MODERN MALWARE: OBFUSCATION AND EMULATION

DEF CON CHINA 1.0 (2019)

by Alexandre Borges
Malware and Security Researcher.
Speaker at DEFCON USA 2018
Speaker at HITB 2019 Amsterdam
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 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

Agenda:

- Introduction
- Anti-reversing
- METASM
- MIASM
- TRITON
- Radare2 + MIASM
- DTRACE on Windows
- Anti-VM
- Conclusion
INTRODUCTION
Every single day we handle malware samples that use several known packers such as ASPack, Armadillo, Petite, FSG, UPX, MPRESS, NSPack, PECompact, WinUnpack and so on. For most of them, it is easy to write scripts to unpack them.

We also know the main API functions, which are used to create and allocate memory such as:

- VirtualAlloc/Ex()
- HeapCreate() / RtlCreateHeap()
- HeapReAlloc()
- GlobalAlloc()
- RtlAllocateHeap()

Additionally, we know how to unpack them using debuggers, breakpoints and dumping unpacked content from memory. Furthermore, pe-sieve from Hasherezade is excellent.

When we realize that the malware use some customized packing techniques, it is still possible to dump it from memory, fix the ImageAddress field using few lines in Python and its respective IAT using impscan plugin to analyze it in IDA Pro:

- export VOLATILITY_PROFILE=Win7SP1x86
- python vol.py -f memory.vmem procdump -p 2096 -D --memory (to keep slack space)
- python vol.py -f memory.vmem impscan --output=idc -p 2096
entry:

msg "Program to dump modules containing executables."
msg "You must be at EP before continuing"
bc       // Clear existing breakpoints
bphwc    // Clear existing hardbreakpoints
bp VirtualAlloc     // Set up a breakpoint at VirtualAlloc
erun      // run and pass all first exceptions to the application

core:

sti       // Single-step
sti       // Single-step
sti       // Single-step
sti       // Single-step
sti       // Single-step
sti       // Single-step
find cip,"C2 1000"  // find the return point of VirtualAlloc
bp $result  // set a breakpoint
erun  // run and pass all first exceptions to the application
cmp eax,0  // test if eax (no allocated memory) is equal to zero
je pcode  // jump to pcode label
bpm eax,0,x  // set executable memory breakpoint and restore it once hit.
erun  // run and pass all first exceptions to the application

// try to find if there is the “This program” string within the module’s memory.
findall $breakpointexceptionaddress,"546869732070726F6772616D"
cmp $result,0  // check if there isn’t any hit
je pcode  // jump to pcode label
$dumpaddr = mem.base($breakpointexceptionaddress)  // find the memory base.
=size = mem.size($breakpointexceptionaddress)  // find the size of memory base.
savedata :memdump:,$dumpaddr,$size  // dump the segment.
msgyn “Memory dumped! Do you want continue?”  // show a dialog
cmp $result,1  // check your choice
je scode  // jump to scode label
bc  // clear existing breakpoints
bphwc  // clear existing hardware breakpoints
ret  // exit
pcode:

msgyn "There isn't a PE file! Do you want continue?"
cmp $result,0  // check if we don’t want continue
je final
sti          // single step.
erun         // run and pass all first exceptions to the application
jmp core     // jump to core label

scode:

msg "Let's go to next dump"  // shows a message box
erun         // run and pass all first exceptions to the application
jmp core     // jump to core label

final:

bc            // clear existing breakpoints
bphwc         // clear existing hardware breakpoints
ret           // exit
ANTI-REVERSING
Obfuscation aims to protect software of being reversed, intellectual property and, in our case, malicious code too. 😊 Honestly, obfuscation does not really protect the program, but it can 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 we analyze common userland malware, droppers written in VBA and Powershell, so it mightn’t seem to be a big deal.

We can use IDA Pro SDK to write plugins to extend the IDA Pro functionalities, analyze some code and data flow and even automatizing unpacking of strange malicious files.

Additionally, if you are facing problems to analyze a modified MBR, so you could even write a loader to load the MBR structure and analyze it in IDA Pro. 😊

Unfortunately, there are packers and protectors such as VMprotect, Themida, Arxan and Agile .NET that use modern obfuscation techniques, so making the procedure of reversing a code very complicated.
✓ Most protectors have 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 provide string encryption.

✓ They protect and 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 is stack based, so it is hard to handle virtualized code statically.

✓ Virtualized code is polymorphic, so there are many representations referring the same CPU instruction.

✓ There are also fake push instructions.

✓ There are many dead and useless codes.

✓ There is some code reordering using unconditional jumps.

✓ All obfuscators use code flattening.

✓ Packers have few anti-debugger and anti-vm tricks. However, few months ago, I saw a not so common anti-virtual machine trick based on temperature (more about it later).
int defcon(int x) 

"Virtualizer" (bytecodes) 

vm_call_1(opcodes, x) 

Fetches bytes, decodes them to instructions and dispatches them 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
Few interesting concepts:

- **Fetching**: the instruction to be executed by Virtual Machine is fetched.

- **Decoding**: the target x86 instruction is decoded using rules from Virtual Machine (remember: usually, the architecture is usually based on RISC instructions)

- **Dispatcher**: Once the handler is determined, so jump to the suitable handler. Dispatchers could be made by a jump table or switch case structure.

- **Handler**: In a nutshell, a handler is the implementation of the Virtual Machine instruction set.
A, B, C, ... are handlers such as handler_add, handler_sub, handler_push...

RVA → RVA + process base address and other tasks.

Opcodes from a custom instruction set.

Instructions are stored in an encrypted format.
The diagram illustrates a function pointer table (likely encrypted) and its relationship with encrypted instructions. The table contains instructions such as `vm_add`, `vm_sub`, `vm_xor`, `vm_push`, `vm_pop`, etc., each corresponding to a function pointer and a handler.

The function pointer table is structured as follows:

- Opcode 1 is mapped to function pointer 1, which is mapped to handler 1.
- Opcode 2 is mapped to function pointer 2, which is mapped to handler 2.
- Opcode 3 is mapped to function pointer 3, which is mapped to handler 3.
- Opcode 4 is mapped to function pointer 4, which is mapped to handler 4.
- Opcode 5 is mapped to function pointer 5, which is mapped to handler 5.
- Opcode 6 is mapped to function pointer 6, which is mapped to handler 6.
- Opcode 7 is mapped to function pointer 7, which is mapped to handler 7.

The encrypted instructions are shown in the top row, labeled `encr_1`, `encr_2`, `encr_3`, etc., and their corresponding decrypted instructions are shown in the bottom row. The indexes are numbered from 1 to n.

The diagram demonstrates the process of recovering and decrypting functions.
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.

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 ➔ 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.

Opaque predicate: Although apparently there is an evaluation (conditional jump: jz/jnz), the result is always evaluated to true (or false), which means an unconditional jump. Thus, there is a dead branch.

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

Polymorphism: it is produced by self-modification code (like shellcodes) and by encrypting resources (similar most malware samples).
It is quick to create a **simple IDA Pro plugin**. Download the IDA SDK from https://www.hex-rays.com/products/ida/support/download.shtml (likely, you will need a professional account). Copy it to a folder (idasdk695/) within the IDA Pro installation directory.

Create a project in Visual Studio 2017 (File ➔ New ➔ Create Project ➔ Visual C++ ➔ Windows Desktop ➔ Dynamic-Link Library (DLL)).

Change few project properties as shown in this slide and next ones.
✓ Include the "__NT__;__IDP__" in Processor Definitions and change Runtime Library to "Multi-threaded" (MT) (take care: it is NOT /MTd).
✓ Add `ida.lib` (from `C:\Program Files (x86)\IDA 6.95\idasdk695\lib\x86_win_vc_32`) to Additional Dependencies and its folder to Additional Library Directories.

✓ Add “/EXPORT:PLUGIN” to Additional Options.
Don’t forget necessary headers.

Initialization function.

Make the plugin available to this idb and keep the plugin loaded in memory.

Clean-up tasks.

Function to be called when user activates the plugin.

Simple (and incomplete) URL regex.
for (int x = 0; x < get_strlist_qty(); x++) {
  get_strlist_item(x, &strinfo);
  if (strinfo.length < sizeof(defcon)) {
    get_many_bytes(strinfo.ea, defcon, strinfo.length);
    
    ea = 0;
    ea = find_text(strinfo.ea, 0, 0, s, SEARCH_REGEX);
    if (ea == strinfo.ea) {
      msg("Address 0x%% - URL %d: %s\n", strinfo.ea, urlcount, defcon);
      urlcount++;
    }
  }
}
return;

char IDAP_comment[] = "The simplest possible plugin";
char IDAP_help[] = "DEFCON plugin";
char IDAP_name[] = "DEFCON plugin";
char IDAP_hotkey[] = "ALT-X";

plugin_t PLUGIN = {
  IDP_INTERFACE_VERSION, 0,
  IDAP_init, IDAP_term, IDAP_run,
  IDAP_comment, IDAP_help, IDAP_name, IDAP_hotkey
};
Hello DEFCON CHINA! We love IDA Pro :)

Address 0x99f990d8 - URL 1: ntp2.usno.navy.mil
Address 0x99f990eb - URL 2: ntp.adc.am
Address 0x99f990f6 - URL 3: tock.usask.ca
Address 0x99f99104 - URL 4: ntp.crifo.org
Address 0x99f99112 - URL 5: ntp1.arnes.si
Address 0x99f99120 - URL 6: ntp.ucsd.edu
Address 0x99f9912d - URL 7: ntp.duckcorp.org
Address 0x99f9913e - URL 8: www.nist.gov
Address 0x99f9914b - URL 9: clock.isc.org
Address 0x99f99159 - URL 10: time.windows.com
Address 0x99f9916a - URL 11: time2.one4vision.de
Address 0x99f9917e - URL 12: time.cerias.purdue.edu
Address 0x99f99195 - URL 13: clock.fihn.net

URLs found within this malicious driver. 😊
✓ IDA processor modules continue being the one of best approach to handle virtualized packers.

✓ Please, you should remember on few important points (as mentioned by Ilfak from Hex-Rays) about how to write an IDA processor modules:

- decodes instructions and fill structures with the result (ana.cpp)
- processes the commands decoded by analyser (amu.cpp)
- creates cross-references.
- tracks the register content.
- tracks the register content.
- Writes the output a handled output containing prefix, comments and xrefs (out.cpp)

✓ The IDA Pro SDK documentation and samples are always great. 😊
```c
#include <stdio.h>

int main (void)
{
    int aborges = 0;
    while (aborges < 30)
    {
        printf("%d\n", aborges);
        aborges++;
    }
    return 0;
}
```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_4= dword ptr -4

push rbp
mov rbp, rsp
sub rsp, 10h
mov [rbp+var_4], 0
jmp short loc_675

loc_675:
cmp [rbp+var_4], 1Dh
jle short loc_65B

loc_65B:
mov eax, [rbp+var_4]
mov esi, eax
lea rdi, format ; "\%d\n"
mov eax, 0
call _printf
add [rbp+var_4], 1

mov eax, 0
leave
retn
main endp
Disavantages:

- Loss of performance
- Easy to identify the CFG flattening
The obfuscator-llvm is an excellent project to be used for code obfuscation. To install it, it is recommended to add a swap file first (because the linkage stage):

- `fallocate -l 8GB /swapfile`
- `chmod 600 /swapfile`
- `mkswap /swapfile`
- `swapon /swapfile`
- `swapon --show`
- `apt-get install llvm-4.0`
- `apt-get install gcc-multilib` (install gcc lib support to 32 bit)
- `git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git`
- `mkdir build ; cd build/`
- `cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_INCLUDE_TESTS=OFF ../obfuscator/`
- `make -j7`

Possible usages:

- `./build/bin/clang alexborges.c -o alexborges -mllvm -fla`
- `./build/bin/clang alexborges.c -m32 -o alexborges -mllvm -fla`
- `./build/bin/clang alexborges.c -o alexborges -mllvm -fla -mllvm -sub`
Prologue and initial assignment

Main dispatcher
Main blocks from the program
General overview of the obfuscated code
```c
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     signed int v3; // eax@5
4     int v4; // eax@8
5     __int64 v6; // [rsp+0h] [rbp-20h]@0
6     signed int v7; // [rsp+14h] [rbp-Ch]@1
7     signed int v8; // [rsp+18h] [rbp-8h]@1
8
9     v8 = 0;
10    v7 = 2118196251;
11    while ( v7 != -803096687 )
12    {
13        if ( v7 == 900748651 )
14            {
15                v4 = printf("%d\n", (unsigned int)v8++, envp, v6, 7317960004152066048LL);
16                v7 = 2118196251;
17                LODWORD(v6) = v4;
18            }
19        else
20            {
21                HIDWORD(v6) = v7 - 2118196251;
22                if ( v7 == 2118196251 )
23                    {
24                        v3 = -803096687;
25                        if ( v8 < 30 )
26                            v3 = 900748651;
27                            v7 = v3;
28                    }
29            }
30    }
31    return 0;
32}
```
Simple opaque predicate and anti-disassembly technique

```
.text:00401000  loc_401000: ; CODE XREF: _main+Fp
.push ebp
.mv ebp, esp
.xor eax, eax
.jz short near ptr loc_40100D+1
.jnz near ptr loc_40100D+4

.text:0040100D  loc_40100D: ; CODE XREF: .text:00401005j
.text:0040100D  ; .text:00401007j
.text:0040100D  jmp near ptr 0D0A8837h
```
Decrypted shellcode

Decryption instructions 😊
Call stack manipulation:

Do you know what’s happening here? 😊
METASM
(keystone + capstone + unicorn)
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`
```ruby
#!/usr/bin/env ruby
#
require "metasm"
include Metasm

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

entry:
  push ebx
  mov ebx, 0xb9
  sub eax, ebx
  pop ebx
  sub eax, 0x55
  sub eax, 0x32
  add eax, ecx
  add eax, 0x50
  add eax, 0x37
  push edx
  push ecx
  mov ecx, 0x49
  mov edx, ecx
  pop ecx
  inc edx
  add edx, 0x70
  dec edx
  add eax, edx
  pop edx
  jmp eax
EOB
```

This instruction was inserted to make the eax register evaluation easier. 😊
initialize and disassemble code since beginning (start).

list the assembly code.

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

initialize the backtracking engine.

determines which is the final instruction to walk back from there.😊
Backtracking from the last instruction.

Show only the effective instructions, which really can alter the final result.
Remember: this is our obfuscated code. 😊

```assembly
0 push ebx
1 mov ebx, 0b9h
6 sub eax, ebx
8 pop ebx
9 sub eax, 55h
0ch sub eax, 32h
0fh add eax, ecx
11h add eax, 50h
14h add eax, 37h
17h push edx
18h push ecx
19h mov ecx, 49h
1eh mov edx, ecx
20h pop ecx
21h inc edx
22h add edx, 70h
25h dec edx
26h add eax, edx
28h pop edx
29h jmp eax
```
DEF CON China data flow follows below:

Processing: eax == eax-ebx
  Processing: eflag_z == (((eax&0xffffffff)-(ebx&0xffffffff))&0xffffffff)==0
  Processing: eflag_s == (((eax&0xffffffff)-(ebx&0xffffffff))&0xffffffff)>>1fh)!=0
  Processing: eflag_c == (eax&0xffffffff)<(ebx&0xffffffff)
  Processing: eflag_o == (((eax&0xffffffff)>>1fh)!=0)==(!(((ebx&0xffffffff)>>1fh)!=0))
  &((&(((eax&0xffffffff)>>1fh)!=0)==0)!=0))=(((((eax&0xffffffff)-(ebx&0xffffffff))&0xffffffff)>>1fh)!=0))

pop ebx

DEF CON China data flow follows below:
  Processing: esp == esp+4
  Processing: ebx == dword ptr [esp]

sub eax, 55h

DEF CON China data flow follows below:
  Processing: eax == eax-55h
  Processing: eflag_z == (((eax&0xffffffff)-((55h)&0xffffffff))&0xffffffff)==0
  Processing: eflag_s == (((eax&0xffffffff)-((55h)&0xffffffff))&0xffffffff)>>1fh)!=0
  Processing: eflag_c == (eax&0xffffffff)<((55h)&0xffffffff)
  Processing: eflag_o == (((eax&0xffffffff)>>1fh)!=0)==(!(((55h)&0xffffffff)>>1fh)!=0))
  &((&(((eax&0xffffffff)>>1fh)!=0)==0)!=0))=(((((eax&0xffffffff)-((55h)&0xffffffff))&0xffffffff)>>1fh)!=0))

sub eax, 32h

DEF CON China data flow follows below:
  Processing: eax == eax-32h
  Processing: eflag_z == (((eax&0xffffffff)-((32h)&0xffffffff))&0xffffffff)==0
  Processing: eflag_s == (((eax&0xffffffff)-((32h)&0xffffffff))&0xffffffff)>>1fh)!=0
  Processing: eflag_c == (eax&0xffffffff)<((32h)&0xffffffff)
  Processing: eflag_o == (((eax&0xffffffff)>>1fh)!=0)==(!(((32h)&0xffffffff)>>1fh)!=0))
  &((&(((eax&0xffffffff)>>1fh)!=0)==0)!=0))=(((((eax&0xffffffff)-((32h)&0xffffffff))&0xffffffff)>>1fh)!=0))
[+] final output: eax

[start] Here is the sequence of expression evaluations eax from 0x29

[new update] instruction 26h add eax, edx,
---> updating expression once again from eax to eax+edx
[new update] instruction 25h dec edx,
---> updating expression once again from eax+edx to eax+edx-1
[new update] instruction 22h add edx, 70h,
---> updating expression once again from eax+edx-1 to eax+edx+6fh
[new update] instruction 21h inc edx,
---> updating expression once again from eax+edx+6fh to eax+edx+70h
[new update] instruction 1eh mov edx, ecx,
---> updating expression once again from eax+edx+70h to eax+ecx+70h
[new update] instruction 19h mov ecx, 49h,
---> updating expression once again from eax+ecx+70h to eax+0b9h
[new update] instruction 14h add eax, 37h,
---> updating expression once again from eax+0b9h to eax+0f0h
[new update] instruction 11h add eax, 50h,
---> updating expression once again from eax+0f0h to eax+140h
[new update] instruction 0fh add eax, ecx,
---> updating expression once again from eax+140h to eax+ecx+140h
[new update] instruction 0ch sub eax, 32h,
---> updating expression once again from eax+ecx+140h to eax+ecx+10eh
[new update] instruction 9 sub eax, 55h,
---> updating expression once again from eax+ecx+10eh to eax+ecx+0b9h
[new update] instruction 6 sub eax, ebx,
---> updating expression once again from eax+ecx+0b9h to eax-ebx+ecx+0b9h
[new update] instruction 1 mov ebx, 0b9h,
---> updating expression once again from eax-ebx+ecx+0b9h to eax+ecx

Game over. 😊
The **effective** instructions are:

1. mov ebx, 0b9h
2. sub eax, ebx
3. sub eax, 55h
4. sub eax, 32h
5. add eax, ecx
6. add eax, 50h
7. add eax, 37h
8. mov ecx, 49h
9. mov edx, ecx
10. inc edx
11. add edx, 70h
12. dec edx
13. add eax, edx

Output originated from backtracing_log.select command (in reverse)
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 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
- 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 DEFCON "push ebx; mov ebx, 0xb9; sub eax, ebx; pop ebx; sub eax, 0x55; sub eax, 0x32; add eax, ecx; add eax, 0x50; add eax, 0x37; push edx; push ecx; mov ecx, 0x49; mov edx, ecx; pop ecx; inc edx; add edx, 0x70; 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, DEFCON, 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;
}
```bash
root@kali:~/programs/defcon#
root@kali:~/programs/defcon# more Makefile
.PHONY: all clean

KEYSTONE_LDFLAGS = -lkeystone -lstdc++ -lm

all:
    ${CC} -o defcon2019 defcon2019.c ${KEYSTONE_LDFLAGS}

clean:
    rm -rf *.o defcon2019

root@kali:~/programs/defcon#
root@kali:~/programs/defcon# make

cc -o defcon2019 defcon2019.c -lkeystone -lstdc++ -lm

root@kali:~/programs/defcon#
root@kali:~/programs/defcon# ./defcon2019

53 bb b9 00 00 00 29 d8 5b 83 e8 55 83 e8 32 01 c8 83 c0 50 83 c0 37 52 51 b9 49 00 00 00 89 ca 59 42 83 c2 70 4a 01 d0 5a

root@kali:~/programs/defcon#
root@kali:~/programs/defcon# ./defcon2019 | xxd -r -p - > defcon2019.bin

root@kali:~/programs/defcon#
root@kali:~/programs/defcon# hexdump -C defcon2019.bin

00000000 53 bb b9 00 00 00 29 d8 5b 83 e8 55 83 e8 32 01 |S......).[U...Z.
00000010 c8 83 c0 50 83 c0 37 52 51 b9 49 00 00 00 89 ca |...P..7RQ.I.....|
00000020 59 42 83 c2 70 4a 01 d0 5a |YB..pJ..Z|
00000029

root@kali:~/programs/defcon# _
```
To install Capstone: apt-get install libcapstone3 libcapstone-dev 😊

```c
#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\xc0\x50\x83\xc0\x37\x52\x51\xb9\x49\x00\x00\x89\xc9\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%s
", instruction[j].address, instruction[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;
}
```
root@kali:~/programs/defcon/capstone# more Makefile

.PHONY: all clean

CAPSTONE_LDFLAGS = -lcapstone -lstdc++ -lm

all:
  ${CC} -o defcon2019_rev defcon2019_rev.c ${CAPSTONE_LDFLAGS}

clean:
  rm -rf *.o defcon2019_rev

root@kali:~/programs/defcon/capstone# make

cc -o defcon2019_rev defcon2019_rev.c -lcapstone -lstdc++ -lm

root@kali:~/programs/defcon/capstone# ./defcon2019_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. 😊
IDA Pro confirms our disassembly task. 😊
- set up before running uEmu

This result confirms our previous conclusion.

- 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.
# git clone https://github.com/unicorn-engine/unicorn.git
# cd unicorn ; ./make.sh
# ./make.sh install

```c
#include <unicorn/unicorn.h>
#include <string.h>

// Our code to be emulated.

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

// Emulation start address and a simple macro.
#define ADDR 0x1000000
#define MIN(x, y) (x < y ? x : y)

// Hook the instruction execution.
static void hook_code(uc_engine *uc, uint64_t address, uint32_t size, void *userdata) {
    int r_eip;
    int r_eax;
    int r_ebx;
    int r_ecx;
    int r_edx;

    uint8_t instr_size[16];
```
printf("\nTracing instruction at 0x%lx , instruction size = 0x%lx\n", address, size);

uc_reg_read(uc, UC_X86_REG_EIP, &r_eip);
uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
uc_reg_read(uc, UC_X86_REG_EBX, &r_ebx);
uc_reg_read(uc, UC_X86_REG ECX, &r_ecx);
u uc_reg_read(uc, UC_X86_REG EDX, &r_edx);

// Print the initial values of registries.
printf("\n>> EIP=0x%x ", r_eip);
printf(" | EAX=0x%x ", r_eax);
printf(" | EBX=0x%x ", r_ebx);
printf(" | ECX=0x%x ", r_ecx);
printf(" | EDX=0x%x ", r_edx);
printf("\n>> Executed hex code: ");

size = MIN(sizeof(instr_size), size);
if (!uc_mem_read(uc, address, instr_size, size)) {
    uint32_t i;
    for (i=0; i<size; i++) {
        printf("%x ", instr_size[i]);
    }
    printf("\n");
}

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

// Declare and initialize few variables

c_engine *uc;
uc_hook traceinstr;
uc_err err;

// Set up the initial registry values.
// We have to set up the ESP register for emulating PUSH/POP instructions.

int r_eax = 0x4;
int r_ebx = 0x0;
int r_ecx = 0x7;
int r_edx = 0x0;
int r_esp = ADDR + 200000;

printf("\nInitial register values: \n");

printf("\n>> EAX = %x ", r_eax);
printf("\n>> EBX = %x ", r_ebx);
printf("\n>> ECX = %x ", r_ecx);
printf("\n>> EDX = %x ", r_edx);

printf("\n\nOur emulated code is: \n");

// We are emulating a 32-bit application in x86 emulator, so initialize the emulator in X86-32bit mode :)
// If we wished to emulate in a x64 emulator, so we would use UC_MODE_64.

err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
if (err != UC_ERR_OK) {
  printf("A fail to use uc_open() has occured and the error returned is: %u\n", err);
  return -1;
}
// We are reserving 4MB memory for this emulation. Additionally, UC_PROT_ALL means: RWX.

uc_mem_map(uc, ADDR, 4 * 1024 * 1024, UC_PROT_ALL);

// write machine code to be emulated to memory

if (uc_mem_write(uc, ADDR, DEFCON_CODE, sizeof(DEFCON_CODE) - 1)) {
    printf("It has happened a fail during the write emulation code to memory!\n");
    return -1;
}

// We need to initialize the machine registers

uc_reg_write(uc, UC_X86_REG_EAX, &r_eax);
uc_reg_write(uc, UC_X86_REG_EBX, &r_ubx);
uc_reg_write(uc, UC_X86_REG_ECX, &r_ecx);
uc_reg_write(uc, UC_X86_REG_EDX, &r_edx);
uc_reg_write(uc, UC_X86_REG_ESP, &r_esp);

// uc: hook handle ; traceinstr: reference to uc_hook ; UC_HOOK_CODE: hook type ;
// hook_code: callback function

uc_hook_add(uc, &traceinstr, UC_HOOK_CODE, hook_code, NULL, 1, 0);
// Start the emulation engine and emulate code in infinite time (first zero below) & unlimited instructions (second zero below).

err = uc_emu_start(uc, ADDR, ADDR + sizeof(DEFCON_CODE) - 1, 0, 0);
if (err) {
    printf("The uc_emu_start() function has failed with error r
returning %u: %s\n", err, uc_strerror(err));
}

// Finally, print out the final registers values.

printf("\nThe final CPU registers contain the following content: \n\n");

uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
uc_reg_read(uc, UC_X86_REG_EBX, &r_ebx);
uc_reg_read(uc, UC_X86_REG ECX, &r_ecx);
uc_reg_read(uc, UC_X86_REG EDX, &r_edx);
printf(">>> EAX = 0x%x", r_eax);
printf("\n>>> EBX = 0x%x", r_ebx);
printf("\n>>> ECX = 0x%x", r_ecx);
printf("\n>>> EDX = 0x%x\n\n", r_edx);

uc_close(uc);

return 0;
Initial register values:

>> EAX = 4
>> EBX = 0
>> ECX = 7
>> EDX = 0

Our emulated code is:

Tracing instruction at 0x1000000 , instruction size = 0x1

>> EIP=0x1000000 | EAX=0x4 | EBX=0x0 | ECX=0x7 | EDX=0x0
>> Executed hex code: 53

Tracing instruction at 0x1000001 , instruction size = 0x5

>> EIP=0x1000001 | EAX=0x4 | EBX=0x0 | ECX=0x7 | EDX=0x0
>> Executed hex code: bb b9 0 0 0

Tracing instruction at 0x1000006 , instruction size = 0x2

>> EIP=0x1000006 | EAX=0x4 | EBX=0xb9 | ECX=0x7 | EDX=0x0
>> Executed hex code: 29 d8

Tracing instruction at 0x1000008 , instruction size = 0x1

>> EIP=0x1000008 | EAX=0xffffffffb | EBX=0xb9 | ECX=0x7 | EDX=0x0
>> Executed hex code: 5b
Tracing instruction at 0x1000021, instruction size = 0x1

>> EIP=0x1000021 | EAX=0xffffffff | EBX=0x0 | ECX=0x7 | EDX=0x49
>> Executed hex code: 42

Tracing instruction at 0x1000022, instruction size = 0x3

>> EIP=0x1000022 | EAX=0xffffffff | EBX=0x0 | ECX=0x7 | EDX=0x4a
>> Executed hex code: 83 c2 70

Tracing instruction at 0x1000025, instruction size = 0x1

>> EIP=0x1000025 | EAX=0xffffffff | EBX=0x0 | ECX=0x7 | EDX=0xba
>> Executed hex code: 4a

Tracing instruction at 0x1000026, instruction size = 0x2

>> EIP=0x1000026 | EAX=0xffffffff | EBX=0x0 | ECX=0x7 | EDX=0xb9
>> Executed hex code: 1 d0

Tracing instruction at 0x1000028, instruction size = 0x1

>> EIP=0x1000028 | EAX=0xb | EBX=0x0 | ECX=0x7 | EDX=0xb9
>> Executed hex code: 5a

The final CPU registers contain the following content:

>>> EAX = 0xb
>>> EBX = 0x0
>>> ECX = 0x7
>>> EDX = 0x0
MIASM
✓ **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 also is able to emulate 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 texinfo texi2html`
  ✓ `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 00000000000001070 (0)
  INFO : generate graph file
  INFO : generate intervals
  [0x1070 0x10A2]
  INFO : total lines 0

✓ **(testing MIASM)** xdot graph_execflow.dot
```
from miasm2.analysis.binary import Container
from miasm2.analysis.machine import Machine
from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE

with open("defcon2019.bin") as fdesc:
    cont=Container.from_stream(fdesc)

machine=Machine('x86_32')
mdis=container.dis_engine(cont.bin_stream)
ourblocks = mdis.dis_multiblock(0)
for block in ourblocks:
    print block

jitter = machine.jitter("llvm")
jitter.init_stack()
s = open("defcon2019.bin").read()
run_addr = 0x40000000
jitter.cpu.EAX=3
jitter.cpu.ECX=6
jitter.vm.add_memory_page(run_addr, PAGE_READ | PAGE_WRITE, s)

def code_sentinelle(jitter):
    jitter.run = False
    jitter.pc = 0
    return True

jitter.add_breakpoint(0x40000028, code_sentinelle)
jitter.push_uint32_t(0x40000028)
jitter.jit.log_regs = True
jitter.jit.log_mn = True
jitter.init_run(run_addr)
jitter.continue_run()

open('defcon2019_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 emulation 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.
Disassembling our code (again) 😊
Our proposed code. 😊
Get the IRA converter.

Initialize and run the Symbolic Execution Engine.
```python
>>> symbolic_pc = symb.run_at(0, step=True)
Instr PUSH EBX
Assignblk:
ESP = ESP + -0x4
@32[ESP + -0x4] = EBX

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

Instr MOV EBX, 0xb9
Assignblk:
EBX = 0xb9

ESP = ESP_init + 0xffffffffc
EBX = 0xb9
@32[ESP_init + 0xffffffffc] = EBX_init

Instr SUB EAX, EBX
Assignblk:
zf = (EAX + -EBX)?(0x0,0x1)
nf = (EAX + -EBX)[31:32]
lf = 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
```
The same conclusion from our previous tests. 😊
TRITON
TRITON

✓ It can be downloaded from https://triton.quarkslab.com/

✓ Triton offers a C/C++/Python interface provides:

✓ dynamic symbolic execution
✓ run time registry information and memory modification
✓ taint engine
✓ Z3 interface to handle contraints
✓ snapshot engine (it is not necessary to restart the program every time, but only restores memory and register states)
✓ access to Pin funtions
✓ symbolic fuzzing
✓ gather code coverage

✓ Supports x86 and x64 architecture.
✓ Triton supports:

✓ **symbolic** execution mode:

  ✓ emulates **instruction effects**.
  ✓ allows us to **emulate only part of the program** (excellent for analyzing branches).

✓ **concolic** execution mode:

  ✓ allows us to **analyze the program only from start**.

✓ **Taint analysis** is amazing because we are able to using in fuzzing tasks to know what registers and memory address are “affected” by the user data input. 😊

✓ During **Virtual Machine’s decoding**, it is interesting to distinguish which instructions are related to user input and which are not. 😊
Installing **Triton without Pin** (Ubuntu 19):

- `apt-get install libboost-all-dev`
- `apt-get install libpython-dev`
- `apt-get install libcapstone-dev`
- **Take care:** DO NOT install `libz3-dev`. If this package is already installed, so remove it.
- `git clone https://github.com/Z3Prover/z3`
- `cd z3/`
- `python scripts/mk_make.py`
- `cd build/`
- `make`
- `make install`
- `git clone https://github.com/JonathanSalwan/Triton.git`
- `cd Triton/`
- `mkdir build`
- `cd build/`
- `cmake ..`
- `make -j install` (my recommendation: 8 GB RAM + 8 GB swapfile)
✓ Installing **Triton with Pin** (Ubuntu 19):

✓ Install the same packages from last slide.
✓ Install Z3 as shown in the last slide.
✓ **wget**
✓ tar zxvf pin-2.14-71313-gcc.4.4.7-linux.tar.gz
✓ cd pin-2.14-71313-gcc.4.4.7-linux/source/tools
✓ **git clone** https://github.com/JonathanSalwan/Triton.git
✓ cd Triton/
✓ mkdir build
✓ cd build
✓ **cmake -DPINTOOL=on -DKERNEL4=on ..**
✓ **make**
✓ cd ..
✓ **./build/triton ./src/examples/pin/ir.py /usr/bin/host** (only to test the installation).
from __future__ import print_function
from triton import TritonContext, ARCH, Instruction, MemoryAccess, CPUSize, OPERAND, REG
import sys

# We define the code to be handled and symbolic executed
mycode = [
    (0x400000, b"\x53"), # push ebx
    (0x400001, b"\xbb\xb9\x00\x00\x00"), # mov ebx, 0x89
    (0x400006, b"\x29\xd8"), # sub eax, ebx
    (0x400008, b"\x5b"), # pop ebx
    (0x400009, b"\xe8\xe8\x55"), # sub eax, 0x55
    (0x40000c, b"\xe8\xe8\x32"), # sub eax, 0x32
    (0x40000f, b"\x01xc8"), # add eax, ecx
    (0x400011, b"\xe8\xc0\x50"), # add eax, 0x50
    (0x400014, b"\xe8\xc0\x37"), # add eax, 0x37
    (0x400017, b"\x52"), # push edx
    (0x400018, b"\x51"), # push ecx
    (0x400019, b"\xb9\x49\x00\x00\x00"), # mov ecx, 0x49
    (0x40001e, b"\xe8\xa9\xca"), # mov edx, ecx
    (0x400020, b"\x59"), # pop ecx
    (0x400021, b"\x42"), # inc edx
    (0x400022, b"\xe8\xc2\x70"), # add edx, 0x70
    (0x400025, b"\x4a"), # dec edx
    (0x400026, b"\x01\xda0"), # add eax, edx
    (0x400028, b"\x5a"), # pop edx
    (0x400029, b"\xff\xe0"), # jmp eax
]
if __name__ == '__main__':

    # Set the context for Triton functions
    context = TritonContext()

    # Set the architecture. In our case, we are using x86 32-bit
    context.setArchitecture(ARCH.X86)

    for (addr, opcode) in mycode:
        # Build an instruction object.
        instruction = Instruction()

        # Setup the opcode
        instruction.setOpcode(opcode)

        # Setup start address
        instruction.setAddress(addr)

        # Process our code
        context.processing(instruction)

        print('------------------------------------------')
        print('The current IP: ', instruction)
        pc = context.getRegisterAst(context.registers.eip).evaluate()
        print('The next IP is: ', hex(pc))
        print('------------------------------------------

        # Display each instruction, determine the operation type and show opcode in
        # Display each instruction, determine the operation type and show opcode in
        print('>>> %s' % instruction)

        print('

        print('Is a memory read? : ', instruction.isMemoryRead())
        print('Is a memory write? : ', instruction.isMemoryWrite())
        print('------------------------------------------
        print('}
for op_entry in instruction.getOperands():
    print('%s %s (op_entry))
    if op_entry.getType() == OPERAND.MEM:
        print('segment :', op_entry.getSegmentRegister())
        print('base : %s (op_entry.getBaseRegister())
        print('index : %s (op_entry.getIndexRegister())
        print('disp : %s (op_entry.getDisplacement())
        print('scale : %s (op_entry.getScale())

print('')

# Display each one of the symbolic expressions
for expression in instruction.getSymbolicExpressions():
    print('	', expression)

print()
print()
print('Registers information')
print('************************************************************************
for k, v in list(context.getSymbolicRegisters().items()):
    print(context.getRegister(k), v)

print()
print('Summary Memory information')
print('************************************************************************
for k, v in list(context.getSymbolicMemory().items()):
    print(hex(k), v)

print()
 sys.exit(0)
This is an educational way to show how to find the hexadecimal representation for each instruction.

However, there are much better ways to do it by opening the binary on IDA Pro, Radare2, Ghidra or even using distorm3.
The current IP: 0x400000: push ebx
The next IP is: 0x400001

>>> 0x400000: push ebx

--------------------------------------------------------
Is a memory read? : False
Is a memory write? : True
--------------------------------------------------------

ebx:32 bv[31..0]

(define-fun ref!0 () (_ BitVec 32) (bvszero 32)); Stack alignment

(define-fun ref!1 () (_ BitVec 8) (((_ extract 31 24) (_ bv0 32))) ; Byte reference - PUSH operation

(define-fun ref!2 () (_ BitVec 8) (((_ extract 23 16) (_ bv0 32))) ; Byte reference - PUSH operation

(define-fun ref!3 () (_ BitVec 8) (((_ extract 15 8) (_ bv0 32))) ; Byte reference - PUSH operation

(define-fun ref!4 () (_ BitVec 8) (((_ extract 7 0) (_ bv0 32))) ; Byte reference - PUSH operation

(define-fun ref!5 () (_ BitVec 32) (concat (((_ extract 31 24) (_ bv0 32))) (((_ extract 23 16) (_ bv0 32))) (((_ extract 15 8) (_ bv0 32))) (((_ extract 7 0) (_ bv0 32)))) ; Temporary concatenation reference - PUSH operation

(define-fun ref!6 () (_ BitVec 32) (_ bv4194305 32)); Program Counter

byte by byte 😊
Registers information

- Save registers:
  - esp:32 bv[31..0] (define-fun ref!112 () (_ BitVec 32) (bvadd ref!79 (_ bv4 32))) ; Stack alignment
  - cf:1 bv[0..0] (define-fun ref!105 () (_ BitVec 1) ( (_ extract 31 31) (bvxor (bvand ref!52 ref!96) (bvand (bvxor (bv
or ref!52 ref!96) ref!103) (bvxor ref!52 ref!96)))) ) ; Carry flag
  - eip:32 bv[31..0] (define-fun ref!114 () (_ BitVec 32) ref!103) ; Program Counter
  - of:1 bv[0..0] (define-fun ref!106 () (_ BitVec 1) ( (_ extract 31 31) (bvand (bvxor ref!52 (bvn
ot ref!96)) (bvxor ref
!52 ref!103)))) ; Overflow flag
  - eax:32 bv[31..0] (define-fun ref!103 () (_ BitVec 32) (bvadd ref!52 ref!96)) ; ADD operation
  - sf:1 bv[0..0] (define-fun ref!108 () (_ BitVec 1) ( (_ extract 31 31) ref!103)) ; Sign flag
  - ebx:32 bv[31..0] (define-fun ref!17 () (_ BitVec 32) (concat ref!1 ref!2 ref!3 ref!4)) ; POP o
  - peration
  - zf:1 bv[0..0] (define-fun ref!109 () (_ BitVec 1) (ite (= ref!103 (_ bv0 32)) (_ bv1 1) (_ bv0
1)))) ; Zero flag
  - ecx:32 bv[31..0] (define-fun ref!78 () (_ BitVec 32) (concat ref!68 ref!69 ref!70 ref!71)) ; P
  - OP operation
  - af:1 bv[0..0] (define-fun ref!104 () (_ BitVec 1) (ite (= (_ bv16 32) (bvand (_ bv16 32) (bvox
r ref!103 (bvxor ref!5
2 ref!96)))) (_ bv1 1) (_ bv0 1))) ; Adjust flag
  - edx:32 bv[31..0] (define-fun ref!111 () (_ BitVec 32) (concat ref!61 ref!62 ref!63 ref!64)) ; P
  - OP operation
  - pf:1 bv[0..0] (define-fun ref!107 () (_ BitVec 1) (bvxor (bvxor (bvxor (bvxor (bvxor (bvxor (b
vxor (bvxor (_ bv1
1)( (_ extract 0 0) (bvls
h
r ( (_ e
xtract 7 0) ref!103) (_ bv0 8)))) ( (_ extract 0 0) (bvls
h
r ( (_ e
xtract 7 0) ref!103) (_ bv2 8)))) ( (_ ex
tract 0 0) (bvls
h
r ( (_ e
xtract 7 0) ref!103) (_ bv3 8)))) ( (_ extract 0 0) (bvls
h
r ( (_ e
xtract 7 0) ref!103) (_ bv4 8)))) ( (_ ex
tract 0 0) (bvls
h
r ( (_ e
xtract 7 0) ref!103) (_ bv5 8)))) ( (_ extract 0 0) (bvls
h
r ( (_ e
xtract 7 0) ref!103) (_ bv6
8)))) ( (_ extract 0 0) (bvls
h
r ( (_ e
xtract 7 0) ref!103) (_ bv7 8)))) ) ; Parity flag
```python
# Define the code to be emulated
mycode = {
    0x400000: b'\x53',  # push ebx
    0x400001: b'\xbb\xb9\x00\x00\x00\x00',  # mov ebx, 0xB9
    0x400006: b'\x29\xd8',  # sub eax, ebx
    0x400008: b'\x5b',  # pop ebx
    0x400009: b'\x83\xe8\x55',  # sub eax, 0x55
    0x40000c: b'\x83\xe8\x32',  # sub eax, 0x32
    0x40000f: b'\x01\xc8',  # add eax, ecx
    0x400011: b'\x83\xc0\x50',  # add eax, 0x50
    0x400014: b'\x83\xc0\x37',  # add eax, 0x37
    0x400017: b'\x52',  # push edx
    0x400018: b'\x51',  # push ecx
    0x400019: b'\xb9\x49\x00\x00\x00',  # mov ecx, 0x49
    0x40001e: b'\x89\xca',  # mov edx, ecx
    0x400020: b'\x59',  # pop ecx
    0x400021: b'\x42',  # inc edx
    0x400022: b'\x83\xc2\x70',  # add edx, 0x70
    0x400025: b'\x4a',  # dec edx
    0x400026: b'\x01\xd0',  # add eax, edx
    0x400028: b'\x5a',  # pop edx
    0x400029: b'\xff\xe0',  # jmp eax
}

# Define the context object to be applied the Triton functions
context = TritonContext()
```
# This function emulates the code.
def defcon(pc):
    while pc in mycode:
        # Build an instruction
        instruction = Instruction()

        # Setup the opcode
        instruction.setOpcode(mycode[pc])

        # Setup start address
        instruction.setAddress(pc)

        # Process the opcodes
        context.processing(instruction)

        # Display the instruction
        print('Curr pc:', instruction)

        # Set the IP to next instruction and update the some registers
        pc = context.getRegisterAst(context.registers.eip).evaluate()
        eax = context.getRegisterAst(context.registers.eax).evaluate()
        ebx = context.getRegisterAst(context.registers.ebx).evaluate()
        ecx = context.getRegisterAst(context.registers.ecx).evaluate()
        edx = context.getRegisterAst(context.registers.edx).evaluate()

        print('Next pc: ', hex(pc))
        print('Next eax: ', hex(eax))
        print('Next ebx: ', hex(ebx))
        print('Next ecx: ', hex(ecx))
        print('Next edx: ', hex(edx))

    return
```python
# This function initializes the context memory. EAX and ECX was randomly chosen.
def startCtx():
    context.setConcreteRegisterValue(context.registers.esp, 0x7fffffff)
    context.setConcreteRegisterValue(context.registers.ebp, 0x7fffffff)
    context.setConcreteRegisterValue(context.registers.eax, 0x2)
    context.setConcreteRegisterValue(context.registers ebx, 0x0)
    context.setConcreteRegisterValue(context.registers.ecx, 0x7)
    context.setConcreteRegisterValue(context.registers.edx, 0x0)
    return

if __name__ == '__main__':
    # Set the architecture. In our case, we have chosen x86 32-bit.
    context.setArchitecture(ARCH.X86)

    # Align the memory
    context.enableMode(MODE.ALIGNED_MEMORY, True)

    # Define the entry point address
    entrypoint = 0x400000

    # Set the memory context
    startCtx()

    # Run the emulation
    defcon(entrypoint)

    sys.exit(0)
```
root@ubuntu19:~/pin214/source/tools/Triton/src/examples/python# python_defcon_sym_2.py

Cur  ip: 0x400000: push ebx
Next  ip: 0x400001
Next eax: 0x2
Next ebx: 0x0
Next ecx: 0x7
Next edx: 0x0

Cur  ip: 0x400001: mov ebx, 0xb9
Next  ip: 0x400006
Next eax: 0x2
Next ebx: 0xb9
Next ecx: 0x7
Next edx: 0x0

Cur  ip: 0x400006: sub eax, ebx
Next  ip: 0x400008
Next eax: 0xffffffff49
Next ebx: 0xb9
Next ecx: 0x7
Next edx: 0x0

Cur  ip: 0x400028: pop edx
Next  ip: 0x400029
Next eax: 0x9
Next ebx: 0x0
Next ecx: 0x7
Next edx: 0x0

Cur  ip: 0x400029: jmp eax
Next  ip: 0x9
Next eax: 0x9
Next ebx: 0x0
Next ecx: 0x7
Next edx: 0x0
RADARE2 + MIASM
root@kali:~/programs/defcon# r2 -b 32 defcon2019.bin
-- EIP = 0x41414141
[0x00000000]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00000000]> ec comment green
[0x00000000]> e asm.emu=true
[0x00000000]> pdf

(fcn) fcn.00000000 41
fcn.00000000 ()

0x00000000                   ; esp=0xfffffffffffffff
0x00000001                   ; 185 ; ebx=0x0b9
0x00000002                   ; ebx=0xffffffff
0x00000003                   ; esp=0x10000000
0x00000004                   ; 'U' ; eax=0xfffffffff2 ; of=0x0 ; sf=0
0x00000005                   ; '2' ; eax=0xffffffffc0 ; of=0x0 ; sf=0
0x00000006                   ; eax=0xfffffffff8 ; of=0x0 ; sf=0
0x00000007                   ; 'P' ; eax=0xffffffff10 ; of=0x0 ; sf=0
0x00000008                   ; '7' ; eax=0xfffffffff7 ; of=0x0 ; sf=0
0x00000009                   ; esp=0xfffffffffffffff
0x0000000a                   ; esp=0xfffffffff8
0x0000000b                   ; 'I' ; 73 ; ecx=0x49
0x0000000c                   ; edx=0x49
0x0000000d                   ; ecx=0xffffffff ; esp=0xffffffff
0x0000000e                   ; edx=0x4a ; of=0x0 ; sf=0x0 ; zf=0x0
0x0000000f                   ; 'p' ; edx=0xba ; of=0x0 ; sf=0x0 ; zf=0x0
0x00000010                   ; edx=0xb9 ; of=0x0 ; sf=0x0 ; zf=0x0
0x00000011                   ; eax=0x100000000 ; of=0x0 ; sf=0x0
0x00000012                   ; edx=0xffffffff ; esp=0x100000000

ESIL comment

=0xl -> 0xb9bb ; zf=0x0 ; pf=0x1 -> 0xb9bb ; cf=0x1 -> 0xb9bb
0x00000008                   ; 5b
0x00000009                   ; 83e855
0x0000000a                   ; sub eax, 0x55

x1 -> 0xb9bb ; zf=0x0 ; pf=0x0 ; cf=0x0
0x0000000c                   ; 83e832
0x0000000d                   ; sub eax, 0x32

x1 -> 0xb9bb ; zf=0x0 ; pf=0x1 -> 0xb9bb ; cf=0x0
0x0000000f                   ; 01c8
0x00000010                   ; add eax, ecx

0xb9bb ; zf=0x0 ; cf=0x0 ; pf=0x1 -> 0xb9bb
0x00000011                   ; 83c050
0x00000012                   ; add eax, 0x50

x1 -> 0xb9bb ; zf=0x0 ; cf=0x0 ; pf=0x0
0x00000014                   ; 83c037
0x00000015                   ; add eax, 0x37

x1 -> 0xb9bb ; zf=0x0 ; cf=0x0 ; pf=0x1 -> 0xb9bb
0x00000017                   ; 52
0x00000018                   ; push ecx
0x00000019                   ; mov ecx, 0x49
0x0000001a                   ; mov edx, ecx
0x0000001b                   ; pop ecx
0x0000001c                   ; inc edx
0x0000001d                   ; pf=0x0
0x00000022                   ; 83c270
0x00000023                   ; add edx, 0x70
f=0x0 ; cf=0x0 ; pf=0x0
0x00000025                   ; 4a
0x00000026                   ; 01d0
0x00000027                   ; add eax, edx
zf=0x1 -> 0xb9bb ; cf=0x1 -> 0xb9bb ; pf=0x1 -> 0xb9bb
0x00000028                   ; 5a
0x00000029                   ; pop edx

DEF CON CHINA 1.0 (2019) 88
- **aer**: handle ESIL registers (set and show)
- **aes**: perform emulated debugger step
- **aecu**: continue until address
R2M2 bridges the radare2 and miasm2 communities: radare2 being the graphical interface of miasm2, and miasm2 simplifying the implementation of new architectures.

How to install it?

- `apt-get install docker`
- `git clone https://github.com/radare/radare2.git`
- `cd radare2/`
- `sys/install.sh`
- **Install MIASM**
- `pip install cffi`
- `pip install jinja2`
- `docker pull guedou/r2m2`
- `docker run --rm -it -e 'R2M2_ARCH=x86_32' guedou/r2m2 bash`

- `[r2m2@fd5662d151e4 ~]$ pwd`
- (another terminal) `docker ps -a`
- (another terminal) `docker cp /root/defcon2019.bin fd5662d151e4:/home/r2m2/defcon2019.bin`

- `[r2m2@fd5662d151e4 ~]$ export R2M2_ARCH=x86_32`
- `[r2m2@fd5662d151e4 ~]$ r2 -A -b 32 -a r2m2 defcon2019.bin`
$ r2 -A -b 32 -a r2m2 defcon2019.bin

/home/r2m2/miasm/miasm/expression/expression.py:924: UserWarning: warnings.warn('DEPRECATION WARNING: use exprmem.ptr instead of exprmem

[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] find and analyze function preludes (aap)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Finding xrefs in noncode section with anal.in = 'io.maps
[x] Analyze value pointers (aav)
[x] Value from 0x00000000 to 0x00000029 (aav)
[x] 0x00000000-0x00000029 in 0x0-0x29 (aav)
[Warning: No SN reg alias for current architecture.
[x] Emulate code to find computed references (aae)
[WARNING: r_reg_get: assertion 'reg && name' failed (line 279)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
   -- Warning, your trial license is about to expire.

[0x00000000]>
[0x00000000]>
[0x00000000]>
[0x00000000]>
[0x00000000]>
[0x00000000]>
ec comment yellow
[0x00000000]>
e asm.emu=true
[0x00000000]>
pd 20
(fcn) fcn.00000000 41
fcn.00000000 (int32_t arg_4h);
    ; arg int32_t arg_4h @ esp+0x4
0x00000000  53  PUSH   EBX
0x00000011  bbb9000000  MOV  EBX, 0xb9
0x00000006  29d8  SUB  EAX, EBX
0x00000008  5b  POP   EBX
0x00000009  83e855  SUB  EAX, 0x55
0x0000000c  83e832  SUB  EAX, 0x32
0x0000000f  01c8  ADD  EAX, ECX
0x00000011  83c050  ADD  EAX, 0x50
0x00000014  83c037  ADD  EAX, 0x37
0x00000017  52  PUSH  EDX
0x00000018  51  PUSH  ECX
0x00000019  b949000000  MOV  ECX, 0x49
0x0000001e  89ca  MOV  EDX, ECX
0x00000020  59  POP   ECX
0x00000021  42  INC   EDX
0x00000022  83c270  ADD  EDX, 0x70
0x00000025  4a  DEC   EDX
0x00000026  01d0  ADD  EAX, EDX
0x00000028  5a  POP   EDX
0x00000029  ffff  // buffer too long //
DTRACE on WINDOWS
DTrace is a dynamic tracing framework, which is very efficient and famous on Solaris operating system.

Dtrace was initially written by Mike Shapiro, Adam Leventhal and Brian Cantrill at Sun Microsystems. Although they were developing DTrace since 2003, it was only introduced in Solaris 10 03/05.

It is used to get a real time overview of a system in user and kernel mode. Furthermore, it can be used to understand how application and systems are behaving.

Few months ago, DTrace was ported to Windows: https://github.com/opendtrace/opendtrace/tree/windows

DTrace is could be summarized as a set of probes (sensors) scattered over the key point in the kernel. Thus, every time that a probe is “activated”, it is possible to register and understand the application behavior.

Using DTrace makes easier to trace the profile of a process and the system, find which system calls are “called”, how many bytes are written/ read by a process, file opened by a process, tracing the sequence of called system calls and so on.
✓ DTrace scripts are written in D language (similar to awk).

✓ Probe names are described by the following syntaxe:

```
provider:module:function:name
```

where:

✓ **provider**: library of probes used to instrument an area of the system. On Windows, the existing providers are syscall, etw, profile, pid and dtrace.

✓ **module**: kernel module where we find the probe.

✓ **function**: function containing the probe.

✓ **name**: specific name or description of the target probe.

✓ **Key concepts:**

✓ **predicates**: user defined conditions.

✓ **actions**: tasks that are run when a probe fires.

✓ **aggregations**: coalesce data using aggregation functions.
To install DTrace:

- Windows 10 x64 (build 18342 or later) from Windows Insider Program.
- bcdedit.exe /set dtrace on
- Download DTrace package:
  http://download.microsoft.com/download/B/D/4/BD4B95A5-0B61-4D8F-837C-F889AAD8DAA2/DTrace.amd64.msi
- _NT_SYMBOL_PATH=srv*C:\symbols*https://msdl.microsoft.com/download/symbols
- Reboot the system.
- Open a command prompt as administrator.
- If you are using fbt (function boundary tracing), so it is necessary to attach the WinDbg and boot the Windows in debug mode.
<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dtrace</td>
<td></td>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>dtrace</td>
<td></td>
<td>END</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>dtrace</td>
<td></td>
<td>ERROR</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```bash
C:\> dtrace -V
Dtrace: Sun D 1.13

C:\> dtrace -l | grep -v "syscall" | grep -v "etw"

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dtrace</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>dtrace</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>dtrace</td>
<td></td>
</tr>
<tr>
<td>2997</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>2998</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>2999</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3001</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3002</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3003</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3004</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3005</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3006</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3007</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3008</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3009</td>
<td>profile</td>
<td></td>
</tr>
<tr>
<td>3044</td>
<td>profile</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION  NAME
BEGIN
END
ERROR
profile-97
profile-199
profile-499
profile-997
profile-1999
profile-4001
profile-4999
tick-1
tick-10
tick-100
tick-500
tick-1000
tick-5000
tick-5sec
```
```
C:\>dtrace -Fn "syscall:::entry /execname==\"notepad.exe\"/ { @num[probefunc] = count(); }"
dtrace: description 'syscall:::entry ' matched 464 probes

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>NtCreateFile</td>
<td>1</td>
</tr>
<tr>
<td>NtQueryAttributesFile</td>
<td>1</td>
</tr>
<tr>
<td>NtQueryInformationFile</td>
<td>1</td>
</tr>
<tr>
<td>NtQueryValueKey</td>
<td>1</td>
</tr>
<tr>
<td>NtWriteFile</td>
<td>1</td>
</tr>
<tr>
<td>NtEnumerateKey</td>
<td>2</td>
</tr>
<tr>
<td>NtQueryInformationToken</td>
<td>2</td>
</tr>
<tr>
<td>NtSetInformationFile</td>
<td>2</td>
</tr>
<tr>
<td>NtSetInformationProcess</td>
<td>2</td>
</tr>
<tr>
<td>NtSetTimer2</td>
<td>2</td>
</tr>
<tr>
<td>NtWaitForWorkViaWorkerFactory</td>
<td>2</td>
</tr>
<tr>
<td>NtTraceEvent</td>
<td>4</td>
</tr>
<tr>
<td>NtClearEvent</td>
<td>6</td>
</tr>
<tr>
<td>NtOpenKeyEx</td>
<td>6</td>
</tr>
<tr>
<td>NtOpenEvent</td>
<td>7</td>
</tr>
<tr>
<td>NtQueryKey</td>
<td>10</td>
</tr>
<tr>
<td>NtAssociateWaitCompletionPacket</td>
<td>12</td>
</tr>
<tr>
<td>NtSetInformationThread</td>
<td>16</td>
</tr>
<tr>
<td>NtAlpcSendWaitReceivePort</td>
<td>30</td>
</tr>
<tr>
<td>NtOpenFile</td>
<td>135</td>
</tr>
<tr>
<td>NtQueryDirectoryFileEx</td>
<td>135</td>
</tr>
<tr>
<td>NtClose</td>
<td>138</td>
</tr>
<tr>
<td>NtQueryInformationProcess</td>
<td>138</td>
</tr>
<tr>
<td>NtCallbackReturn</td>
<td>616</td>
</tr>
</tbody>
</table>
```
### List of Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NtDeviceIoControlFile</td>
<td>32</td>
</tr>
<tr>
<td>NtDuplicateObject</td>
<td>40</td>
</tr>
<tr>
<td>NtFreeVirtualMemory</td>
<td>50</td>
</tr>
<tr>
<td>NtAllocateVirtualMemory</td>
<td>56</td>
</tr>
<tr>
<td>NtQueryInformationThread</td>
<td>75</td>
</tr>
<tr>
<td>NtFindAtom</td>
<td>163</td>
</tr>
<tr>
<td>NtSetTimerResolution</td>
<td>187</td>
</tr>
<tr>
<td>NtQuerySystemInformation</td>
<td>202</td>
</tr>
<tr>
<td>NtCreateEvent</td>
<td>328</td>
</tr>
<tr>
<td>NtClose</td>
<td>381</td>
</tr>
<tr>
<td>NtQueryInformationProcess</td>
<td>396</td>
</tr>
<tr>
<td>NtClearEvent</td>
<td>428</td>
</tr>
<tr>
<td>NtAlertThreadByThreadId</td>
<td>604</td>
</tr>
<tr>
<td>NtWaitForAlertByThreadId</td>
<td>604</td>
</tr>
<tr>
<td>NtSetIoCompletionEx</td>
<td>684</td>
</tr>
<tr>
<td>NtAssociateWaitCompletionPacket</td>
<td>1020</td>
</tr>
<tr>
<td>NtSetIoCompletion</td>
<td>1050</td>
</tr>
<tr>
<td>NtDelayExecution</td>
<td>1215</td>
</tr>
<tr>
<td>NtFlushProcessWriteBuffers</td>
<td>1335</td>
</tr>
<tr>
<td>NtRemoveIoCompletionEx</td>
<td>1702</td>
</tr>
<tr>
<td>NtReadFile</td>
<td>2175</td>
</tr>
<tr>
<td>NtWriteFile</td>
<td>2242</td>
</tr>
<tr>
<td>NtSetEvent</td>
<td>2824</td>
</tr>
<tr>
<td>NtWaitForSingleObject</td>
<td>4319</td>
</tr>
<tr>
<td>NtRemoveIoCompletion</td>
<td>8600</td>
</tr>
</tbody>
</table>
It is possible to use a different type of provider named “fbt” (function boundary tracing), which tracks the sequence of system calls being executed through the NTFS in the kernel.

The “fbt” provider only it is available when there is kernel debugger attached to the Windows 10.
dtrace: description 'fbt:ntfs::' matched 7752 probes
CPU FUNCTION
  0  ->  NtfsFsdDispatchWait
    0  ->  memset
    0  <-  memset
    0  ->  NtfsFsdDispatchSwitch
        0  ->  NtfsInitializeTopLevelIrp
        0  <-  NtfsInitializeTopLevelIrp
        0  ->  memset
        0  <-  memset
        0  ->  NtfsInitializeIrpContextInternal
        0  <-  NtfsInitializeIrpContextInternal
        0  ->  NtfsUpdateIrpContextWithTopLevel
        0  <-  NtfsUpdateIrpContextWithTopLevel
        0  ->  NtfsPreRequestProcessingExtend
        0  <-  NtfsPreRequestProcessingExtend
        0  ->  NtfsCommonQueryInformation
            0  ->  NtfsAcquireExclusiveFcb
            0  <-  NtfsAcquireExclusiveFcb
            0  ->  TxfSetupTransactionContextFromCcb
            0  <-  TxfSetupTransactionContextFromCcb
            0  ->  NtfsQueryNameInfo
Your Windows Insider Build ran into a problem and needs to restart. We're just collecting some error info, and then you can restart.

100% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: DRIVER_IRQL_NOT_LESS_OR_EQUAL

What failed: traceext.sys
Traceext.sys: exposes functionality used by DTrace to tracing.
Windows 10 Kernel Version 18362 MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffff802`21e17000 PsLoadedModuleList = 0xfffff802`2225a290
Debug session time: Sun Apr 28 19:11:07.480 2019 (UTC - 7:00)
System Uptime: 0 days 2:40:06.813
It is extremely easy writing malware samples using anti-VM techniques designed to detect VMWare (checking I/O port communication), VirtualBox, Parallels, SeaBIOS emulator, QEMU emulator, Bochs emulator, QEMU emulator, Hyper-V, Innotek VirtualBox, sandboxes (Cuckoo).

Furthermore, 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.
- We have already know every single anti-VM technique around the world and all of them are documented.
- Most current techniques use WMI and it is quick to write a C# program using them.
using System;
using System.Management;

namespace Test_VM
{
    class Program
    {
        static void Main(string[] args)
        {
            ManagementClass bioscClass =
                new ManagementClass("Win32_BIOS");
            ManagementObjectCollection biosc =
                bioscClass.GetInstances();
            ManagementObjectCollection.ManagementObjectEnumerator
                bioscEnumerator =
                biosc.GetEnumerator();
            while (bioscEnumerator.MoveNext())
            {
                ManagementObject biosc1 =
                    (ManagementObject)bioscEnumerator.Current;
                Console.WriteLine(
                    "Attributes:
                    Attributes:\n\n" + "Version:\t" + biosc1["version"].ToString());
                Console.WriteLine(
                    "SerialNumber:\t" + biosc1["SerialNumber"].ToString());
                Console.WriteLine(
                    "OperatingSystem:\t" + biosc1["TargetOperatingSystem"].ToString());
                Console.WriteLine(
                    "Manufacturer:\t" + biosc1["Manufacturer"].ToString());
            }
            //return 0;
        }
    }
}
The code from last slide does not have any news:

- The **ManagementClass class** represents a Common Information Model (CIM) management class.
- **Win32_BIOS WMI class** represents the attributes of BIOS and members of this class enable you to access WMI data using a specific WMI class path.

- `GetInstances()` acquires a collection of all instances of the class.
- `GetEnumerator()` returns the enumerator (IEnumerator) for the collection.
- `IEnumerator.Current()` returns the same object.
- `IEnumerator.MoveNext()` advances the enumerator to the next element of the collection.

**Physical host:**

C:\> 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
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);
                }
            }
        }
    }
}

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
Double-click the result....
Object editor for Win32_TemperatureProbe.DeviceID="root\cimv2 0"

Qualifiers
- dynamic: CIM_BOOLEAN, TRUE
- Locale: CIM_SINT32, 1033 (0x409)
- provider: CIM_STRING, CIMWin32
- uuid: CIM_STRING, {4615eb2b-946f-11d2-93}

Add Qualifier, Edit Qualifier, Delete Qualifier

Properties
- Hide System Properties, Local Only
- Caption: CIM_STRING, Numeric Sensor
- ConfigManagerErrorCode: CIM_UINT32, <null>
- ConfigManagerUserConfig: CIM_BOOLEAN, <null>
- CreationClassName: CIM_STRING, Win32_TemperatureProbe
- CurrentReading: CIM_SINT32, <null>
- Description: CIM_STRING, CPU Internal Temperature
- DeviceID: CIM_STRING, root\cimv2 0

Add Property, Edit Property, Delete Property

Methods

Add Method, Edit Method, Delete Method

Update type
- Create only
- Update only
- Either
- Compatible
- Safe
- Force
using System;
using System.Management;

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

            foreach (ManagementObject aborges in tempinstance)
            {
                try
                {
                    if (!string.IsNullOrWhiteSpace(aborges.GetProperties("Status").ToString()))
                    {
                        string buffer = aborges.GetProperties("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("\nThis program IS RUNNING in a virtual machine!");
        }
    }
}
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.

Physical Host:
C:/> VM_Test2.exe
Status: OK  Thus, the program is running in a physical host!

Virtual Machine:
C:/> VM_Test2.exe
This program IS RUNNING in a virtual machine!
FEW CONCLUSIONS:

✓ Before trying to **unpack modern protectors**, it is really necessary to understand the **common anti-reversing techniques**.

✓ **MIASM, METASM** and **TRITON** are amazing tools to handle and deobfuscate complex codes.

✓ **Emulation** is an possible alternative to understand small and complicated piece of codes.

✓ **DTrace** has done an excellent job on Solaris and it may be an excellent tool on Windows operating system. Stay tuned. 😊

✓ Although excellent researches have found sophisticated **anti-vm** techniques, many other simples and smart ones exist. Take care.
Acknowledgments to:

✓ DEF CON’s staff, who have been always very kind with me.

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

✓ Remember: 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

- Malware and Security Researcher.
- Speaker at DEFCON USA 2018
- Speaker at HITB2019 Amsterdam
- 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