MODERN TECHNIQUES TO DEOBFUSCATE AND UEFI/BIOS MALWARE

HITBSecConf2019 - Amsterdam

by Alexandre Borges
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

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
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.
Proectors 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
```
#!/usr/bin/env ruby
#
require "metasm"
include Metasm

coderef = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOB)
  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

addrstart = 0
textcode = coderef.init_disassembler
textcode.disassemble(addrstart)
hitb_di = textcode.di_at(addrstart)
hitb = hitb_di.block
puts "\n[$] Our HITB 2019 Amsterdam test code:\n"
puts hitb.list

[v] based on metasm.rb file and Bruce Dang code.

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

Starts the disassembler engine and disassemble from the first address (zero).
HITB 2019 AMSTERDAM

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

Starts the backtrace engine and walk back in the code.

Determines which is the final instruction to walk back from there. 😊
values = textcode2.backtrace(final, dd.list.last.address, {:log => backlog, :include_start => true})

backlog.each{|item|
  case type = item.first
  when :start
    item, expression, address = item
    puts "[start] Here is the sequence of expression evaluations: #{expression} from 0x#{address.to_s(16)}\n"
  when :di
    item, new, old, instruction = item
    puts "[new update] instruction #{instruction}, \n updating expression once again old #{old} new #{new}\n"
  end
}
effective = backlog.select{|y| y.first==:di}.map{|y| y[3]}.reverse
puts "\nThe effective instructions are:\n\nputs effective
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 9h sub eax, 0a3h,
   --> updating expression once again old eax+ecx+166h new eax+ecx+0c3h

[new update] instruction 6h sub eax, ebx,
   --> updating expression once again old eax+ecx+0c3h new eax-ebx+ecx+0c3h

[new update] instruction 1h mov ebx, 0c3h,
   --> updating expression once again old eax-ebx+ecx+0c3h new eax+ecx

Great! ☺
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 as 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:/root/programs/metasm
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```


```c
#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;
}
```
HITB 2019 AMSTERDAM

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

```
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......RQ.b|
00000020  00 00 89 ca 59 42 83 c2 61 4a 01 d0 5a |....YB..aJ..Z|
0000002d  `````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
IDA Pro confirms our disassembly task. 😊
To install Capstone: `apt-get install libcapstone3 libcapstone-dev` 😊
HITB 2019 AMSTERDAM

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

Original code disassembled by Capstone. 😊
- 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.

This result confirms our previous conclusion.
MIASM is one of the most impressive frameworks 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 000000000000001070 (0)
INFO : generate graph file
INFO : generate intervals
[0x1070 0x10A2]
INFO : total lines 0

✓ (testing MIASM) xdot graph_execflow.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.

Generates a dot graph.

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

Sets 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.
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_0000000000000002D:0x0000002d
loc_0000000000000002D:0x0000002d
Our proposed code. 😊
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)
FFFF3D)) & ((EAX_init + ECX_init + 0xFFFFFF3D) ^ 0xFFFFFF3C)) ^ (EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFF
F3D) ^ 0xC3)[3:32]
pf = parity((EAX_init + ECX_init) & 0xFF)
zf = (EAX_init + ECX_init)?(0x0, 0x1)
af = (((EAX_init + ECX_init) ? (0x0, 0x1)
F3D) ^ 0xC3)[4:5]
of = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFF)
FF3D)) & ((EAX_init + ECX_init + 0xFFFFFF3D) ^ 0xFFFFFF3C)))[3:32]

Instr POP EDX
Assignblk:
IRDst = loc_0000000000000002D:0x0000002d

EAX = EAX_init + ECX_init

cf = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFF)
FFFF3D)) & ((EAX_init + ECX_init + 0xFFFFFF3D) ^ 0xFFFFFF3C)) ^ (EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFF
F3D) ^ 0xC3)[3:32]
pf = parity((EAX_init + ECX_init) & 0xFF)
zf = (EAX_init + ECX_init)?(0x0, 0x1)
af = (((EAX_init + ECX_init) ? (0x0, 0x1)
F3D) ^ 0xC3)[4:5]
IRDst = 0x2D
of = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFF)
FF3D)) & ((EAX_init + ECX_init + 0xFFFFFF3D) ^ 0xFFFFFF3C)))[3:32]
nf = (EAX_init + ECX_init)[3:32]

@32[ESP_init + 0xFFFFFF8] = ECX_init
@32[ESP_init + 0xFFFFFFF8] = 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 (PNPInicializeBadCriticalDrivers) 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. 😊
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 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:
    xor    ax, ax
    mov    ss, ax
    mov    sp, 7C00h
    mov    es, ax
    mov    ds, ax
    mov    si, 7C00h
    mov    di, 600h
    mov    cx, 200h
    cld
    rep movsb
    push   ax
    push   61Ch
    retf

    sti
    mov    cx, 4
    mov    bp, 7BEh

seg000:7C23  loc_7C23:
    cmp    byte ptr [bp+0], 0
    jl     short loc_7C34

seg000:7C27  loc_7C27:
    jnz    loc_7D3B
    add    bp, 10h

seg000:7C30  loc_7C30:
    loop   loc_7C23
    int    18h

seg000:7C32  loc_7C32:

seg000:7C37  loc_7C37:

seg000:7C3A  loc_7C3A:

seg000:7C3D  loc_7C3D:

seg000:7C3F  loc_7C3F:

seg000:7C49  loc_7C49:

seg000:7C53  loc_7C53:

seg000:7C59  loc_7C59:

seg000:7C5A  loc_7C5A:

seg000:7C61  loc_7C61:
    mov    [bp+0], dl
    push   bp
    mov    byte ptr [bp+11h], 5
    mov    byte ptr [bp+10h], 0
    mov    ah, 41h
```

Clean MBR. 😊
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
mbr_infected.bin: DOS/MBR boot sector

Install Bochs and create a bochsrc file pointing to the converted image above:

```plaintext
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"
```
Reading configuration from bochs.
Installing win32 module as the Bochs GUI
Using log file null.

Next at t=0

(0) [0x0000000000000000] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b ; ea5be000f0

(bochs:1) lb 0x7c00

(bochs:2) info break

Num Type Disp   Enb Address
  1 breakpoint keep y 0x0000000000000007c00

(bochs:3) c

(0) Breakpoint 1, 0x0000000000000007c00 in ?? ()

Next at t=17404827

(0) [0x0000000000000007c00] 0000:7c00 (unk. ctxt): cli ; fa

(bochs:4) u /8

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00007c00:</td>
<td>cli</td>
<td>; fa</td>
</tr>
<tr>
<td>00007c01:</td>
<td>xor eax, eax</td>
<td>; 6631c0</td>
</tr>
<tr>
<td>00007c04:</td>
<td>mov ss, ax</td>
<td>; 8ed0</td>
</tr>
<tr>
<td>00007c06:</td>
<td>mov es, ax</td>
<td>; 8ec0</td>
</tr>
<tr>
<td>00007c08:</td>
<td>mov ds, ax</td>
<td>; 8ed8</td>
</tr>
<tr>
<td>00007c0a:</td>
<td>mov sp, 0x7c00</td>
<td>; bc007c</td>
</tr>
<tr>
<td>00007c0d:</td>
<td>sti</td>
<td>; fb</td>
</tr>
<tr>
<td>00007c0e:</td>
<td>mov byte ptr ds:0x7c93, dl</td>
<td>; 8816937c</td>
</tr>
</tbody>
</table>

Next at t=17404828

(0) [0x0000000000000007c01] 0000:7c01 (unk. ctxt): xor eax, eax ; 6631c0

(bochs:6) s

Next at t=17404829
Infected MBR being debugged 😊
Another way to debug and analyze a MBR using IDA Pro is also simple:

1. Download the ida.py from http://hexblog.com/ida_pro/files/mbr_bochs.zip

2. 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:

```python
# 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. 😊
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

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.

IoCallDriver()
✓ 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 work 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 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.
CPU boot ROM

Boot Guard runs the Authenticated Code Module

SEC + PEI (IBB)

BIOS

Loaded into Authenticated Code RAM

Verifies the IBB (Initial Boot Block)

IBB verifies the BIOS content

SPI Flash Memory

✓ 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.

It is possible to modify a UEFI DXE driver by compromising the SPI flash protection, so bypassing/disabling the UEFI Secure Boot.
https://github.com/LongSoft/UEFITool
Capsule update is used to update the UEFI components. 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. 😊
```bash
PS C:\chipsec-master> python chipsec_util.py spi dump spihitb.bin
```

```text
#CHIPSEC: Platform Hardware Security Assessment Framework

[CHIPSEC] Version 1.3.5.dev1

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

[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 see
[CHIPSEC] BIOS region: base = 0x00600000, limit = 0x009FFFFFF
[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

<table>
<thead>
<tr>
<th>Region</th>
<th>CPU</th>
<th>ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Flash Descriptor</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>1 BIOS</td>
<td>RW</td>
<td>R</td>
</tr>
<tr>
<td>2 Intel ME</td>
<td>RW</td>
<td>RW</td>
</tr>
<tr>
<td>3 GBe</td>
<td>RW</td>
<td>RW</td>
</tr>
</tbody>
</table>

✓ 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**.
Unfortunately, the SMM BIOS write protection (SMM_BWP), which protects the entire BIOS area, is not enabled. 😞
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
- Module: BIOS Interface Lock (including Top Swap Mode)
- 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)

**SUMMARY**
- Time elapsed: 0.043
- Modules total: 1
- Modules failed to run: 0
- Modules passed: 1
- PASSED: chipsec.modules.common.bios_ts
- Modules failed: 0
- Modules with warnings: 0
- Modules skipped: 0

- **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)
  [0x0] Type = 4  << SMRR memory type
  [12] PhysBase = CB0000 << 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)
  [12] PhysMask = FF8000 << 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 = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[CPU1] SMRR_PHYSBASE = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[CPU2] SMRR_PHYSBASE = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[CPU3] SMRR_PHYSBASE = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[CPU4] SMRR_PHYSBASE = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[CPU5] SMRR_PHYSBASE = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[CPU6] SMRR_PHYSBASE = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[CPU7] SMRR_PHYSBASE = 0x00000000CB000004, SMRR_PHYSMASK = 0x00000000FF800800
[+] OK so far. SMRR range base/mask match on all logical CPUs
[*] Trying to read memory at SMRR base 0xCB00000..
[+] 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 `SmmIsBufferOutsideSmmValid()` 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

- **Website:** http://blackstormsecurity.com

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

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