MALWARE:
ANTI-FORENSICS

BHACK 2018
TOC:

- Obfuscation
- Emulation
- Anti-Reversing
- Anti-VM

- Malware and Security Researcher.
- Speaker at DEFCON USA 2018
- Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics and Rookits.
- Reviewer member of the The Journal of Digital Forensics, Security and Law.
- Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response
- Member of Digital Law and Compliance Committee at OAB/SP
- Instructor at Oracle and (ISC)². Ex-instructor at Symantec.
OBfuscation
The problem is: how to reverse the obfuscation and, from status 4, to reach the position 1?
There are many ways to handle obfuscation and one of them is by using METASM, which:

- It is written in Ruby.
- Works as:
  - a disassembler
  - assembler
  - debugger
  - linker and compiler.
- However, it does not work with intermediate language.
- It is able to separate control and data flows.
OBFUSCATION

✓ Installing METASM is simple:

✓ 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

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

✓ Possible values:
✓ PowerPC
✓ Ia32
✓ x86_64
✓ MachO

This instruction was inserted to make the eax register evaluation easier. 😊
asmcode = mycode.init_disassembler
asmcode.disassemble(0)
 bhack = asmcode.di_at(0).block
puts "\n[\$\$\$] Our BHACK 2018 test code:\n "
puts bhack.list

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

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

asmcode2 = mycode.init_disassembler
asmcode2.disassemble(0)

dd = asmcode2.block_at(0)
final = asmcode2.get_xrefs_x(dd.list.last).first
puts "\n[+] final output: #{final}"
values = asmcode2.backtrace(final, dd.list.last.address, {:log => bt_log = [] , :include_start => true})

bt_log.each{|entry|
  case type = entry.first
  when :start
    entry, expr, addr = entry
    puts "[start] Here is the sequence of expression evaluations #{expr} from 0x#{addr.to_s(16)}\n"
  when :di
    entry, to, from, instr = entry
    puts "[new update] instr #{instr},\n --> updating expression once at gain from #{from} to #{to}\n"
  when :found
    entry, final = entry
    puts "[found] possible value:#{final.first}\n"
  when :up
    entry, to, from, addr_down, addr_up = entry
    puts "[up] addr 0x#{addr_down.to_s(16)} -> 0x#{addr_up.to_s(16)}\n"
  end
}

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

```plaintext
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
```

[###] Our BHACK 2018 test code:
OBfuscation

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

```assembly
[###] push ebx
BHACK 2018 data flow follows below:
* esp => esp-4
* dword ptr [esp] => ebx

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

[###] sub eax, ebx
BHACK 2018 data flow follows below:
* eax => eax-ebx
* eflag_z => (((eax&0xffffffffh)-(ebx&0xffffffffh))&0xffffffffh)==0
* eflag_s => (((eax&0xffffffffh)-(ebx&0xffffffffh))&0xffffffffh)>>1fh)!=0
  * eflag_c => (eax&0xffffffffh)<(ebx&0xffffffffh)
  * eflag_o => (((eax&0xffffffffh)>>1fh)!=0)==(!(((ebx&0xffffffffh)>>1fh)!=0))&(((eax&0xffffffffh)>>1fh)!=0)==(((((eax&0xffffffffh)-(ebx&0xffffffffh))&0xffffffffh)>>1fh)!=0))
```
[+] final output: eax
[start] Here is the sequence of expression evaluations eax from 0x2d
[new update] instr 2ah add eax, edx,
  --> updating expression once again from eax to eax+edx
[new update] instr 29h dec edx,
  --> updating expression once again from eax+edx to eax+edx-1
[new update] instr 26h add edx, 61h,
  --> updating expression once again from eax+edx-1 to eax+edx+60h
[new update] instr 25h inc edx,
  --> updating expression once again from eax+edx+60h to eax+edx+61h
[new update] instr 22h mov edx, ecx,
  --> updating expression once again from eax+edx+61h to eax+ecx+61h
[new update] instr 1dh mov ecx, 62h,
  --> updating expression once again from eax+ecx+61h to eax+0c3h
[new update] instr 16h add eax, 0a3h,
  --> updating expression once again from eax+0c3h to eax+166h
[new update] instr 13h add eax, 38h,
  --> updating expression once again from eax+166h to eax+19eh
[new update] instr 11h add eax, ecx,
  --> updating expression once again from eax+19eh to eax+ecx+19eh
[new update] instr 0eh sub eax, 38h,
  --> updating expression once again from eax+ecx+19eh to eax+ecx+166h
[new update] instr 9h sub eax, 0a3h,
  --> updating expression once again from eax+ecx+166h to eax+ecx+0c3h
[new update] instr 6h sub eax, ebx,
  --> updating expression once again from eax+ecx+0c3h to eax-ebx+ecx+0c3h
[new update] instr 1h mov ebx, 0c3h,
  --> updating expression once again from eax-ebx+ecx+0c3h to eax+ecx
We can realize that the number of instructions is smaller than the initial set because many of them were not useful to compose the final result. Great! 😊
EMULATION
**EMULATION**

- Emulation is another excellent method to solve many practical reverse engineering problems.
- Focusing on an easy approach to handle our small case, it is possible to use uEmu + Keystone Engine. 😊

- First, install the fantastic Keystone Engine as shown below:
  - root@kali:~/Desktop# wget https://github.com/keystone-engine/keystone/archive/0.9.1.tar.gz
  - root@kali:~/programs# cp /root/Desktop/keystone-0.9.1.tar.gz .
  - root@kali:~/programs# tar -zxvf keystone-0.9.1.tar.gz
  - root@kali:~/programs/keystone-0.9.1# apt-get install cmake
✓ root@kali:~/programs/keystone-0.9.1# mkdir build
✓ root@kali:~/programs/keystone-0.9.1# cd build/
✓ root@kali:~/programs/keystone-0.9.1/build# apt-get install time
✓ root@kali:~/programs/keystone-0.9.1/build# ../make-share.sh
✓ root@kali:~/programs/keystone-0.9.1/build# make install
✓ root@kali:~/programs/keystone-0.9.1/build# ldconfig
✓ root@kali:~/programs/keystone-0.9.1/build# tail -3 /root/.bashrc
✓ export PATH=$PATH:/root/programs/phantomjs-2.1.1-linux-x86_64/bin:/usr/local/bin/kstool
✓ export RUBYLIB=$RUBYLIB:~/programs/metasm
✓ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
EMULATION

```
root@kali:~/programs/bhack_2018# more bhack2018.c
#include <stdio.h>
#include <keystone/keystone.h>

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

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

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

All instructions from our original problem.

Creating a Keystone engine for x86.
Assembling our instructions using Keystone.
root@kali:~/programs/bhack_2018# more Makefile

.PHONY: all clean
KEYSTONE_LDFLAGS = -lkeystone -lstdc++ -lm
all:

  ${CC} -o bhack2018 bhack2018.c

  ${KEYSTONE_LDFLAGS}

clean:

  rm -rf *.o bhack2018
EMULATION

root@kali:~/programs/bhack_2018# make

cc -o bhack2018 bhack2018.c -lkeystone -lstdc++ -lm

root@kali:~/programs/bhack_2018# ./bhack2018

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/bhack_2018# ./bhack2018 | xxd -r -p - > output.bin

root@kali:~/programs/bhack_2018# hexdump -C output.bin

00000000 53 bb c3 00 00 00 29 d8 5b 2d a3 00 00 00 83 e8 |S.....)[-......|
00000010 38 01 c8 83 c0 38 05 a3 00 00 00 52 51 b9 62 00 |8....8.....RQ.b.|
00000020 00 00 89 ca 59 42 83 c2 61 4a 01 d0 5a |....YB..aJ..Z|
0000002d
Checking whether our hexcode can be reverted to initial the assembly

```c
#include <inttypes.h>
#include <capstone/capstone.h>

#define CODE
"\x53\xb2\xc3\x00\x00\x00\x00\x29\xd8\x5b\x2d\xa3\x00\x00\x83\nx8\xb1\xc8\x83\xc0\x38\x05\xa3\x00\x00\x00\x00\x52\x51\xb9\x62\x00\x00\x00\x89\xca\x59\x42\x83\xc2\x61\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"PRIx16":\t%s\n", instruction[j].address, instruction[j].mnemonic, instruction[j].op_str);
        }
    } else
        printf("Error: It's happened an error during the disassembling!
"");

    cs_free(instruction, count);

    cs_close(&cs_handle);

    return 0;
}
```
EMULATION

```bash
root@kali:/programs/bhack_2018/capstone# more Makefile

.PHONY: all clean

CAPSTONE_LDFLAGS = -lcapstone -lstdc++ -lm

all:
    ${CC} -o bhack2018_rev bhack2018_rev.c ${CAPSTONE_LDFLAGS}

clean:
    rm -rf *.o bhack2018_rev

root@kali:/programs/bhack_2018/capstone# make

root@kali:/programs/bhack_2018/capstone# make

cc -o bhack2018_rev bhack2018_rev.c -lcapstone -lstdc++ -lm

root@kali:/programs/bhack_2018/capstone#

root@kali:/programs/bhack_2018/capstone# ./bhack2018_rev

0x1: push  ebx
0x2: mov   ebx, 0xc3
0x7: sub   eax, ebx
0x9: pop   ebx
0xa: sub   eax, 0xa3
0xf: sub   eax, 0x38
0x12: add  eax, ecx
0x14: add  eax, 0x38
0x17: add  eax, 0xa3
0x1c: push  edx
0x1d: push  ecx
0x1e: mov   ecx, 0x62
0x23: mov   edx, ecx
0x25: pop   ecx
0x26: inc   edx
0x27: add   edx, 0x61
0x2a: dec   edx
0x2b: add   eax, edx
0x2d: pop   edx
```

It is perfect!
Our instructions perfectly assembled by the Keystone. 😊
According to METASM, but emulating the code, it is the expected result, isn’t it?! 😊
EMULATION

✓ 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
EMULATION

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

![Listing](image)

It is only a simple test with a random shellcode to make sure that MIASM is working (it is not related to our initial obfuscation problem). 😊
Again, we are using a x86 in the MIASM engine.

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

From slide 21, the last instruction is at 2c, so: $0x40000000 + 0x2c$ 😊

MIASM disassembling the code.
It is not a problem, of course. 😊

MIASM perfectly disassembled the binary.
The expected result according to initial values. 😊
Let’s solve the obfuscation problem again, but this time we will try a symbolic solution using MIASM.

root@kali:~/programs/miasm# python

Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)

[GCC 7.3.0] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> from miasm2.analysis.binary import import Container

>>> from miasm2.analysis.machine import import Machine

>>> from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE
EMULATION

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

>>> bhackmach=Machine('x86_32')
>>> aborgesdis=bhackmach.dis_engine(cont.bin_stream)
>>> ourblocks = aborgesdis.dis_multiblock(0)
```

WARNING: not enough bytes in str
WARNING: cannot disasm at 2D
WARNING: not enough bytes in str
WARNING: cannot disasm at 2D
EMULATION

>>> sym = bhackmach.ira()
>>> for block in ourblocks:
...    sym.add_block(block)
...

[<miasm2.ir.ir.IRBlock object at 0x7ff0974a0d70>]
[]

>>> from miasm2.ir.symbexec import SymbolicExecutionEngine
>>> sb = SymbolicExecutionEngine(sym, bhackmach.mn.regs.regs_init)
>>> symbolic_pc = sb.run_at(0, step=True)
EMULATION

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

ESP = ESP_init + 0xFFFFFFFFC
@32[ESP_init + 0xFFFFFFFFC] = EBX_init

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

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

Instr: SUB EAX, EBX
Assignblk:
zf = (EAX + -EBX)?(0x0,0x1)
f = (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
EMULATION

```
EAX = EAX_init + 0xFFFFFFFF3D
cf = (EAX_init ^ ((EAX_init ^ (EAX_init + 0xFFFFFFFF3D)) & (EAX_init ^ 0xC3))) ^ (EAX_init + 0xFFFFFFFF3D) ^ 0xC3)[31:32]
pf = parity((EAX_init + 0xFFFFFFFF3D) & 0xFF)
zf = (EAX_init + 0xFFFFFFFF3D)?(0x0,0x1)
af = (EAX_init ^ (EAX_init + 0xFFFFFFFF3D) ^ 0xC3)[4:5]
ESP = ESP_init + 0xFFFFFFFFC
of = ((EAX_init ^ (EAX_init + 0xFFFFFFFF3D)) & (EAX_init ^ 0xC3))[31:32]
EBX = 0xC3
nf = (EAX_init + 0xFFFFFFFF3D)[31:32]
@32[ESP_init + 0xFFFFFFFFC] = EBX_init

Instr POP   EBX
AssignBlk:
EBX = @32[ESP]
ESP = ESP + 0x4
```

```
EAX = EAX_init + 0xFFFFFFFF3D
cf = (EAX_init ^ ((EAX_init ^ (EAX_init + 0xFFFFFFFF3D)) & (EAX_init ^ 0xC3))) ^ (EAX_init + 0xFFFFFFFF3D) ^ 0xC3)[31:32]
pf = parity((EAX_init + 0xFFFFFFFF3D) & 0xFF)
zf = (EAX_init + 0xFFFFFFFF3D)?(0x0,0x1)
af = (EAX_init ^ (EAX_init + 0xFFFFFFFF3D) ^ 0xC3)[4:5]
of = ((EAX_init ^ (EAX_init + 0xFFFFFFFF3D)) & (EAX_init ^ 0xC3))[31:32]
nf = (EAX_init + 0xFFFFFFFF3D)[31:32]
@32[ESP_init + 0xFFFFFFFFC] = EBX_init
```
EMULATION

I’ve skipped many lines to show the last ones....
Fantastic! The same output as other solutions.
(eax = eax + ecx)
ANTI-REVERSING TECHNIQUES
ANTI-REVERSING

✓ IDA Pro faces few problems with obfuscation, though exist interesting solutions for helping us. Additionally, we always can write IDA processor modules to make the analysis less painful and repeatable.

✓ Unfortunately, protectors such as VMProtect and Themida usually deploy heavy layers of obfuscation techniques. For example, VMProtect holds the following characteristics:

✓ Most of the time, code protected with VMProtect is seen in 64-bit malwares.
✓ Any function from the original malware is removed of the IAT. This is means that IAT shown by pestudio and PE Bear tools is associated to the packer itself.
✓ VMProtect checks the file memory integrity. Therefore, any attempt to change the malware on memory is easily detected.
✓ Instructions (CPU code) are virtualized and transformed into virtual machine instructions (RISC instruction).
The obfuscation is stack based.
The virtualized code is polymorphic, so there are many representations referring the same CPU instruction.
The original code is never entirely decrypted on the memory.
There are many dead and useless codes. Additionally, there is some code reordering using unconditional jumps.
There are many hooks on calls such as LoadString( ) and LdrAccessResource( ) functions (resources are usually encrypted).
It has few anti-debugger and anti-vm tricks.
Calls to IAT functions are replaced by calls at VMProtect section (VMProtect’s IAT).
There are also fake push instructions.
ANTI-REVERSING

Non-disclosure Agreement

Packed with Themida

There is only one imported function
ANTI-REVERSING

After Unpacking – Without Themida

The Import list is longer when viewed in IDA Pro 😊
ANTI-REVERSING

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

  int aborges(int x) -> “Virtualizer” (bytecodes) -> vm_call_1(opcodes, x)

  Fetching bytes, decoding to instructions and dispatching to handlers

  ✓ Handlers, which are independent of one each other, usually configure registers, a encryption key and memory.

  ✓ Additionally, there is one handler by instruction type.

  ✓ The challenge is to revert “virtualized instructions” back to original ones.
ANTI-REVERSING

✔ Furthermore, there are many “obstacles” to circumvent:

✔ Instructions’ operands are encrypted using keys (initializing code) provided by handlers. Furthermore, handlers are “started” by the VM dispatcher.

✔ Code obfuscation: VM instructions are obfuscated

✔ Dead (garbage) code: this technique is implemented by inserting codes whose results will be overwritten in next lines of code or, worse, they won’t be used anymore.

✔ Code-flow graph flattening:

✔ This technique is used to hide the real control flow of a program.

✔ In a general way, the idea is to break the control-flow by removing if-statements and loops, transforming the flow in a series of switch-case statements.
Thus, there is a dispatcher handing over the control flow to handlers, which each handler updates the instruction pointer to the value of the next handler to be executed (virtualize the flow control).

Usually there is an invocation stub, which makes the transition to from native instructions to the virtualized instruction.

This approach presents two reversing problems: the mapping can be from CISC to RISC instruction and the original registers can be turned into special registers from VM.

Because trade-offs, CFG is only applied to specific functions.
#include <stdio.h>

int main (void)
{
    int aborges = 0;
    while (aborges < 30)
    {
        printf ("%d\n", aborges);
        aborges++;
    }

    return 0;
}
ANTI-REVERSING

Disadvantages:

- Loss of performance
- Easy to identify the CFG flattening

```
loading libs

cc = 1

cc != 0

switch(cc)

cc = 1

aborges = 0

cc = 2

cc = 0

cc = 3

break

aborges < 30

aborges++

cc = 2

break

printf

cc = 3

break
```
Instruction

Instruction decoder

DISPATCHER

A, B, C, ... are handlers as shown as, for example, handler_add, handler_sub, handler_push...

Opcodes from a custom instruction set.
ANTI-REVERSING

✓ Remember: obfuscate is transforming a code from X to X’ using any trick, including virtualization.

✓ Identify if the program is virtualized or not.

✓ Identify the input points.

✓ Understand the virtual program counter (known as vpc).

✓ Try to understand each component of the proposed VM.

✓ Different approaches such as Dynamic Symbolic Execution (DSE) and Dynamic Taint Analysis could help us.
ANTI-REVERSING

✓ 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: exchange of one instruction by a set of equivalent instructions.

✓ Abusing inline functions.

✓ Code duplication: different paths coming into the same destination (used by virtualization obfuscators).

✓ Control indirection 1: call instruction → stack pointer update → return skipping some junk code after the call instruction (RET x).

✓ Control indirection 2: malware trigger an exception → registered exception is called → new branch of instructions.
ANTI-REVERSING

✓ Opaque predicate: Although apparently there is an evaluation, 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.

✓ Polymorphism: it is produced by self-modification code (like shellcodes) and by encrypting resources (similar most malware samples).

✓ Anti-debugging: used as irritating techniques to slow the process analysis.

✓ Call stack manipulation: Changes the stack flow by using instruction tricks composed with the ret instruction, making the real ret hidden.

✓ Anti-VM techniques: prevents the malware sample to run inside a VM.
ANTI-REVERSING

Opaque predicate and anti-disassembly techniques

```
.text:00401000 loc_401000: ; CODE XREF: _main+Fp

.text:00401000          push    ebp
.text:00401001          mov     ebp, esp
.text:00401003          xor     eax, eax
.text:00401005          jz      short near ptr loc_40100D+1
.text:00401007          jnz     near ptr loc_40100D+4

.text:0040100D

.text:0040100D loc_40100D: ; CODE XREF: .text:00401005j
.text:0040100D          ; .text:00401007j

.text:0040100D          jmp     near ptr 0D0A8837h
```
# ANTI-REVERSING

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td>call</td>
<td>call + $5</td>
</tr>
<tr>
<td>00401045</td>
<td>pop</td>
<td>pop ecx</td>
</tr>
<tr>
<td>00401046</td>
<td>inc</td>
<td>inc ecx</td>
</tr>
<tr>
<td>00401047</td>
<td>inc</td>
<td>inc ecx</td>
</tr>
<tr>
<td>00401048</td>
<td>add</td>
<td>add ecx, 4</td>
</tr>
<tr>
<td>00401049</td>
<td>add</td>
<td>add ecx, 4</td>
</tr>
<tr>
<td>0040104A</td>
<td>push</td>
<td>push ecx</td>
</tr>
<tr>
<td>0040104B</td>
<td>ret</td>
<td>ret</td>
</tr>
<tr>
<td>0040104C</td>
<td>sub</td>
<td>sub ecx, 6</td>
</tr>
<tr>
<td>0040104D</td>
<td>dec</td>
<td>dec ecx</td>
</tr>
<tr>
<td>0040104E</td>
<td>dec</td>
<td>dec ecx</td>
</tr>
<tr>
<td>0040104F</td>
<td>jmp</td>
<td>jmp 0x401320</td>
</tr>
</tbody>
</table>

- **Call stack manipulation:**

  - Do you know what’s happening here? 😊
Is it possible to deobfuscate virtualized instructions? Yes, it is possible using reverse recursive substitution (similar -- not equal -- to backtracking feature from Metasm).

Additionally, symbolic equation system is another good approach (again...., Metasm and MIASM!).

There are many good plugins such as Code Unvirtualizer, VMAttack, VMSweeper, and so on, which could be used to handle simple virtualization problems.

Some evolution of the instruction virtualizers has risen using simple and efficient concepts of cryptography as Substitution Boxes (S-Boxes).
ANTI-DEBUGGING
TECHNIQUE
There is a powerful anti-debugging technique named “int 2d”, which trigger an exception.

However, the trick is: when a debugger is attached, the exception triggered by the int 2d instruction is handled by debugger.

However when a debugger is not attached, is the malware that handle the exception (for example, using the combination try / catch from C++) 😊.

Therefore, the malware only will trigger + catch the exception whether there is not an attached debugger.

Additionally, there is little side effect here: when the exception happen, the exception is triggered, but the next 1 byte following the int 2D is skipped. This side effect can be used to deceive the analyst.
ANTI-VM TECHNIQUES

```c
static BOOL DebuggerPresent = TRUE;

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

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

    PVOID hnd1 = AddVectoredExceptionHandler(1, VectoredHandler);
    DebuggerPresent = TRUE; /* At this point is our indication that the exception was consumed! */
    abint2d(); /* Calling the INT 2D instruction from abint2d*/
    RemoveVectoredExceptionHandler(hnd1); /* Remove the Vectored Exception Handler*/
    return DebuggerPresent;
}
```
Exception_Pointer structure contains an exception record with a machine-independent description of an exception and a context record with a machine-dependent description of the processor context at the time of the exception.
ANTI-VM TECHNIQUES

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

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

There is NOT a debugger attached. :)

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

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

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

There is a DEBUGGER attached. :<

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

- Anti-VM techniques detecting VMWare (checking I/O port communication), VirtualBox, Parallels, SeaBIOS emulator, QEMU emulator, Bochs emulator, QEMU emulator, Hyper-V, Innotek VirtualBox, sandboxes (Cuckoo).

There are dozens of techniques that could be used for detection Vmware sandboxes:

- Examining the registry (`OpenSubKey()` function) to try to find entries related to tools installed in the guest (`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VirtualMachine\Guest\Parameters`).

- Using WMI to query the `Win32_BIOS` management class to interact with attributes from the physical machine.
ANTI-VM TECHNIQUES

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

- The **ManagementClass** class represents a **Common Information Model (CIM)** management class.

- The **Win32_BIOS** WMI class represents the attributes of BIOS and members of this class enable you to access WMI data using a specific WMI class path.

- **GetInstances()** acquires a collection of all instances of the class.

- **GetEnumerator()** returns the enumerator (**IEnumerator**) to the collection.

- **IEnumerator.Current()** returns the same object.

- **IEnumerator.MoveNext()** advances the enumerator to the next element of the collection.
**ANTI-VM TECHNIQUES**

- **Physical host:**
  ```
c:\Users\Administrador\source\repos\Test_VM\Test_VM\bin\Debug> Test_VM.exe
  ```
  **Attributes:**
  - **Version:** DELL - 6222004
  - **SerialNumber:** D5965S1
  - **OperatingSystem:** 0
  - **Manufacturer:** Dell Inc.

- **Guest virtual machine:**
  ```
  E:\> Test_VM.exe
  ```
  **Attributes:**
  - **Version:** LENOVO - 6040000
  - **SerialNumber:** VMware-56 4d 8d c3 a7 c7 e5 2b-39 d6 cc 93 bf 90 28 2d
  - **OperatingSystem:** 0
  - **Manufacturer:** Phoenix Technologies LTD
ANTI-VM TECHNIQUES

❖ PhysicalHost:

C:\> TestVM_2.exe
This program is NOT RUNNING in a VMware guest

❖ VirtualMachine:

E:▷ TestVM_2.exe
This program is running in a VMware guestVMware Virtual Platform
Manufacturer: VMware, Inc.
ANTI-VM TECHNIQUES

```csharp
namespace TestVM_3
{
    class Program
    {
        static void Main(string[] args)
        {
            ManagementClass tempClass =
                new ManagementClass("Win32_TemperatureProbe");
            ManagementObjectCollection tempInstance =
                tempClass.GetInstances();
            foreach (ManagementObject aborges in tempInstance)
            {
                string buffer = aborges.GetPropertyValue("CurrentReading").ToString();
            }
            Console.WriteLine("Temperature:\t" + buffer);
        }
    }
}
```

This code does NOT work. 😞

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

HTTP://WWW.BLACKSTORMSECURITY.COM
ANTI-VM TECHNIQUES

Connect

Namespace: root\cimv2
Connection: Using: IWBemLocator (Namespaces)
Returning: IWbemServices

Credentials
User:
Password:
Authority:

Locale
How to interpret empty password
- NULL
- Blank

Impersonation level
- Identify
- Impersonate
- Delegate

Authentication level
- None
- Packet
- Connection
- Packet integrity
- Call
- Packet privacy

Windows Management Instrumentation Tester

Namespace: root\cimv2

IWbemServices

- Enum Classes...
- Enum Instances...
- Open Namespace...
- Edit Context...
- Create Class...
- Create Instance...
- Query...
- Create Refresher...
- Open Class...
- Open Instance...
- Notification Query...
- Delete Class...
- Delete Instance...
- Execute Method...

Method Invocation Options
- Asynchronous
- Synchronous
- Semisynchronous
- Use NextAsync (enum. only)
- Enable All Privileges
- Use Amended Qualifiers
- Direct Access on Read Operations

Timeout (msec., -1 for infinite)
- 10
- 5000

Query

Enter Query
select * from Win32_TemperatureProbe

Query Result

WQL: select * from Win32_TemperatureProbe

1 objects
max. batch: 1
Done

Win32_TemperatureProbe.DeviceID="root\cimv2 0"
Now we know why our prior code didn’t worked. 😊
ANTI-VM TECHNIQUES

```
using System;
using System.Management;

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

            foreach (ManagementObject aborges in tempinstance)
            {
                try
                {
                    if (!string.IsNullOrWhiteSpace(aborges.GetPropertyValue("Status").ToString()))
                    {
                        string buffer = aborges.GetPropertyValue("Status").ToString();
                        Console.WriteLine("\nStatus: " + buffer + " Thus, the program is running in a physical host!");
                    }
                }
                catch (NullReferenceException e)
                {
                    Console.WriteLine("\nSomething Wrong Happened!", e);
                }
            }
            Console.WriteLine("This program IS RUNNING in a virtual machine!");
        }
    }
}```
ANTI-VM TECHNIQUES
There is **not support** for acquiring temperature data in virtual machines. Therefore, malwares are able to know whether they are running on virtual machines or not. 😊
ANTI-VM TECHNIQUES

❖ Physical Host:

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

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

❖ Virtual Machine

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

This program IS RUNNING in a virtual machine!
ANTI-VM TECHNIQUES

✓ Additional informations about detection of virtualized environments:

✓ Win32_Processor class / ProcessorId attribute to acquire NumberOfCores (most of the time, there is only one core in virtualized systems).

✓ Win32_NetworkAdapterConfiguration WMI class represents the attributes and behaviors of a network adapter. Therefore, it is possible to use this class to acquire MAC addresses to detect the possible virtual machines.

✓ MAC Addresses:

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

✓ IP_ADAPTER_INFO structure
✓ Malloc(IP_ADAPTER_INFO )
✓ GetAdaptersInfo( )
ANTI-VM TECHNIQUES

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

- CreateToolhelp32Snapshot()
- Process32First()
- Process32Next()
ANTI-VM TECHNIQUES

✓ Many malwares authors have tried to detect Cuckoo sandboxes. As Cuckoo saves the hooking information (fs:[tls_hooking_information]) and the returns address in the TLS, so it is possible to check the Cuckoo’s presence by:

✓ reading the address of this hook information (int hook_1), which is usually saved at fs:0x44.

✓ adding the size of the saved hooking information to its address (location_1 = hook_1 + size).

✓ adding the location_1 variable to the usual extra space used by Cuckoo for saving the hook information (location_2 = location_1 + reserved space).

✓ Of course, if there is something (saved return address) within this address range (location_1 to location_2), so the malware can be running on Cuckoo.
ANTI-VM TECHNIQUES

- Detect if the process is running under WoW
- RegOpenKeyEx( ) (it depends on the first step)
- RegCloseKey( )
ANTI-VM TECHNIQUES

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

Installed VMware Drivers:

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

- GetWindowsDirectory( )
- PathCombine( )
- GetFileAttributes( )
ANTI-VM TECHNIQUES

✓ There are dozens of other checks:

✓ mouse (inactivity): `GetCursorPos()`

✓ memory (small size): `GlobalMemoryStatusEx()` (Retrieves information about the system's current usage of both physical and virtual memory).

✓ disk drivers characteristics (strings like wmare, vbox and so on):

  ✓ The `SetupDiGetClassDevs()` function returns a handle to a device information set that contains requested device information elements for a local computer.

  ✓ `SetupDiEnumDeviceInfo()` function returns a `SP_DEVINFO_DATA` structure that specifies a device information element in a device information set.

  ✓ `SetupDiGetDeviceRegistryProperty()` function retrieves a specified Plug and Play device property (SPDRP_HARDWAREID)
Malwares have checked the existence of important tools, which could be running in the system:

- tcpview.exe
- autorunsc.exe
- regmon.exe
- proexp.exe
- ProcessHacker.exe
- procmon.exe
- autoruns.exe

- ImmunityDebugger.exe
- Wireshark.exe
- idaq.exe
- idaq64.exe
- ollydbg.exe
- windbg.exe
- x64dbg.exe

Malwares have also checked the existence of these same tools in the PATH variable and standard installation locations.
THANK YOU FOR ATTENDING MY LECTURE.

- Malware and Security Researcher.
- Speaker at DEFCON USA 2018
- Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics and Rookits.
- Reviewer member of the The Journal of Digital Forensics, Security and Law.
- Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response
- Member of Digital Law and Compliance Committee at OAB/SP
- Instructor at Oracle and (ISC)2. Ex-instructor at Symantec.

Twitter: @ale_sp_brazil
@blackstormsecbr

Website: http://blackstormsecurity.com

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

E-mail: alexandreborges@blackstormsecurity.com