Access**, preventing an attacker to escalate privileges to SMM by altering the SPI.
 - ✓ **BIOS update**, which an attacker could update/replace the BIOS code with a bad-BIOS version through a DXE driver.
- ✓ **BIOS Guard** forces that **only trusted modules**, authorized by ACM, are able to modify the flash memory.
- ✓ Thus, protecting **against implants**.

✓ Modifying an existing DXE driver (or add a new one) could allow malicious execution at DXE stage.

✓ The Windows uses the UEFI to load the Hypervisor and Secure Kernel.

malware and exploits attack here 😊

IBB



OS Secure Boot
Acts on drivers that are executed before Windows being loaded and initialized.

✓ It is possible to modify a UEFI DXE driver by compromising the SPI flash protection, so bypassing/disabling the UEFI Secure Boot.

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

UEFITool NE alpha 55 (Feb 10 2019) - spihitb.bin

File Action View Help

Structure

Name	Action	Type	Subtype	Text
Intel image		Image	Intel	
Descriptor region		Region	Descriptor	
GbE region		Region	GbE	
ME region		Region	ME	
BIOS region		Region	BIOS	
Padding		Padding	Non-empty	
EfiFirmwareFileSystemGuid		Volume	FFSv2	
Microcode		File	Raw	
Volume free space		Free space		
EfiFirmwareFileSystemGuid		Volume	FFSv2	
S3Restore		File	PEI module	S3Resume
PEI dependency section		Section	PEI dependency	
PE32 image section		Section	PE32 image	
UI section		Section	UI	
FV_MAIN_NESTED		File	Volume image	FV_MAINNested
Compressed section		Section	Compressed	
Volume image section		Section	Volume image	
EfiFirmwareFileSystemGuid		Volume	FFSv2	
UI section		Section	UI	
Volume free space		Free space		
EfiFirmwareFileSystemGuid		Volume	FFSv2	
Pad-file		File	Pad	
Capsule		File	PEI module	Capsule
PEI dependency section		Section	PEI dependency	
PE32 image section		Section	PE32 image	
UI section		Section	UI	
OEMPEI		File	PEI module	OEMPEI
TcgPei		File	PEI module	TcgPei
TxtPei		File	PEI module	TxtPei
BiosAc		File	Raw	
Pad-file		File	Pad	
B1AEE818-B959-487B-A795-16C2A54CB36E		File	PEI core	PeiMain
FB8415B7-EA7E-4E6D-9381-005C3BD1DAD7		File	PEI module	DellEcConfigPei
A157968A-163A-45E8-8848-C39CD50125D5		File	PEI module	DellIsItXfrConfigPei
5924BE03-9DD8-4BAB-808F-C21CABFE0B4B		File	PEI module	DellErrorHandlerPei
E747D8FF-1794-48C6-96D7-A419D9C60F11		File	PEI module	DellSioPolicyConfigPei
DEBA5A2C-D788-47FB-A0B5-20CA8E58DFEC		File	PEI module	DellSystemIdConfigPei
A27E7C62-249F-4B7B-BD5C-807202035DEC		File	PEI module	DellFlashUpdatePei
MemoryInit		File	PEI module	MemoryInit
81F0BCF2-F1AD-4DDE-9E5B-75EB3427ABC4		File	PEI module	DellMfgModePeiDriver
WdtPei		File	PEI module	WdtPei
CORE PEI		File	PEI module	CORE PEI

Information

Fixed: Yes
Base: 0h
Address: FF60000h
Offset: 0h
Full size: A0000h (10485760)
Flash chips: 2
Regions: 4
Masters: 3
PCH straps: 18
PROC straps: 1

Structure → Capsule update is used update the UEFI components.

Name	Action	Type	Subtype	Text
UEFI capsule		Capsule	UEFI 2.0	
UEFI image		Image	UEFI	
EfiFirmwareFileSystem2Guid		Volume	FFSv2	
D1157A19-7DD0-4483-AAD1-3B1F969644EF		File	Volume image	
24400798-3807-4A42-B413-A1ECEE205DD8		Section	GUID defined	
Volume image section		Section	Volume image	
EfiFirmwareFileSystem2Guid		Volume	FFSv2	
3D93D660-6EBA-463F-8D8F-A2B62F83E85C		File	Freeform	
098D0689-4245-4F65-80C9-7F3202C5F44E		File	Freeform	
D005D5F0-9875-4AEA-8F39-96FC50DAEB94		File	Freeform	
AFCCAA0E-E825-441E-A353-157F1E9D8289		File	Raw	
5BA2DCF0-B551-47D6-9608-C3EA68E52E4C		File	Raw	
ADB9C28D-5CC8-4FB5-8179-849358B68441		File	Raw	
A90AF0B1-865D-439E-8294-1197D24F85B8		File	Raw	
1150E536-25E0-47EA-A54F-A4FC4AF34E3C		File	Raw	
94B5FCF2-0173-4F0B-8D43-64DF9588B8C7		File	DXE driver	
29FF2C20-0C83-4D57-9ED1-26BE925216EB		File	DXE driver	
E002109B-4077-4EA9-8B85-90562C57B093		File	DXE driver	
DXE dependency section		Section	DXE dependency	
Compressed section		Section	Compressed	
6A467FFC-621B-4...		File	DXE driver	
E75C7ED6-65F1-4...		File	DXE driver	
E44985E2-EEC6-4...		File	DXE driver	
CB7F3CBD-C6C8-4...		File	DXE driver	
9D8C2ADC-CC44-4...		File	DXE driver	
27DC3F10-CA4B-4...		File	DXE driver	
C3DB7E95-DB3C-4...		File	DXE driver	
Volume free spa...		Free space		
12D58591-E491-4E89-AE...		File	Freeform	
B1B697A6-669B-4DB9-1...		Section	Freeform subtype GUID	
895733C8-D6E2-4E88-5...		Section	Freeform subtype GUID	
91AFFF70-F98E-4D3A-7...		Section	Freeform subtype GUID	
8A7D7ACD-CFF4-4B17-7...		Section	Freeform subtype GUID	
65AA90FF-C660-4C9D-7...		Section	Freeform subtype GUID	
8A7445BE-DEED-4883-5...		Section	Freeform subtype GUID	
4D84F7CA-37D8-42DB-1...		Section	Freeform subtype GUID	
8CB42898-12A6-44E7-AD...		File	Raw	
Volume free space		Free space		

- Hex view... Ctrl+D
- Body hex view... Ctrl+Shift+D
- Extract as is... Ctrl+E
- Extract body... Ctrl+Shift+E
- Rebuild Ctrl+Space
- Insert into... Ctrl+I
- Insert before... Ctrl+Alt+I
- Insert after... Ctrl+Shift+I
- Replace as is... Ctrl+R
- Replace body... Ctrl+Shift+R
- Remove Ctrl+Del

Possible place to compromise the UEFI image.

- ✓ Exists other **SPI flash protections** that are set up at **DXE stage**:
 - ✓ **SMM_BWP (SMM BIOS Write Protection)**: protects SPI flash **against writing from malware running outside of the SMM**.
 - ✓ **BLE (BIOS Lock Enable bit)**: protects the SPI flash **against unauthorized writes**. Unfortunately, it can be modified by malware with SMM privileges.
 - ✓ **BIOSWE (BIOS Write Enable Bit)**: it is a kind of “control bit”, which is used to allow a BIOS update.
 - ✓ **Protected Ranges**: it is designed to **protect specific regions as SPI flash**, for example.
 - ✓ Additionally, there are **six Protected Ranges registers: PR0 to PR5**.
 - ✓ No doubts, it is a **good protection against changes from SMM** because its policies can't be changed from SMM. 😊

✓ chipsec_util.py spi dump spihitb.bin

```

PS C:\chipsec-master> python chipsec_util.py spi dump spihitb.bin

#####
##                                                                 ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                                 ##
#####
[CHIPSEC] Version 1.3.5.dev1

WARNING: *****
WARNING: Chipsec should only be used on test systems!
WARNING: It should not be installed/deployed on production end-user systems.
WARNING: See WARNING.txt
WARNING: *****

[CHIPSEC] API mode: using CHIPSEC kernel module API
[CHIPSEC] Executing command 'spi' with args ['dump', 'spihitb.bin']

[CHIPSEC] dumping entire SPI flash memory to 'spihitb.bin'
[CHIPSEC] it may take a few minutes (use DEBUG or VERBOSE logger options to se
[CHIPSEC] BIOS region: base = 0x00600000, limit = 0x009FFFFF
[CHIPSEC] dumping 0x00A00000 bytes (to the end of BIOS region)
[spi] reading 0xa00000 bytes from SPI at FLA = 0x0 (in 163840 0x40-byte chunks
[CHIPSEC] completed SPI flash dump to 'spihitb.bin'
[CHIPSEC] (spi dump) time elapsed 134.150

```

✓ chipsec_util.py decode spi.bin

Master Read/Write Access to Flash Regions

Region	CPU	ME
0 Flash Descriptor	R	R
1 BIOS	RW	
2 Intel ME		RW
3 GBe	RW	RW

- ✓ Remember that a **BIOS update** could be composed by different parts such as **CPU microcode** (internal firmware), **Gbe** (hardware network stack), **BMC** (Baseboard Management Controller, which provides monitoring and management), **AMT** (Active Management Platform, which provides remote access to devices), **ME** (Management engine), **EC** (Embedded Controller) and so on.
- ✓ **ME**: an x86 controller that provides **root-of-trust**.
- ✓ **EC**: defines which component has read/write access to other regions. It also works as **security root of trust**.

✓ chipsec_main --module common.bios_wp

```
[*] running module: chipsec.modules.common.bios_wp
[X] [
[X] [ Module: BIOS Region Write Protection
[X] [
[*] BC = 0x02 << BIOS Control (b:d.f 00:31.0 + 0xDC)
[00] BIOSWE = 0 << BIOS Write Enable
[01] BLE = 1 << BIOS Lock Enable
[02] SRC = 0 << SPI Read Configuration
[04] TSS = 0 << Top Swap Status
[05] SMM_BWP = 0 << SMM BIOS Write Protection
[!] Enhanced SMM BIOS region write protection has not been enabled

[*] BIOS Region: Base = 0x00600000, Limit = 0x009FFFFFFF
SPI Protected Ranges
-----
PRX (offset) | Value | Base | Limit | WP? | RP?
-----
PR0 (74) | 00000000 | 00000000 | 00000000 | 0 | 0
PR1 (78) | 00000000 | 00000000 | 00000000 | 0 | 0
PR2 (7C) | 00000000 | 00000000 | 00000000 | 0 | 0
PR3 (80) | 00000000 | 00000000 | 00000000 | 0 | 0
PR4 (84) | 00000000 | 00000000 | 00000000 | 0 | 0
[!] None of the SPI protected ranges write-protect BIOS region
```

✓ Unfortunately, the SMM BIOS write protection (SMM_BWP), which protects the entire BIOS area, is not enabled. ☹️

✓ chipsec_main.py -m common.spi_lock

```

[*] running module: chipsec.modules.common.spi_lock
[X] [ =====
[X] [ Module: SPI Flash Controller Configuration Lock
[X] [ =====
[*] HSFS = 0xE008 << Hardware Sequencing Flash Status Register (SPIBAR + 0x4)
[00] FDONE = 0 << Flash Cycle Done
[01] FCERR = 0 << Flash Cycle Error
[02] AEL = 0 << Access Error Log
[03] BERASE = 1 << Block/Sector Erase Size
[05] SCIP = 0 << SPI cycle in progress
[13] FDOPSS = 1 << Flash Descriptor Override Pin-Strap Status
[14] FDV = 1 << Flash Descriptor Valid
[15] FLOCKDN = 1 << Flash Configuration Lock-Down
[+] PASSED: SPI Flash controller configuration is locked

```

```

[CHIPSEC] ***** SUMMARY *****
[CHIPSEC] Time elapsed 0.038
[CHIPSEC] Modules total 1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed 1:
[+] PASSED: chipsec.modules.common.spi_lock
[CHIPSEC] Modules failed 0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****

```

✓ The **HSFSS.FLOCKDN bit**, which comes from HSFSTS SPI MMIO Register, prevents changes to Write Protection Enable bit.

✓ At end, a **malware** couldn't disable the SPI protected ranges to enable access to SPI flash memory. 😊

✓ python chipsec_main.py --module common.bios_ts

```
[+] loaded chipsec.modules.common.bios_ts
[*] running loaded modules ..

[*] running module: chipsec.modules.common.bios_ts
[X] [ =====
[X] [ Module: BIOS Interface Lock (including Top Swap Mode)
[X] [ =====
[*] BiosInterfaceLockDown (BILD) control = 1
[*] BIOS Top Swap mode is disabled (TSS = 0)
[*] RTC TopSwap control (TS) = 0
[+] PASSED: BIOS Interface is locked (including Top Swap Mode)
```

```
[CHIPSEC] ***** SUMMARY *****
[CHIPSEC] Time elapsed          0.043
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed          1:
[+] PASSED: chipsec.modules.common.bios_ts
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped      0:
[CHIPSEC] *****
```

- ✓ BIOS Top Swap Mode allows a fault-tolerant update of BIOS boot block.
- ✓ If BIOS Top Swap Mode is not locked, so malware could redirect the reset vector execution to the backup bootblock, so loading a malicious bootblock code. 😊

✓ python chipsec_main.py --module common.smrr

```
[*] Checking SMRR range base programming.
[*] IA32_SMRR_PHYSBASE = 0xCB000004 << SMRR Base Address MSR (MSR 0x1F2)
    [00] Type          = 4 << SMRR memory type
    [12] PhysBase     = CB00 << SMRR physical base address
[*] SMRR range base: 0x00000000CB000000
[*] SMRR range memory type is Write-through (WT)
[+] OK so far. SMRR range base is programmed

[*] Checking SMRR range mask programming.
[*] IA32_SMRR_PHYSMASK = 0xFF800800 << SMRR Range Mask MSR (MSR 0x1F3)
    [11] Valid        = 1 << SMRR valid
    [12] PhysMask     = FF80 << SMRR address range mask
[*] SMRR range mask: 0x00000000FF800000
[+] OK so far. SMRR range is enabled

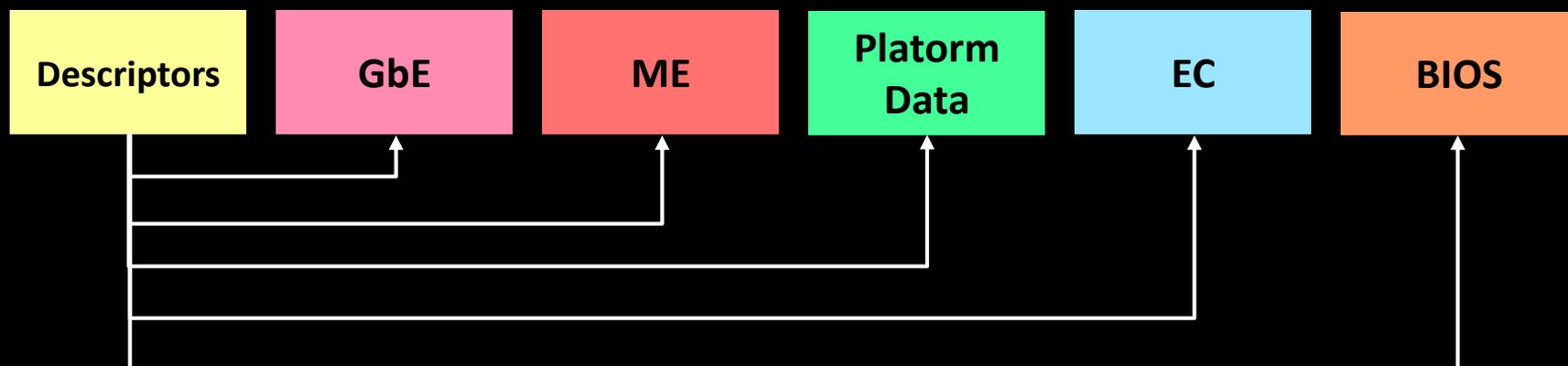
[*] Verifying that SMRR range base & mask are the same on all logical CPUs..
[CPU0] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[CPU1] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[CPU2] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[CPU3] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[CPU4] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[CPU5] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[CPU6] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[CPU7] SMRR_PHYSBASE = 00000000CB000004, SMRR_PHYSMASK = 00000000FF800800
[+] OK so far. SMRR range base/mask match on all logical CPUs
[*] Trying to read memory at SMRR base 0xCB000000..
[+] PASSED: SMRR reads are blocked in non-SMM mode

[+] PASSED: SMRR protection against cache attack is properly configured
```

- ✓ **SMRR (System Management Range Registers)** block the access to SMRAM (reserved by BIOS SMI handlers) while CPU is not in SMM mode, preventing it to execute SMI exploits on cache.

✓ Few important side notes are:

- ✓ Just in case the **SMM code** try to “read” some information from outside of **SMM**, so it would be interesting to check if the pointer is valid by using **SmmlsBufferOutsideSmmValid()** function.
- ✓ **ME (Management Engine) Applications** can use the **HECI (Host Embedded Communication Interface)** to communicate with the kernel from Windows, for example. So **ME and HECI handlers** are a very critical components.
- ✓ The same concern should be dedicated to **AMT**, which is an **ME application**.
- ✓ The **SPI flash** is composed by **descriptors, GbE, ME, Data, EC and BIOS**, where **ME has full access to the DRAM** and it is always working. 😊



✓ My sincere thank you to:

✓ HITB Conference staff.

✓ You, who have reserved some time attend my talk.

✓ Please, you should never forget:

“ The best of this life are people.” 😊



THANK YOU FOR ATTENDING MY TALK. 😊

➤ **Twitter:**

@ale_sp_brazil

@blackstormsecbr

- ✓ **Malware and Security Researcher.**
- ✓ **Speaker at DEFCON USA 2018**
- ✓ **Speaker at DEFCON CHINA 2019**
- ✓ **Speaker at CONFidence Conf. 2019**
- ✓ **Speaker at BSIDES 2018/2017/2016**
- ✓ **Speaker at H2HC 2016/2015**
- ✓ **Speaker at BHACK 2018**
- ✓ **Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics and Rootkits.**
- ✓ **Reviewer member of the The Journal of Digital Forensics, Security and Law.**
- ✓ **Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response**

➤ **Website:** <http://blackstormsecurity.com>

➤ **LinkedIn:**
<http://www.linkedin.com/in/aleborges>

➤ **E-mail:**
alexandreborges@blackstormsecurity.com