MODERN MALWARE THREAT: HANDLING OBFUSCATED CODE

CONFIDENCE CONFERENCE (2019)

by Alexandre Borges

- ✓ Malware and Security Researcher.
- ✓ Speaker at DEF CON USA 2018
- ✓ Speaker at DEF CON China 2019
- ✓ Speaker at HITB 2019 Amsterdam
- ✓ 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
- ❖ 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

x64dbg script 1/3

```
//#################################################################
// FileName    : dumpexe.txt (first draft)
// Comment     : Dump memory segments containing executables
// Author      : Alexandre Borges
// Date        : today
//#################################################################

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
```

x64dbg
script 2/3

```
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,"5468697320070726F6772616D”

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
```

x64dbg
script 3/3

```
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
```

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

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.

- ✓ We see obfuscated code every single day when we analyze commom userland malware, droppers written in VBA and Powershell, so it mightn't seem to be a big deal.

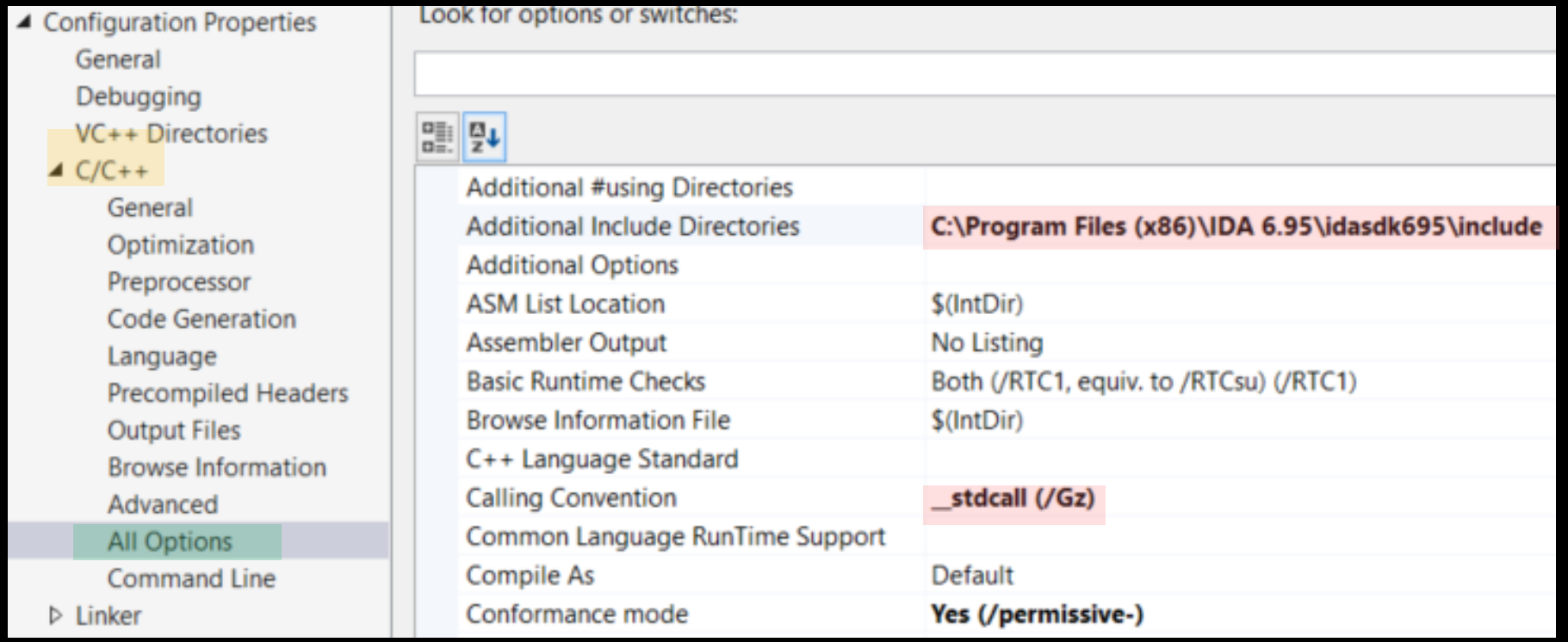
- ✓ We can use IDA Pro SDK to write plugins to extend the IDA Pro functionalities, analyze some code and data flow and even automatizing unpacking of strange malicious files.

- ✓ 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. ☺

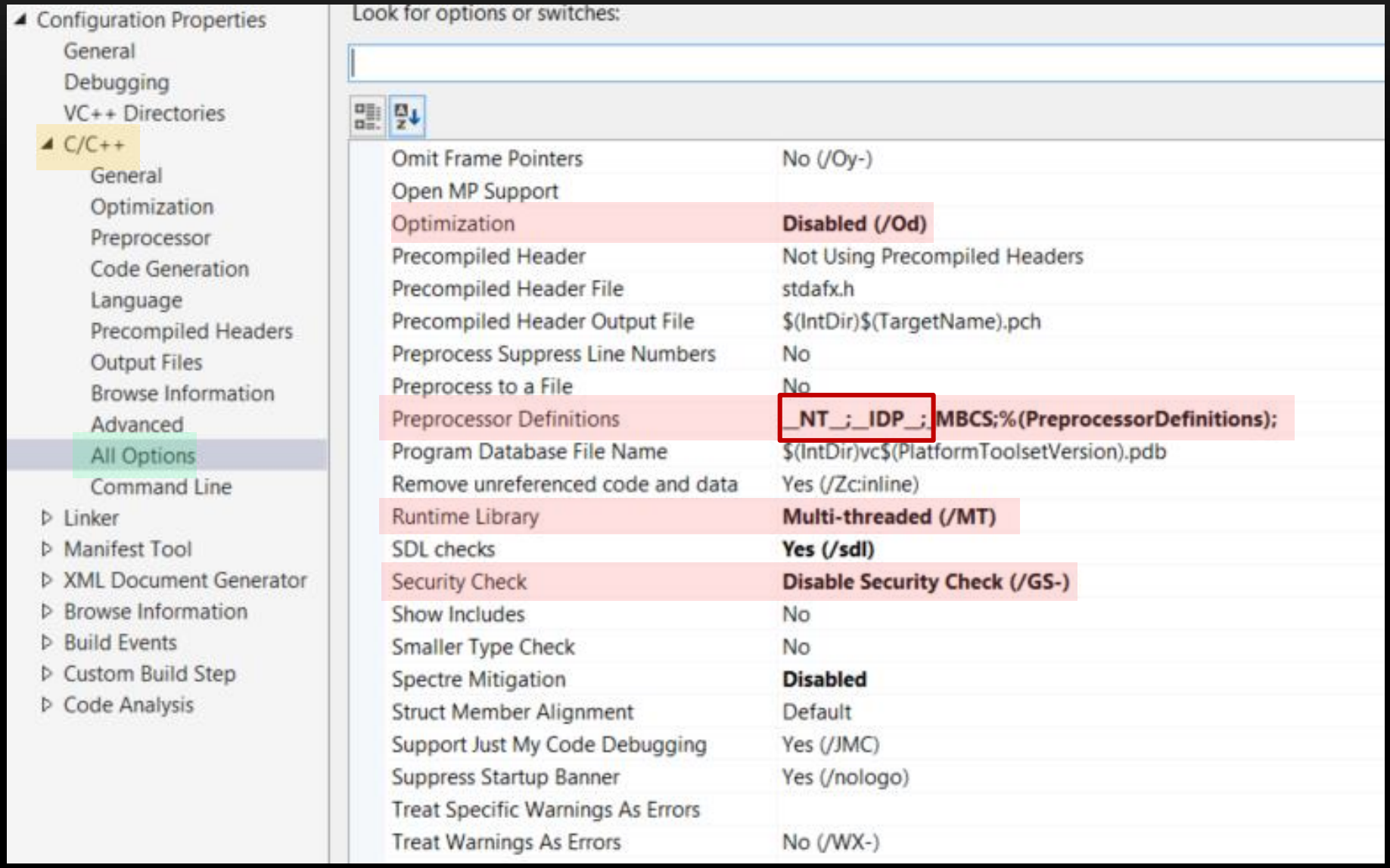
- ✓ It is quick to create a simple IDA Pro plugin. Download the IDA SDK from https://www.hex-rays.com/products/ida/support/download.shtml (likely, you will need a professional account). Copy it to a folder (idasdk695/) within the IDA Pro installation directory.

- ✓ 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.

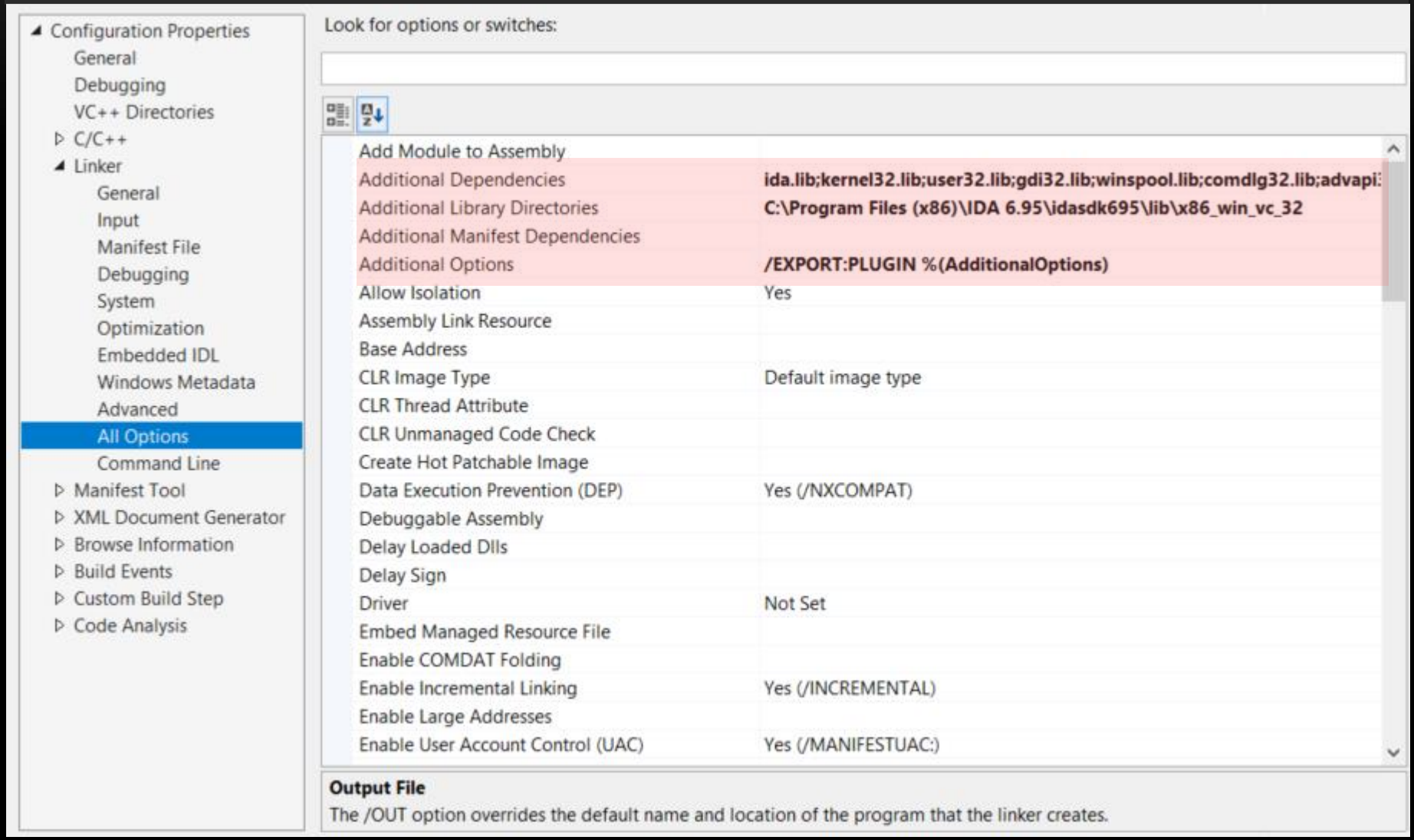


✓ Include the "__NT__;__IDP__" in Processor Definitions and change Runtime Library to "Multi-threaded" (MT) (take care: it is NOT /MTd).



- ✓ 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.



```
1   #include <ida.hpp>
2   #include <idp.hpp>
3   #include <loader.hpp>
4   #include <allins.hpp>
5   #include <strlist.hpp>
6   #include <search.hpp>
7
8
9   int IDAP_init()
10  {
11      return PLUGIN_KEEP;
12  }
13
14  void IDAP_term(void)
15  {
16
17  }
18
19  void IDAP_run(int arg)
20  {
21      msg("Hello CONFidence Conference! We love IDA Pro :)\n\n");
22
23      char confidence[MAXSTR];
24      string_info_t strinfo;
25      char s[] = "[a-z0-9]+[\.]{1,}[a-zA-Z0-9_-]+[\.]{1,}[a-z]{2,}";
26      auto last = BADADDR;
27      auto ea = 0;
28      auto urlcount = 1;
```

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. ☺

```
30      for (int x = 0; x < get_strlist_qty(); x++) {
31
32          get_strlist_item(x, &strinfo);
33          if (strinfo.length < sizeof(confidence)) {
34
35              get_many_bytes(strinfo.ea, confidence, strinfo.length);
36
37              {
38                  ea = 0;
39                  ea = find_text(strinfo.ea, 0, 0, s, SEARCH_REGEX);
40
41                  if (ea == strinfo.ea) {
42                      msg("Address 0x%x - URL %d: %s\n", strinfo.ea, urlcount, confidence);
43                      urlcount++;
44                  }
45              }
46          }
47      }
48
49      return;
50
51  }
52
53  char IDAP_comment[] = "The simplest possible plugin";
54  char IDAP_help[] = "CONFidence plugin";
55  char IDAP_name[] = "CONFidence plugin";
56  char IDAP_hotkey[] = "ALT-C";
57
58  plugin_t PLUGIN =
59  {
60      IDP_INTERFACE_VERSION,
61      0,
62      IDAP_init,
63      IDAP_term,
64      IDAP_run,
65      IDAP_comment,
66      IDAP_help,
67      IDAP_name,
68      IDAP_hotkey
69  };
```

It gets the number of strings from “Strings view”.

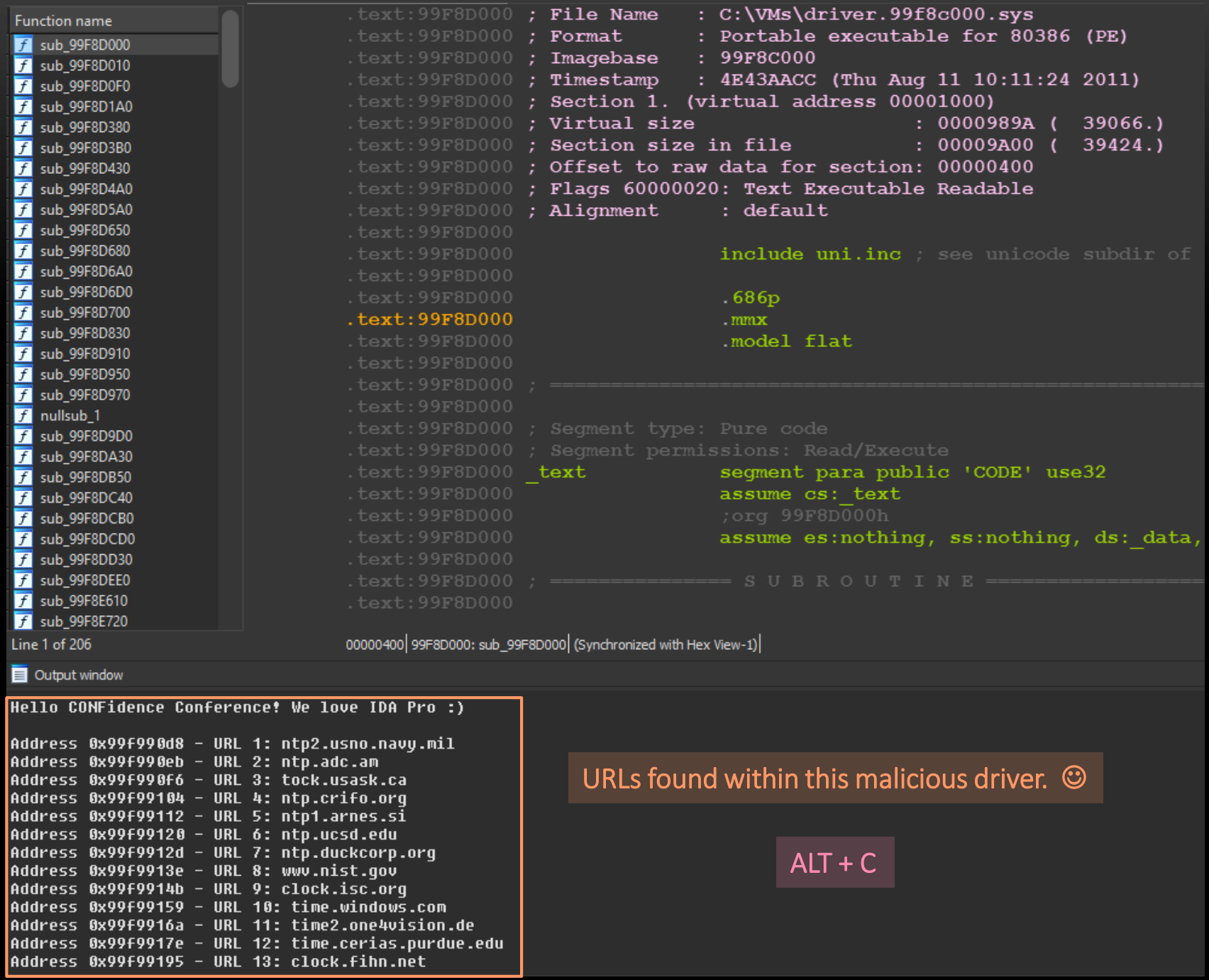
It gets strings.

The core logic is only it. It checks whether the string matches to the URL regex.

If checks, so ea == strinfo.ea. ☺

Plugin will be activated by combination ALT-C. ☺

Plugin structure.



URLs found within this malicious driver. ☺

ALT + C

- ✓ 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).

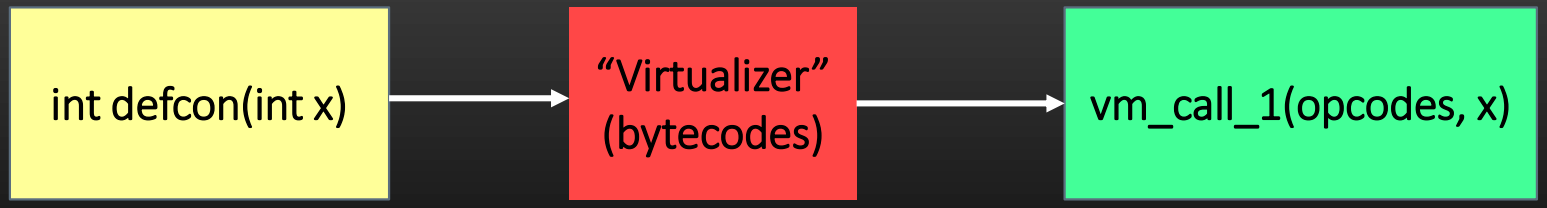
- ✓ Original IAT is removed from the original code (as usually applied by any packer). However, IAT from packers like Themida keeps only one function (TlsSetValue).

- ✓ 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.

- ✓ .NET protectors rename classes, methods, fields and external references.
- ✓ Instructions (x86/x64 code) are virtualized and transformed into virtual machine instructions (RISC instructions).
- ✓ Instructions are encrypted 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 found a not so common anti-vmware trick based on temperature (more about it later).

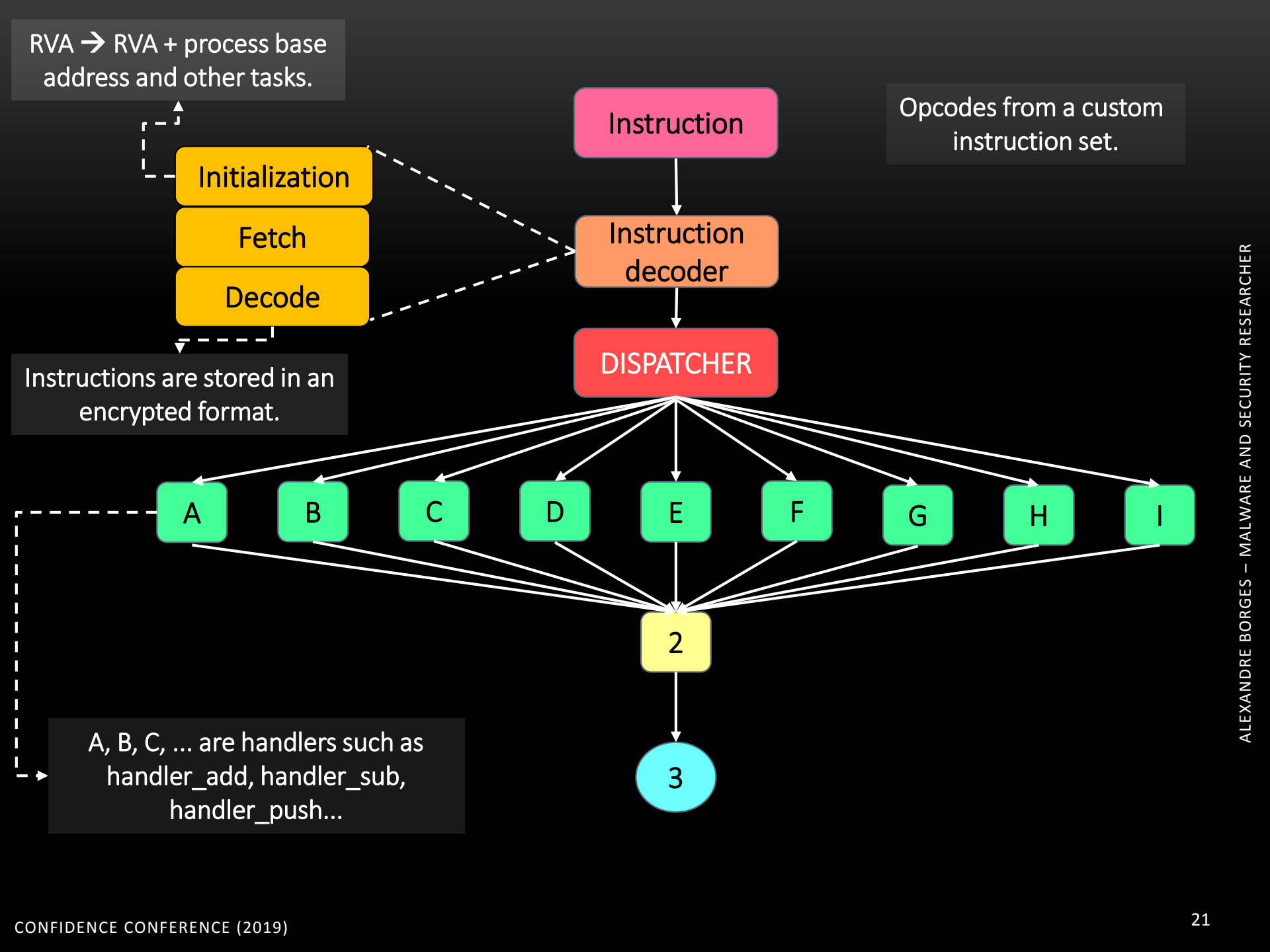


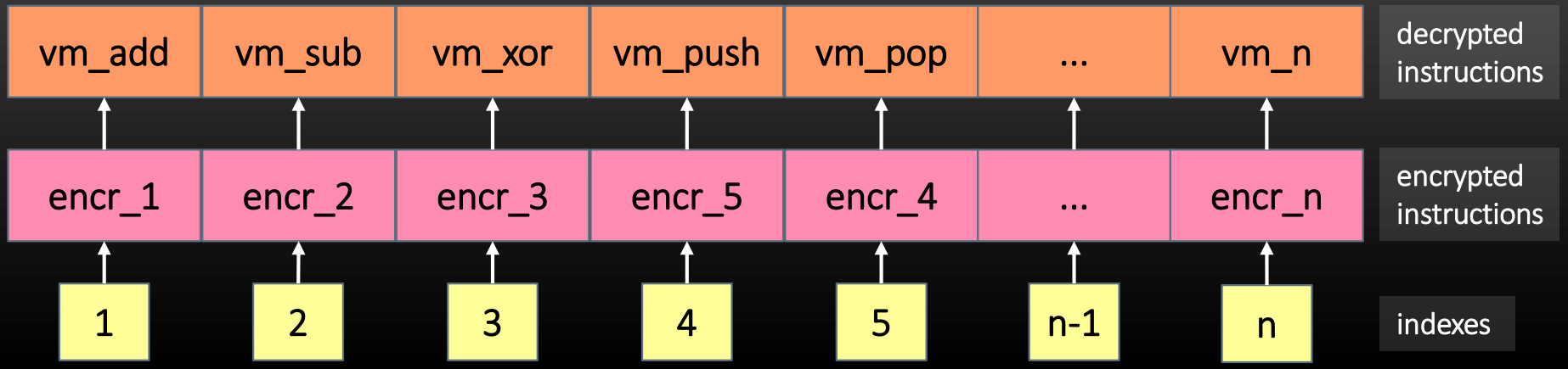
Fetches bytes, decodes them to instructions and dispatches them to handlers

- 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 oposite 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

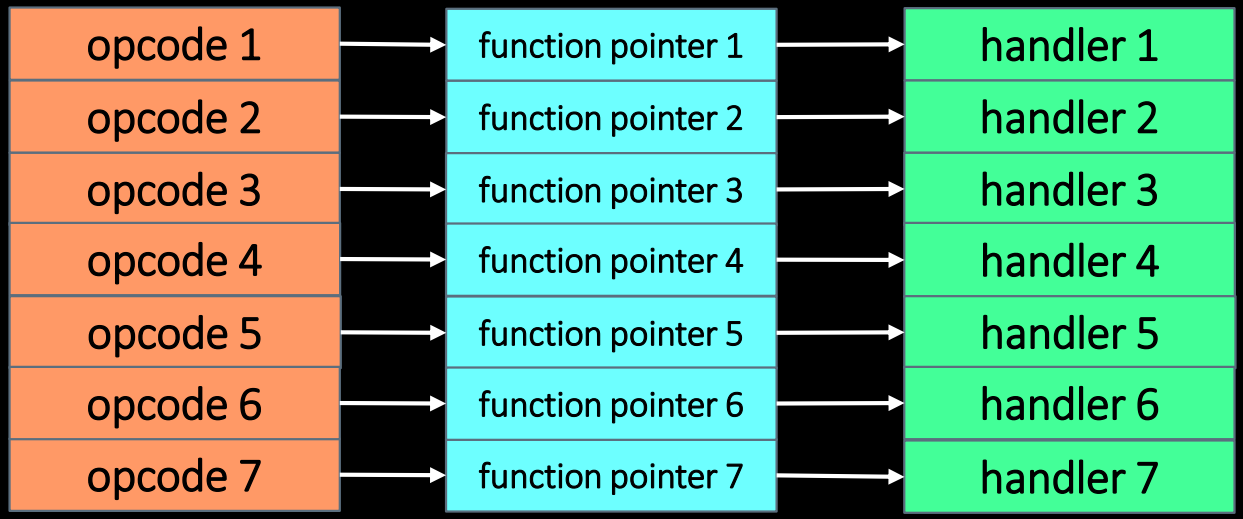
✓ Few interesting concepts:

- ✓ 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.


RVA → RVA + process base address and other tasks.
Initialization
Fetch
Decode
Instructions are stored in an encrypted format.
Instruction
Instruction decoder
DISPATCHER
Opcodes from a custom instruction set.
A
B
C
D
E
F
G
H
I
2
3
A, B, C, ... are handlers such as handler_add, handler_sub, handler_push...



recovering and decrypting funcions



function pointer table (likely encrypted)

- ✓ Is it easy to reverse virtualized and packed code? Certainly, it is not. The number of challenges might be huge ☺
- ✓ Remember: obfuscating is transforming a code from A to B by using any tricks (including virtualization).
- ✓ It is not so easy to identify whether the program is virtualized or not.
- ✓ Prologues and epilogues from each function could be not virtualized. Take care. ☺
- ✓ Have you tried to open an advanced packer in IDA Pro? First sight: only red and grey blocks (non-functions and data). ☹

- ✓ Sometimes VM handlers come from data blocks...

- ✓ Original code section could be "splitted" and "scattered" around the program (data and instructions are mixed in the binary, without having just one instruction block)

- ✓ Instructions which reference imported functions could have been either zeroed or replaced by NOP. ☹ Most certainly, they will be restored (re-inserted) dynamically by the packer later.

- ✓ If references are not zeroed, so they are usually translated to short jumps using RVA, for the same import address ("IAT obfuscation") ☺

- ✓ Worse, the API names could be hashed (as used in shellcodes). ☺
- ✓ Custom packers usually don't virtualize all x86 instructions.
- ✓ It is common to see a kind of mix between virtualized, native instructions and data after the packing procedure.
- ✓ Native APIs could be redirected to stub code, which forwards the call to (copied) native DLLs (from the respective APIs).
- ✓ The "hidden" function code could be copied (memcpy( )) to memory allocated by VirtualAlloc( ) ☺ Of course, there must be a fixup in the code to get these instructions.

- ✓ By the way, how many virtualized instructions exist?
- ✓ Are we able to classify virtualized instructions in groups according to operands and their purpose (memory access, conditional/unconditional jumps, arithmetic, general, an so on)?
- ✓ Pay attention to instruction's stem to put similar classes of instructions together (for example, jump instructions, direct calls, indirect calls and so on).
- ✓ Find similarity between virtualized instructions and x86 instructions.

- ✓ What are the “key instructions” that are responsible to make the transition from x86 mode to “virtualized mode” and vice-versa?

- ✓ It is interesting to find out the VM instruction’s size, which we might fit into a structure that represents encryption key, data, RVA (location), opcode (type) and so on.

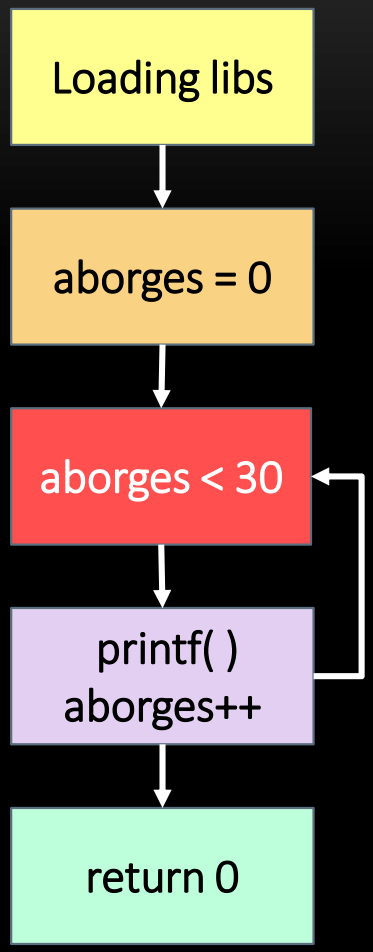
- ✓ It is recommended to try to find handlers to native x86 instructions (non-virtualized instruction)

- ✓ In this case, x86 instructions are also kept encrypted and compressed together with the virtualized instructions.

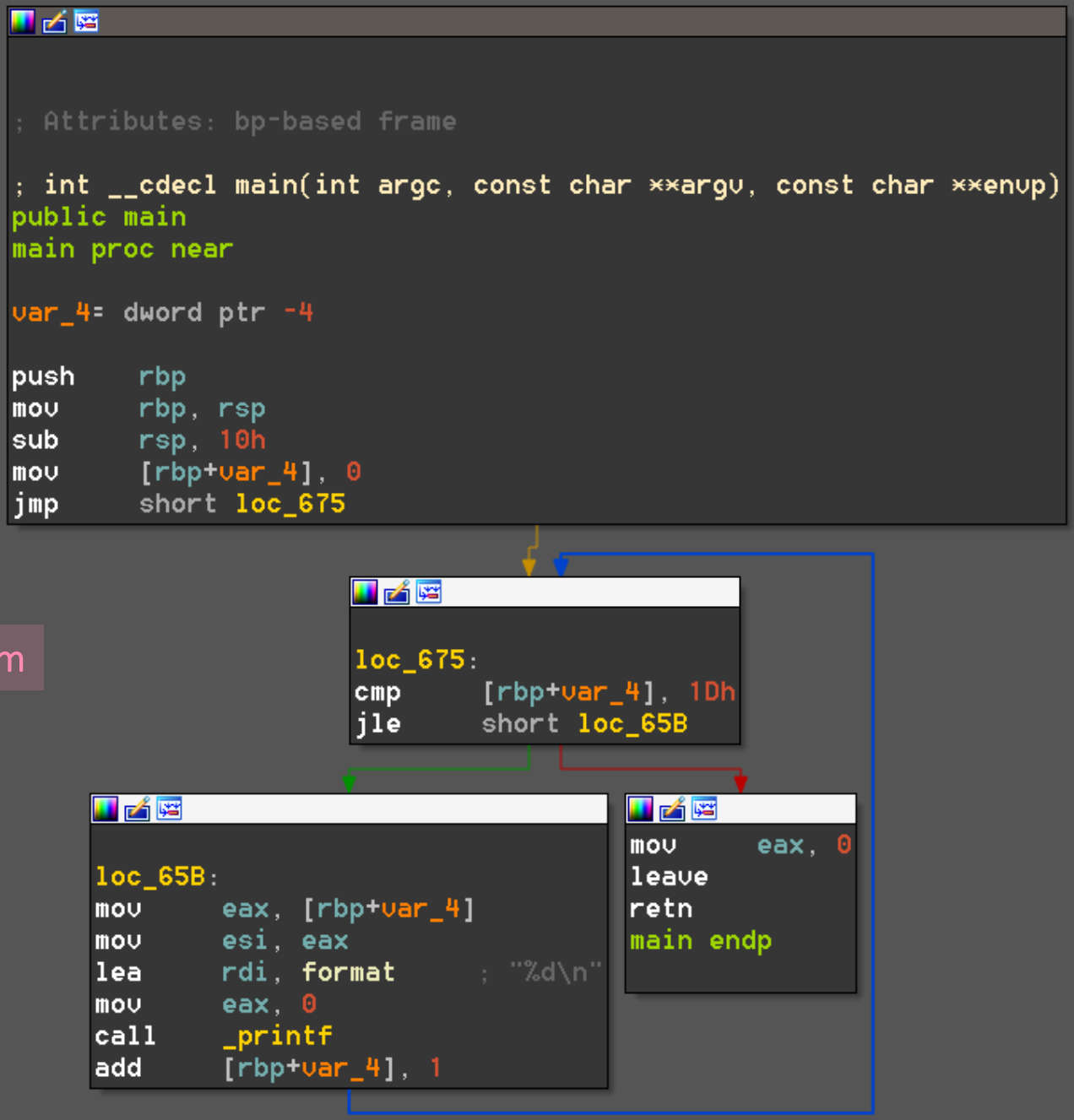
- ✓ Constant unfolding: technique used by obfuscators to replace a contant by a bunch of code that produces the same resulting constant's value.
- ✓ Pattern-based obfuscation: exchange of one instruction by a set of equivalent instructions.
- ✓ Abusing inline functions.
- ✓ Anti-VM techniques: prevents 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).

```c
#include <stdio.h>

int main (void)

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

        return 0;

}
```

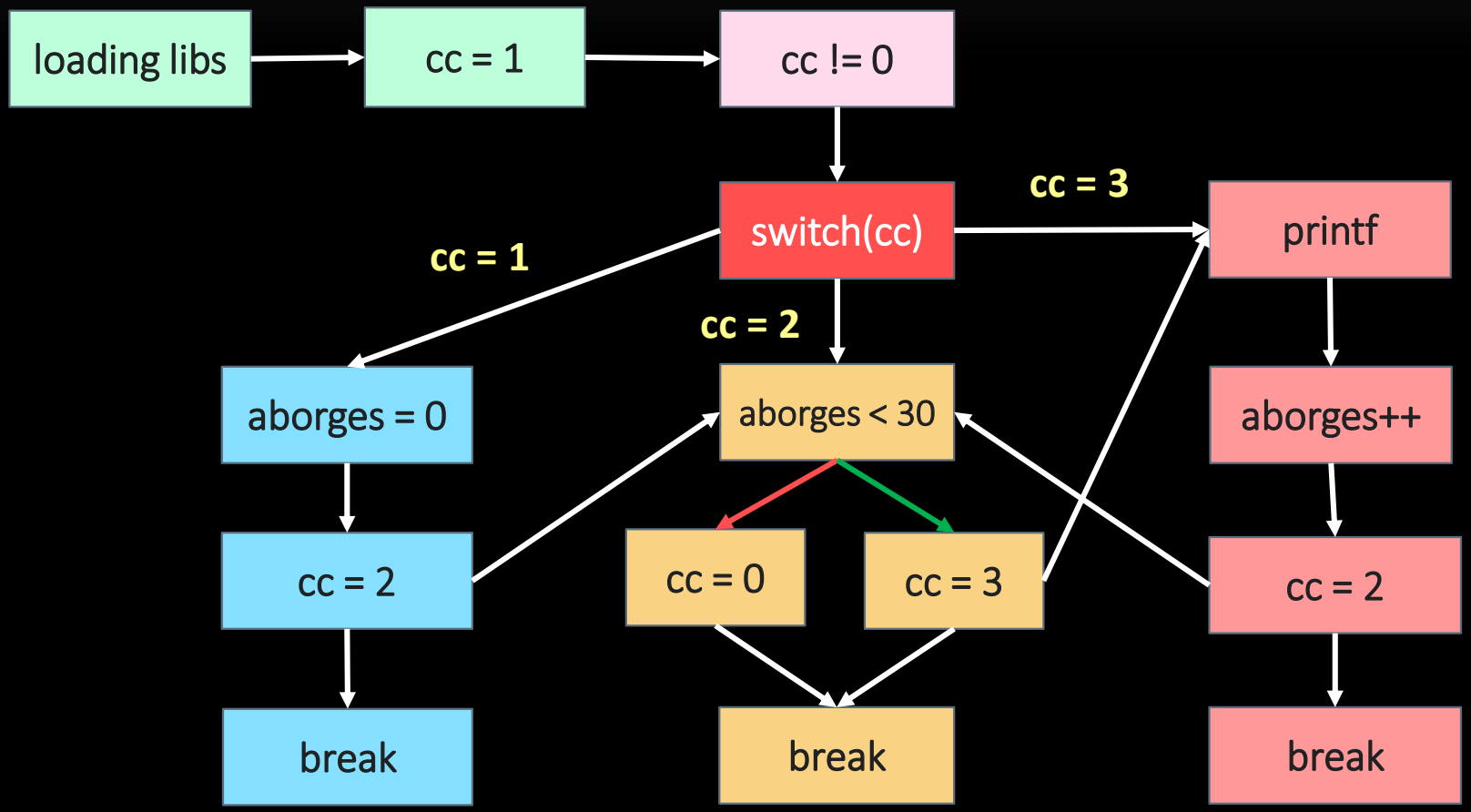


Original Program



❖ Disavantages:

- ✓ Loss of performance
- ✓ Easy to identify the CFG flattening



- ✓ The obfuscator-llvm is an excellent project to be used for code obsfuscation. To install it, it is recommended to add a swap file first (because the linkage stage):

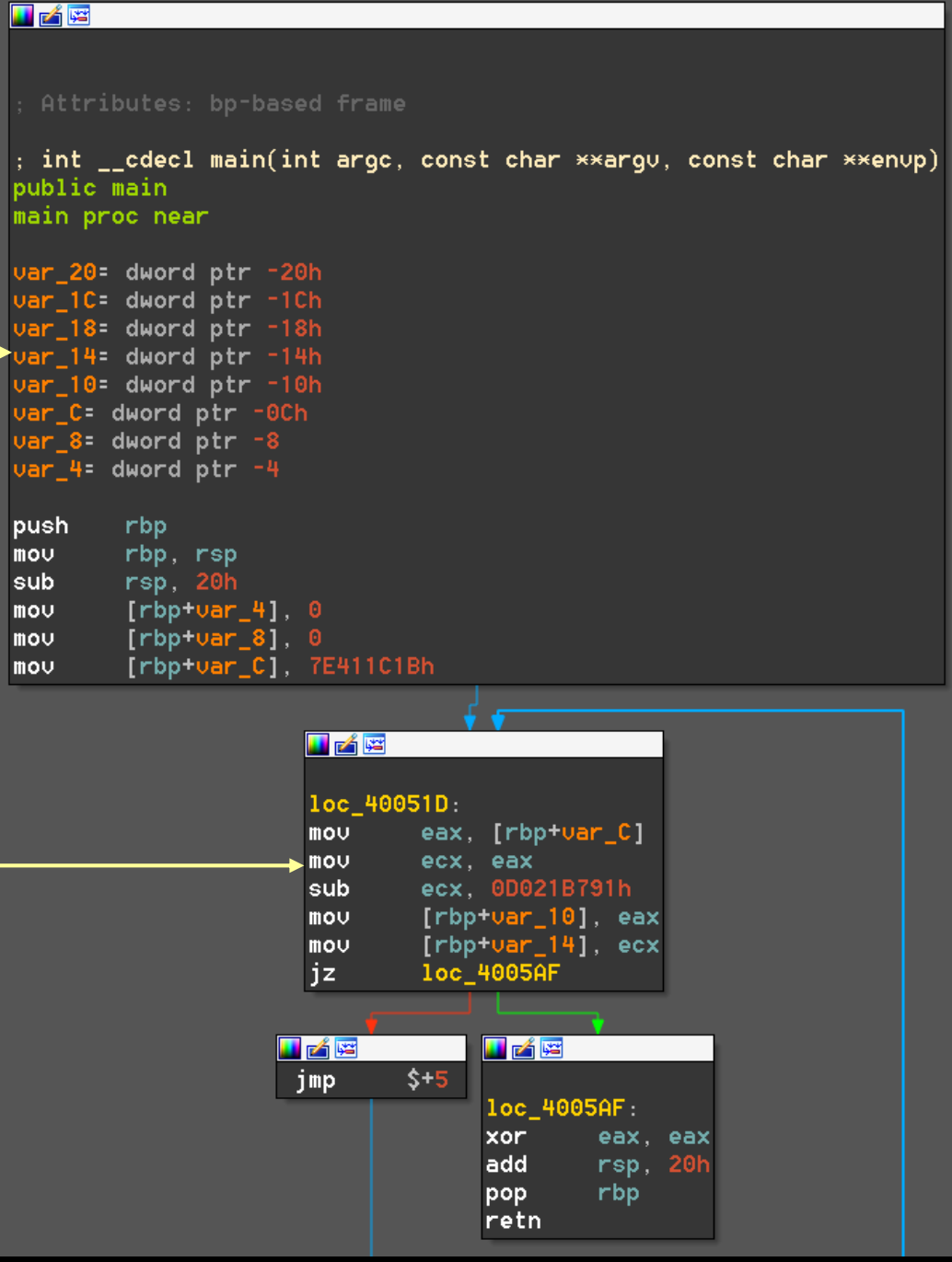
  - ✓ fallocate -l 8GB /swapfile
  - ✓ chmod 600 /swapfile
  - ✓ mkswap /swapfile
  - ✓ swapon /swapfile
  - ✓ swapon --show
  - ✓ apt-get install llvm-4.0
  - ✓ apt-get install gcc-multilib (install gcc lib support to 32 bit)
  - ✓ git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git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

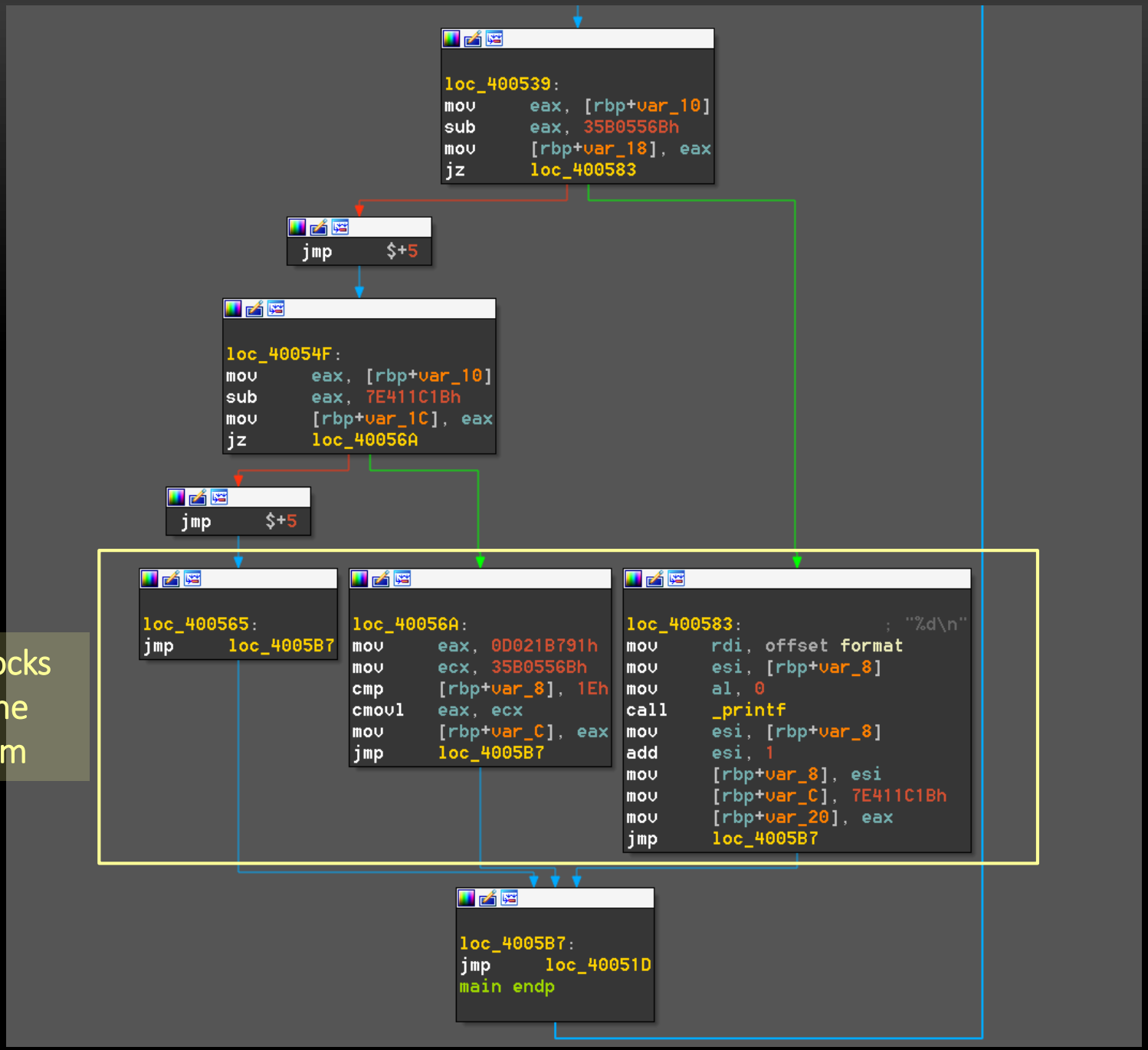
- ✓ Possible usages:

  - ✓ ./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

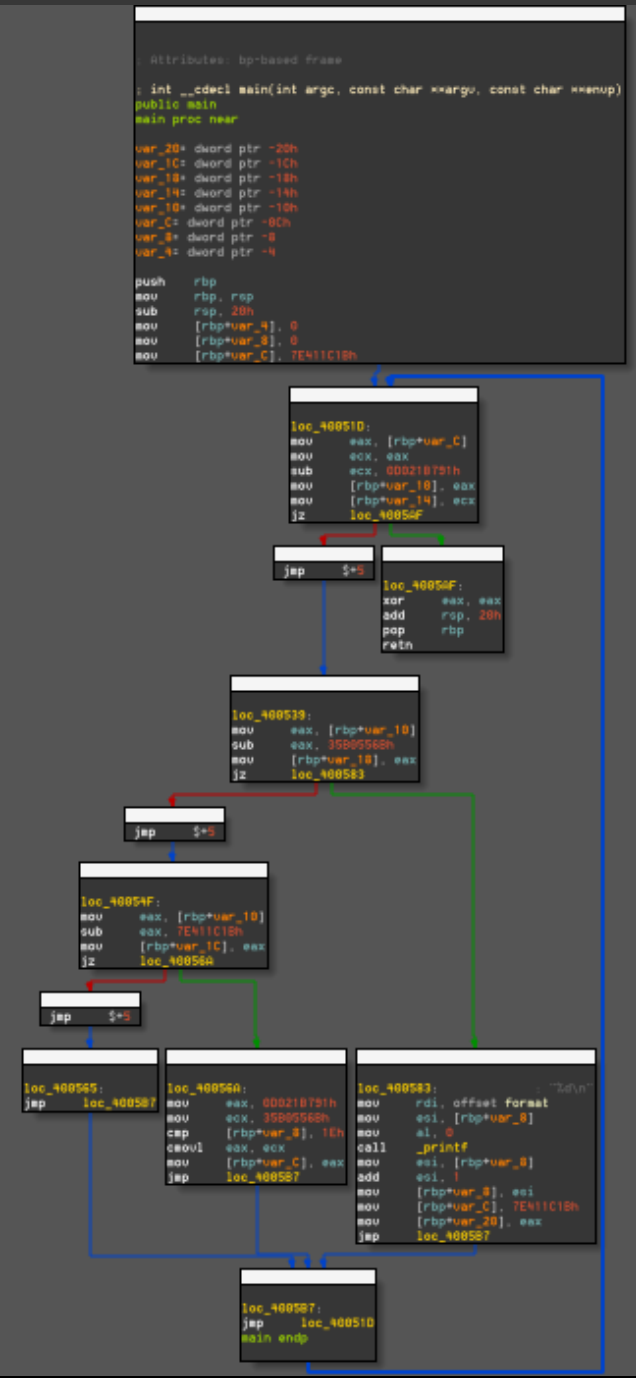
Main dispatcher





Main blocks from the program

General overview of the obfuscate code



```
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, 731796000415206604 8LL);
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

.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
```

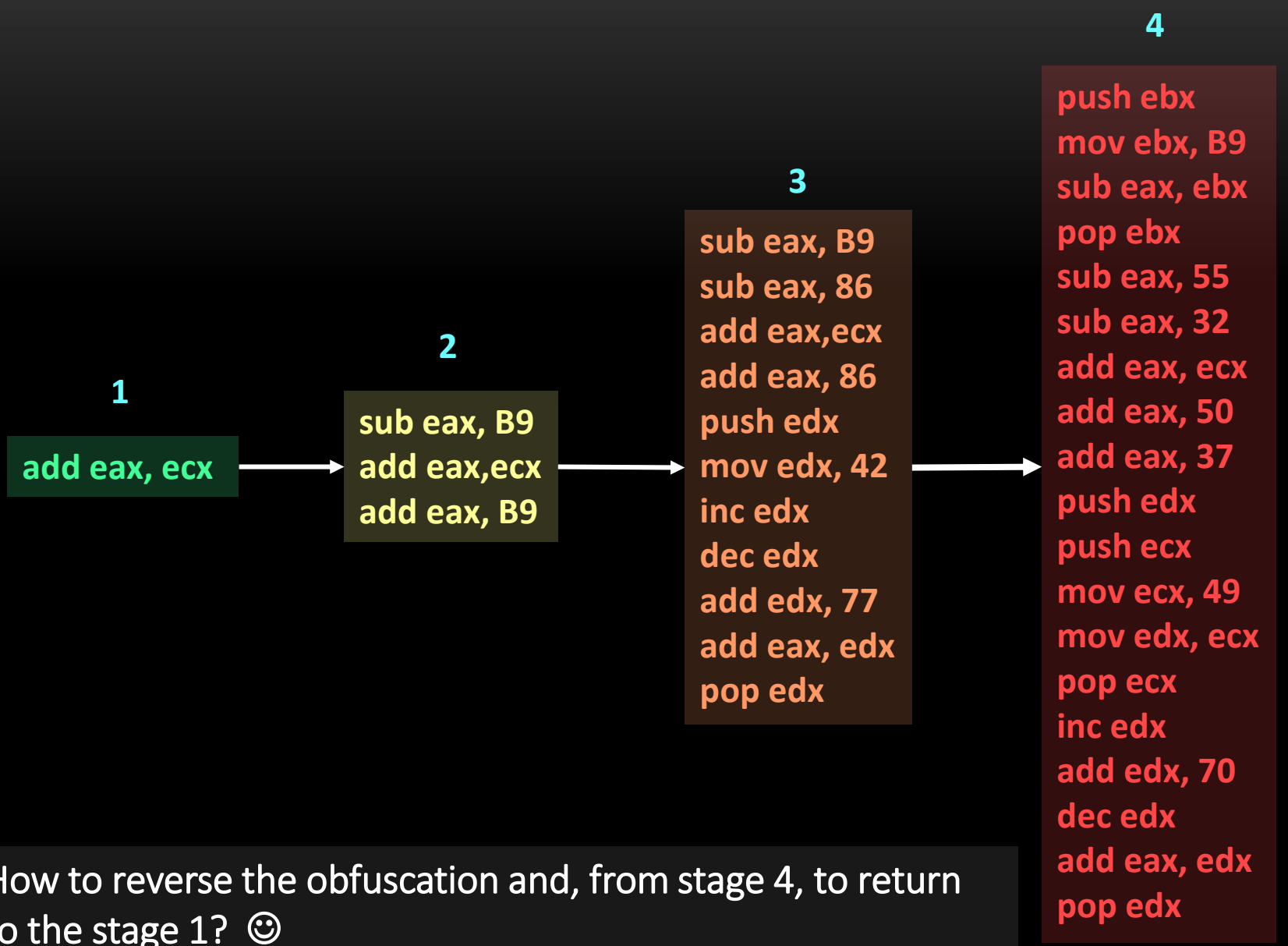
ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER


seg000:0000020C loc_20C: ; CODE XREF: sub_208+14↓j
seg000:0000020C lodsb
seg000:0000020D mov dl, al
seg000:0000020F sub dl, 41h ; 'A'
seg000:00000212 shl dl, 4
seg000:00000215 lodsb
seg000:00000216 sub al, 41h ; 'A'
seg000:00000218 add al, dl
seg000:0000021A stosb
seg000:0000021B dec ecx
seg000:0000021C jnz short loc_20C
seg000:0000021E retn
seg000:0000021E sub_208 endp
seg000:0000021F ; ----------
seg000:0000021F loc_21F: ; CODE XREF: seg000:00000206↑j
seg000:0000021F call sub_208
seg000:00000224 mov ebp, esp
seg000:00000226 sub esp, 40h
seg000:0000022C jmp loc_364
seg000:00000231 ; =============== S U B R O U T I N E ===============
seg000:00000231 sub_231 proc near ; CODE XREF: sub_252+1F↓p
seg000:00000231 arg_0 = dword ptr 4
seg000:00000231 push esi
seg000:00000232 push edi
seg000:00000233 mov esi, [esp+8+arg_0]
seg000:00000237 xor edi, edi
seg000:00000239 cld
seg000:0000023A loc_23A: ; CODE XREF: sub_231+15↓j
seg000:0000023A xor eax, eax
seg000:0000023C lodsb
seg000:0000023D cmp al, ah
seg000:0000023F jz short loc_24B
seg000:00000241 ror edi, 0Dh
seg000:00000244 add edi, eax
seg000:00000246 jmp loc_23A
seg000:0000024B ; ----------
seg000:0000024B loc_24B: ; CODE XREF: sub_231+E↑j
seg000:0000024B mov eax, edi
seg000:0000024D pop edi
seg000:0000024E pop esi
Decryption instructions ☺
Decrypted shellcode

```
00401040    call + $5
00401045    pop ecx
00401046    inc ecx
00401047    inc ecx
00401048    add ecx, 4
00401049    add ecx, 4
0040104A    push ecx
0040104B    ret
0040104C    sub ecx, 6
0040104D    dec ecx
0040104E    dec ecx
0040104F    jmp 0x401320
```

- 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

```
#!/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
```

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

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

```
addrstart = 0
asmcode = mycode.init_disassembler
asmcode.disassemble(addrstart)
confidence_di = asmcode.di_at(addrstart)
confidence = confidence_di.block
puts "\n<!!!> CONFidence Conference 2019:\n "
puts confidence.list


confidence.list.each{|aborges|
        puts "\n<!!!> #{aborges.instruction}"
        back = aborges.backtrace_binding()
        v = back.values
        k = back.keys
        j = k.zip(v)
        puts "CONFidence Conference data flow follows below:\n"
        j.each do |mykeys, myvalues|

            puts " Processing: #{mykeys} ==> #{myvalues}"

            if aborges.opcode.props[:setip]
              puts "\nCONFidence Conference control flow follows below:\n"
              puts " >>> #{asmcode.get_xrefs_x(aborges)}"
            end
        end
}

addrstart2 = 0
asmcode2 = mycode.init_disassembler
asmcode2.disassemble(addrstart2)
```

initialize and disassemble code since beginning (start).

list the assembly code.

initialize the backtracking engine.

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

```
dd = asmcode2.block_at(addrstart2)
final = asmcode2.get_xrefs_x(dd.list.last).first
puts "\n[+] final output: #{final}"

values = asmcode2.backtrace(final, dd.list.last.address, {:log => backtracing_log
= [] , :include_start => true})
backtracing_log.each{|record|
        case type = record.first
        when :start
                record, expression, addresses = record
                puts "[start] Here is the sequence of expression evaluations  #{ex
pression} from 0x#{addresses.to_s(16)}\n"

        when :di
                record, new, old, instruction = record
                puts "[new update] instruction #{instruction},\n --> updating expr
ession once again from #{old} to #{new}\n"

        end
}

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

Backtracking from the last instruction.

logs the sequence of backtracked instructions.

Show only the effective instructions, which really can alter the final result.

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

<!!!> CONFidence Conference 2019:

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

<!!!> push ebx
CONFidence Conference data flow follows below:
 Processing: esp ==> esp-4
 Processing: dword ptr [esp] ==> ebx

<!!!> mov ebx, 0b9h
CONFidence Conference data flow follows below:
 Processing: ebx ==> 0b9h
```

Remember: this is our obfuscated code. ☺

```
<!!!> sub eax, ebx
CONFidence Conference data flow follows below:
 Processing: eax ==> eax-ebx
 Processing: eflag_z ==> (((eax&0ffffffffh)-(ebx&0ffffffffh))&0ffffffffh)==0
 Processing: eflag_s ==> ((((eax&0ffffffffh)-(ebx&0ffffffffh))&0ffffffffh)>>1fh)!=0
 Processing: eflag_c ==> (eax&0ffffffffh)<(ebx&0ffffffffh)
 Processing: eflag_o ==> ((((eax&0ffffffffh)>>1fh)!=0)==(!(((ebx&0ffffffffh)>>1fh)!=0)
))&&((((eax&0ffffffffh)>>1fh)!=0)!=(((((eax&0ffffffffh)-(ebx&0ffffffffh))&0ffffffffh)>
>1fh)!=0))

<!!!> pop ebx
CONFidence Conference data flow follows below:
 Processing: esp ==> esp+4
 Processing: ebx ==> dword ptr [esp]

<!!!> sub eax, 55h
CONFidence Conference data flow follows below:
 Processing: eax ==> eax-55h
 Processing: eflag_z ==> (((eax&0ffffffffh)-((55h)&0ffffffffh))&0ffffffffh)==0
 Processing: eflag_s ==> ((((eax&0ffffffffh)-((55h)&0ffffffffh))&0ffffffffh)>>1fh)!=0
 Processing: eflag_c ==> (eax&0ffffffffh)<((55h)&0ffffffffh)
 Processing: eflag_o ==> ((((eax&0ffffffffh)>>1fh)!=0)==(!((((55h)&0ffffffffh)>>1fh)!=
0)))&&((((eax&0ffffffffh)>>1fh)!=0)!=(((((eax&0ffffffffh)-((55h)&0ffffffffh))&0fffffff
fh)>>1fh)!=0))

<!!!> sub eax, 32h
CONFidence Conference data flow follows below:
 Processing: eax ==> eax-32h
 Processing: eflag_z ==> (((eax&0ffffffffh)-((32h)&0ffffffffh))&0ffffffffh)==0
 Processing: eflag_s ==> ((((eax&0ffffffffh)-((32h)&0ffffffffh))&0ffffffffh)>>1fh)!=0
 Processing: eflag_c ==> (eax&0ffffffffh)<((32h)&0ffffffffh)
 Processing: eflag_o ==> ((((eax&0ffffffffh)>>1fh)!=0)==(!((((32h)&0ffffffffh)>>1fh)!=
0)))&&((((eax&0ffffffffh)>>1fh)!=0)!=(((((eax&0ffffffffh)-((32h)&0ffffffffh))&0fffffff
fh)>>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
6 sub eax, ebx
9 sub eax, 55h
0ch sub eax, 32h
0fh add eax, ecx
11h add eax, 50h
14h add eax, 37h
19h mov ecx, 49h
1eh mov edx, ecx
21h inc edx
22h add edx, 70h
25h dec edx
26h 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-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

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

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

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

    }

    ks_free(encode);
    ks_close(keyeng);

    return 0;
}
```

instructions from the original obsfuscated code

Creating a keystone engine

Assembling our instructions using keystone engine.

Freeing memory and closing engine.

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

KEYSTONE_LDFLAGS = -lkeystone -lstdc++ -lm

all:
	${CC} -o confidence2019 confidence2019.c ${KEYSTONE_LDFLAGS}

clean:
	rm -rf *.o confidence2019
root@kali:~/programs/confidence#
root@kali:~/programs/confidence# make
cc -o confidence2019 confidence2019.c -lkeystone -lstdc++ -lm
root@kali:~/programs/confidence#
root@kali:~/programs/confidence# ./confidence2019
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/confidence#
root@kali:~/programs/confidence#
root@kali:~/programs/confidence# ./confidence2019 | xxd -r -p - > confidence20
19.bin
root@kali:~/programs/confidence#
root@kali:~/programs/confidence# hexdump -C confidence2019.bin
00000000  53 bb b9 00 00 00 29 d8  5b 83 e8 55 83 e8 32 01  |S.....).[..U..2.|
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/confidence#
```

```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\xc
8\x83\xc0\x50\x83\xc0\x37\x52\x51\xb9\x49\x00\x00\x00\x89\xca\x59\x42\x83\xc2\x70
\x4a\x01\xd0\x5a"

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

        if (cs_open(CS_ARCH_X86, CS_MODE_32, &cs_handle) != CS_ERR_OK)
                return -1;
        count = cs_disasm(cs_handle, CODE, sizeof(CODE)-1, 0x0001, 0, &instructio
n);
        if (count > 0) {
                size_t j;
                for (j = 0; j < count; j++) {
                        printf("0x%"PRIx32":\t%s\t\t%s\n", 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;
}
```

To install Capstone: apt-get install libcapstone3 libcapstone-dev ☺

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

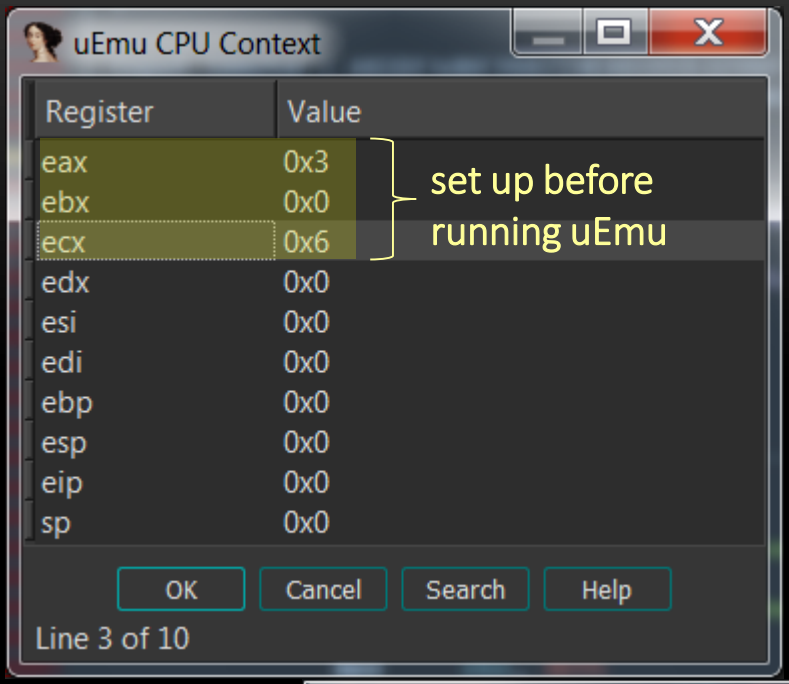
```
root@kali:~/programs/confidence/capstone# more Makefile
.PHONY: all clean

CAPSTONE_LDFLAGS = -lcapstone -lstdc++ -lm

all:
	${CC} -o confidence2019_rev confidence2019_rev.c ${CAPSTONE_LDFLAGS}

clean:
	rm -rf *.o confidence2019_rev
root@kali:~/programs/confidence/capstone#
root@kali:~/programs/confidence/capstone# make
cc -o confidence2019_rev confidence2019_rev.c -lcapstone -lstdc++ -lm
root@kali:~/programs/confidence/capstone#
root@kali:~/programs/confidence/capstone# ./confidence2019_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
root@kali:~/programs/confidence/capstone#
```

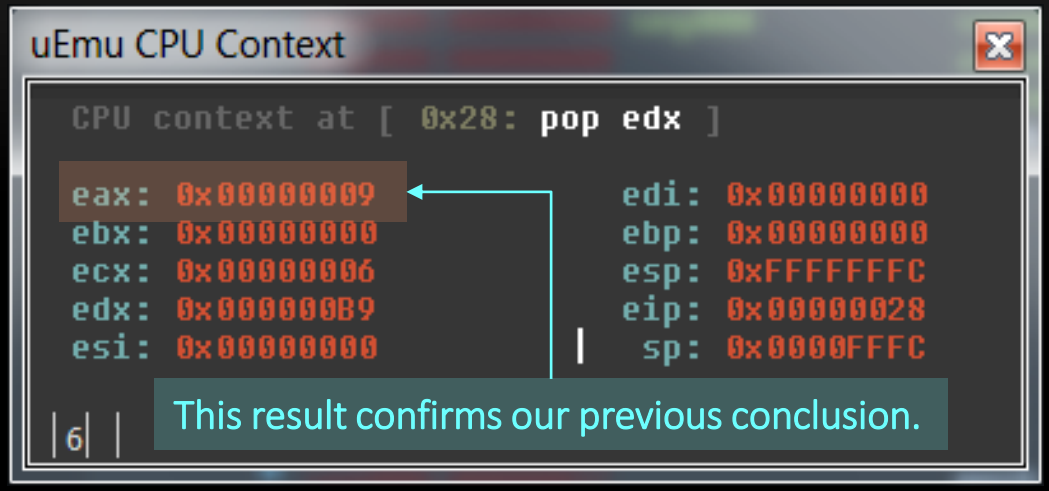
Original code disassembled by Capstone. ☺

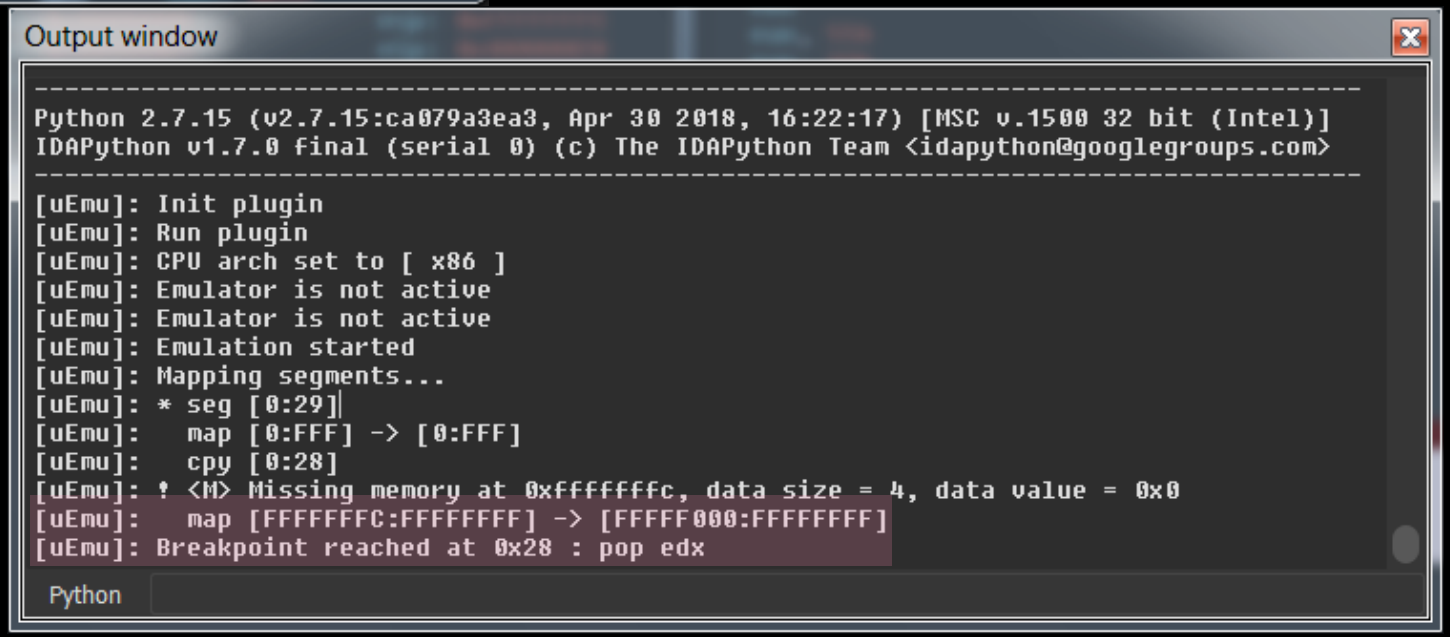
```
seg000:00000000 ; File Name   : C:\VMs\confidence2019.bin
seg000:00000000 ; Format      : Binary file
seg000:00000000 ; Base Address: 0000h Range: 0000h - 0029h Loaded length: 0029h
seg000:00000000
seg000:00000000                 .686p
seg000:00000000                 .mmx
seg000:00000000                 .model flat
seg000:00000000
seg000:00000000 ; ===========================================================
seg000:00000000
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:n
seg000:00000000                 push    ebx
seg000:00000001                 mov     ebx, 0B9h
seg000:00000006                 sub     eax, ebx
seg000:00000008                 pop     ebx
seg000:00000009                 sub     eax, 55h
seg000:0000000C                 sub     eax, 32h
seg000:0000000F                 add     eax, ecx
seg000:00000011                 add     eax, 50h
seg000:00000014                 add     eax, 37h
seg000:00000017                 push    edx
seg000:00000018                 push    ecx
seg000:00000019                 mov     ecx, 49h
seg000:0000001E                 mov     edx, ecx
seg000:00000020                 pop     ecx
seg000:00000021                 inc     edx
seg000:00000022                 add     edx, 70h
seg000:00000025                 dec     edx
seg000:00000026                 add     eax, edx
seg000:00000028                 pop     edx
seg000:00000028 seg000          ends
```

IDA Pro confirms our disassembly task. ☺

- ✓ 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

```
 1 #include <unicorn/unicorn.h>
 2 #include <string.h>
 3
 4 // Our code to be emulated.
 5
 6 #define CONFidence_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\x8
   9\xca\x59\x42\x83\xc2\x70\x4a\x01\xd0\x5a"
 7
 8 // Emulation start address and a simple macro.
 9
10 #define ADDR 0x1000000
11 #define MIN(x, y) (x < y? x : y)
12
13 // Hook the instruction execution.
14
15 static void hook_code(uc_engine *uc, uint64_t address, uint32_t size, void
    *user_data)
16 {
17         int r_eip;
18         int r_eax;
19         int r_ebx;
20         int r_ecx;
21         int r_edx;
22
23         uint8_t instr_size[16];
24
```

```
25          printf("\nTracing instruction at 0x%x , instruction size = 0x%x\n", addr
    ess, size);
26
27          uc_reg_read(uc, UC_X86_REG_EIP, &r_eip);
28          uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
29          uc_reg_read(uc, UC_X86_REG_EBX, &r_ebx);
30          uc_reg_read(uc, UC_X86_REG_ECX, &r_ecx);
31          uc_reg_read(uc, UC_X86_REG_EDX, &r_edx);
32
33  // Print the initial values of registries.
34
35          printf("\n>> EIP=0x%x ", r_eip);
36          printf(" | EAX=0x%x ", r_eax);
37          printf(" | EBX=0x%x ", r_ebx);
38          printf(" | ECX=0x%x ", r_ecx);
39          printf(" | EDX=0x%x ", r_edx);
40          printf("\n>> Executed hex code: ");
41
42          size = MIN(sizeof(instr_size), size);
43          if (!uc_mem_read(uc, address, instr_size, size)) {
44                  uint32_t i;
45                  for (i=0; i<size; i++) {
46                          printf("%x ", instr_size[i]);
47                  }
48                  printf("\n");
49          }
50  }
51
52  int main(int argc, char **argv, char **envp)
53  {
54
```

```
55 // Declare and initialize few variables
56
57         uc_engine *uc;
58         uc_hook traceinstr;
59         uc_err err;
60
61 // Set up the initial registry values.
62 // We have to set up the ESP register for emulating PUSH/POP instructions.
63
64         int r_eax = 0x4;
65         int r_ebx = 0x0;
66         int r_ecx = 0x7;
67         int r_edx = 0x0;
68         int r_esp = ADDR + 200000;
69
70         printf("\nInitial register values: \n");
71
72         printf("\n>> EAX = %x ", r_eax);
73         printf("\n>> EBX = %x ", r_ebx);
74         printf("\n>> ECX = %x ", r_ecx);
75         printf("\n>> EDX = %x ", r_edx);
76
77         printf("\n\nOur emulated code is: \n");
78
79
80 // We are emulating a 32-bit application in x86 emulator, so initialize the emul
   ator in X86-32bit mode :)
81 // If we wished to emulate in a x64 emulator, so we would use UC_MODE_64.
82
83         err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
84         if (err != UC_ERR_OK) {
85                 printf("A fail to use uc_open() has occured and the error return
   ed is: %u\n", err);
86                 return -1;
87         }
```

```
88
89 // We are reserving 4MB memory for this emulation. Additionally, UC_PROT_ALL mea
   ns: RWX.
90
91         uc_mem_map(uc, ADDR, 4 * 1024 * 1024, UC_PROT_ALL);
92
93 // write machine code to be emulated to memory
94
95         if (uc_mem_write(uc, ADDR, CONFidence_CODE, sizeof(CONFidence_CODE) - 1)
   ) {
96                 printf("It has happened a fail during the write emulation code t
   o memory!\n");
97                 return -1;
98 }
99
100 // We need to initialize the machine registers
101
102         uc_reg_write(uc, UC_X86_REG_EAX, &r_eax);
103         uc_reg_write(uc, UC_X86_REG_EBX, &r_ebx);
104         uc_reg_write(uc, UC_X86_REG_ECX, &r_ecx);
105         uc_reg_write(uc, UC_X86_REG_EDX, &r_edx);
106         uc_reg_write(uc, UC_X86_REG_ESP, &r_esp);
107
108 // uc: hook handle ; traceinstr: reference to uc_hook ; UC_HOOK_CODE: hook type
   ; hook_code: callback function
109
110         uc_hook_add(uc, &traceinstr, UC_HOOK_CODE, hook_code, NULL, 1, 0);
111
112
```

```
112
113 // Start the emulation engine and emulate code in infinite time (first zero
     below) & unlimited instructions (second zero below).
114
115         err=uc_emu_start(uc, ADDR, ADDR + sizeof(CONFidence_CODE) - 1, 0, 0
    );
116         if (err) {
117
118                 printf("The uc_emu_start() function has failed with error r
    eturning %u: %s\n", err, uc_strerror(err));
119
120 }
121
122 // Finally, print out the final registers values.
123
124         printf("\nThe final CPU registers contain the following content: \n
    \n");
125
126         uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
127         uc_reg_read(uc, UC_X86_REG_EBX, &r_ebx);
128         uc_reg_read(uc, UC_X86_REG_ECX, &r_ecx);
129         uc_reg_read(uc, UC_X86_REG_EDX, &r_edx);
130         printf(">>> EAX = 0x%x", r_eax);
131         printf("\n>>> EBX = 0x%x", r_ebx);
132         printf("\n>>> ECX = 0x%x", r_ecx);
133         printf("\n>>> EDX = 0x%x\n\n", r_edx);
134
135         uc_close(uc);
136
137         return 0;
138 }
```

```
root@kali:~/programs/confidence/unicorn# ./unicorn_confidence

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=0xffffff4b  | EBX=0xb9  | ECX=0x7  | EDX=0x0
>> Executed hex code: 5b
```

```
Tracing instruction at 0x1000021 , instruction size = 0x1

>> EIP=0x1000021  | EAX=0xffffff52  | EBX=0x0  | ECX=0x7  | EDX=0x49
>> Executed hex code: 42

Tracing instruction at 0x1000022 , instruction size = 0x3

>> EIP=0x1000022  | EAX=0xffffff52  | EBX=0x0  | ECX=0x7  | EDX=0x4a
>> Executed hex code: 83 c2 70

Tracing instruction at 0x1000025 , instruction size = 0x1

>> EIP=0x1000025  | EAX=0xffffff52  | EBX=0x0  | ECX=0x7  | EDX=0xba
>> Executed hex code: 4a

Tracing instruction at 0x1000026 , instruction size = 0x2

>> EIP=0x1000026  | EAX=0xffffff52  | 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.

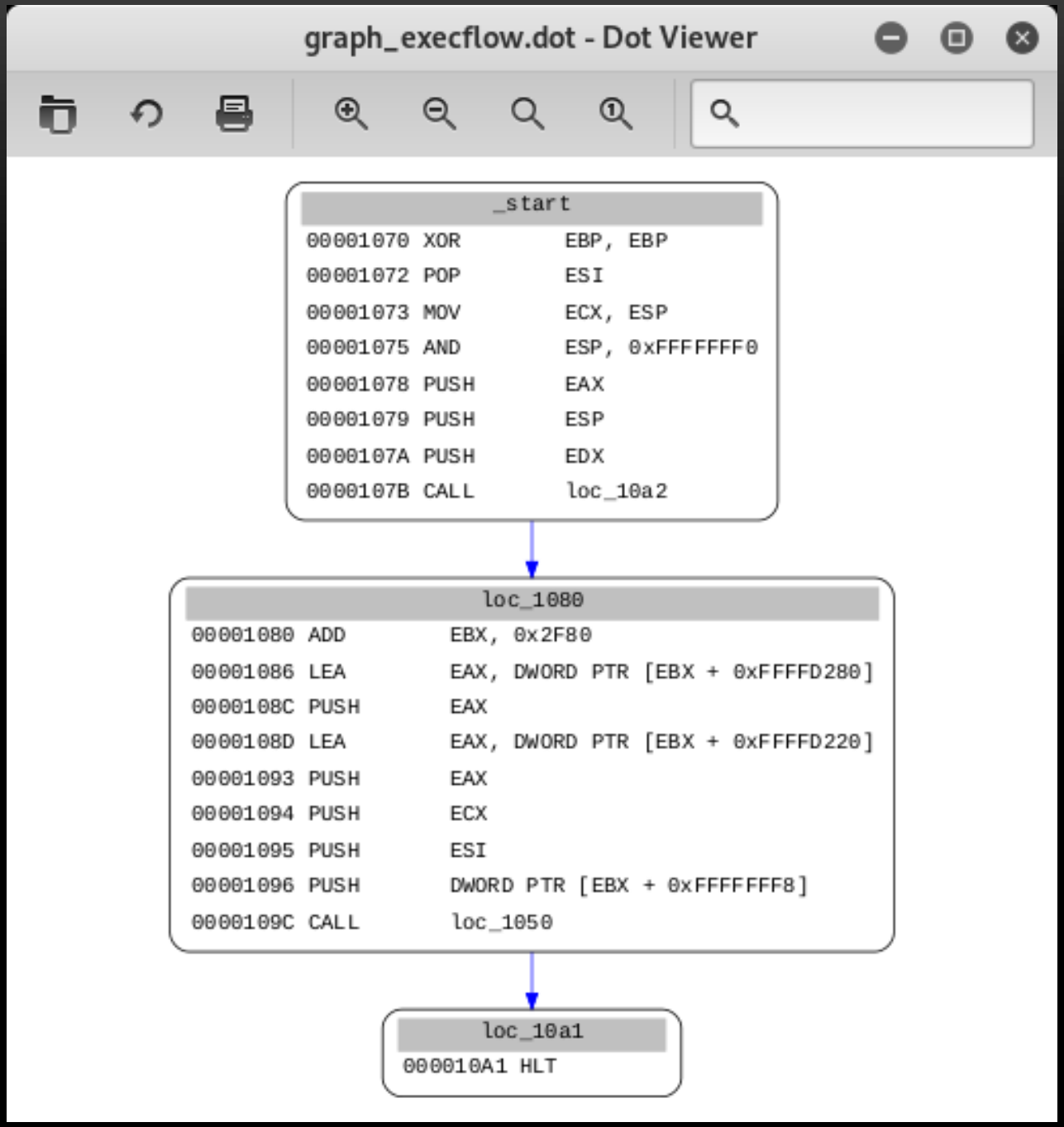
- ✓ Installing MIASM:

  - ✓ 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 0000000000001070 (0)
INFO : generate graph file
INFO : generate intervals
[0x1070 0x10A2]
INFO : total lines 0
```

- ✓ (testing MIASM) xdot graph_execflow.dot


graph_execflow.dot - Dot Viewer
_start
00001070 XOR EBP, EBP
00001072 POP ESI
00001073 MOV ECX, ESP
00001075 AND ESP, 0xFFFFFFF0
00001078 PUSH EAX
00001079 PUSH ESP
0000107A PUSH EDX
0000107B CALL loc_10a2
loc_1080
00001080 ADD EBX, 0x2F80
00001086 LEA EAX, DWORD PTR [EBX + 0xFFFFD280]
0000108C PUSH EAX
0000108D LEA EAX, DWORD PTR [EBX + 0xFFFFD220]
00001093 PUSH EAX
00001094 PUSH ECX
00001095 PUSH ESI
00001096 PUSH DWORD PTR [EBX + 0xFFFFFFF8]
0000109C CALL loc_1050
loc_10a1
000010A1 HLT

```
1  from miasm2.analysis.binary import Container
2  from miasm2.analysis.machine import Machine
3  from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE
4
5  with open("confidence2019.bin") as fdesc:
6      cont=Container.from_stream(fdesc)
7
8  machine=Machine('x86_32')
9  mdis=machine.dis_engine(cont.bin_stream)
10 ourblocks = mdis.dis_multiblock(0)
11 for block in ourblocks:
12     print block
13 jitter = machine.jitter("llvm")
14 jitter.init_stack()
15 s = open("confidence2019.bin").read()
16 run_addr = 0x40000000
17 jitter.cpu.EAX=3
18 jitter.cpu.ECX=6
19 jitter.vm.add_memory_page(run_addr, PAGE_READ | PAGE_WRITE, s)
20 def code_sentinelle(jitter):
21     jitter.run = False
22     jitter.pc = 0
23     return True
24 jitter.add_breakpoint(0x40000028, code_sentinelle)
25 jitter.push_uint32_t(0x40000028)
26 jitter.jit.log_regs = True
27 jitter.jit.log_mn = True
28 jitter.init_run(run_addr)
29 jitter.continue_run()
30
31 open('confidence2019_cfg.dot', 'w').write(ourblocks.dot())
32
```

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.

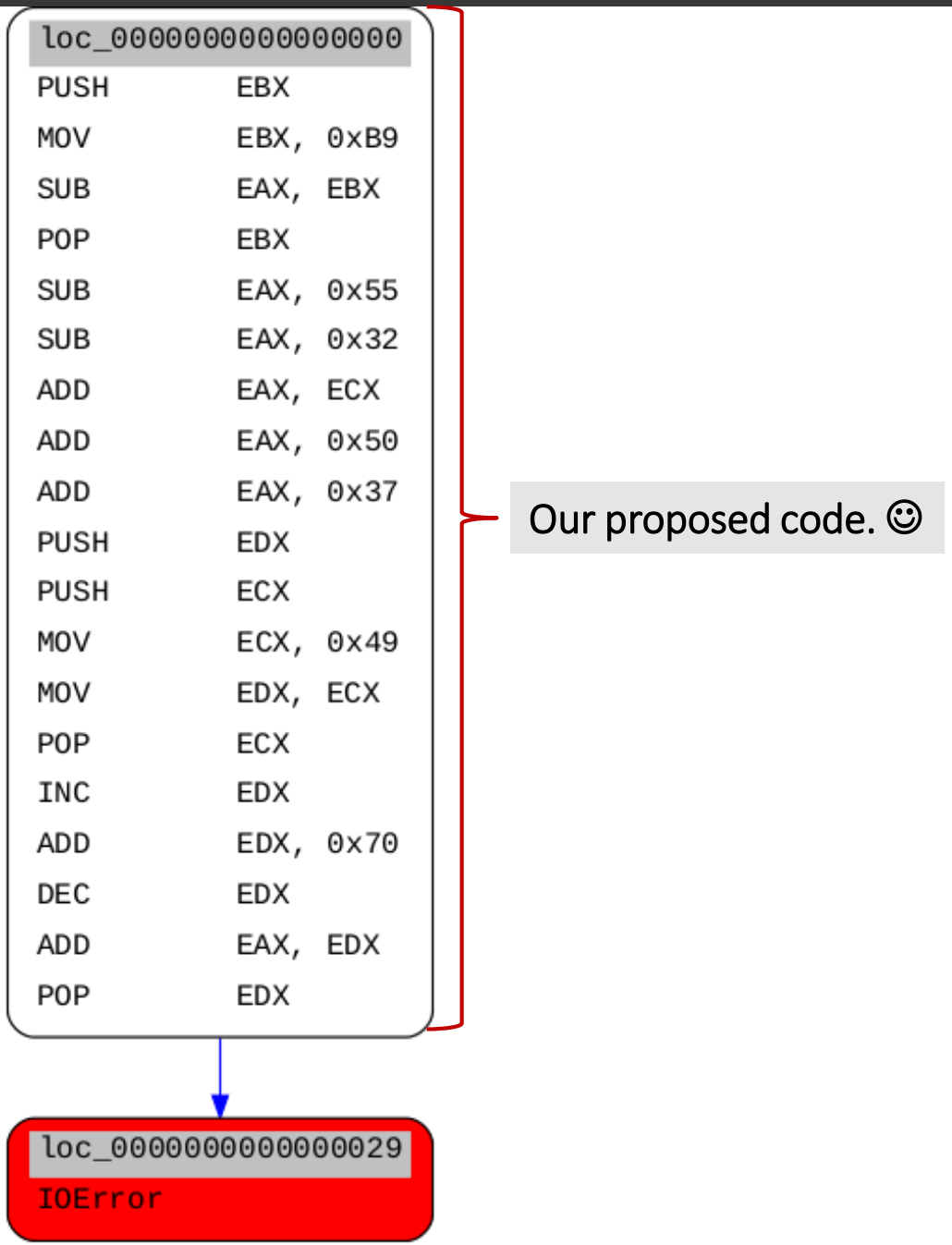
Run the emulation.

Generates a dot graph.

```
root@kali:~/programs/confidence# python miasm.py
WARNING: not enough bytes in str
WARNING: cannot disasm at 29
WARNING: not enough bytes in str
WARNING: cannot disasm at 29
loc_0000000000000000:0x00000000
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
->      c_next:loc_0000000000000029:0x00000029
loc_0000000000000029:0x00000029
```

Disassembling our code (again) ☺

```
40000000 PUSH        EBX
EAX 00000003 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000001 MOV         EBX, 0xB9
EAX 00000003 EBX 000000B9 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000006 SUB         EAX, EBX
EAX FFFFFF4A EBX 000000B9 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 1
40000008 POP         EBX
EAX FFFFFF4A EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 1
40000009 SUB         EAX, 0x55
EAX FFFFFEF5 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
4000000C SUB         EAX, 0x32
EAX FFFFFEC3 EBX 00000000 ECX 00000006 EDX 00000000 ESI 00000000 EDI 00000000
ESP 0123FFFC EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0



EAX FFFFFF50 EBX 00000000 ECX 00000006 EDX 00000049 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 1 of 0 cf 0
40000021 INC         EDX
EAX FFFFFF50 EBX 00000000 ECX 00000006 EDX 0000004A ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000022 ADD         EDX, 0x70
EAX FFFFFF50 EBX 00000000 ECX 00000006 EDX 000000BA ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000025 DEC         EDX
EAX FFFFFF50 EBX 00000000 ECX 00000006 EDX 000000B9 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 0
40000026 ADD         EAX, EDX
EAX 00000009 EBX 00000000 ECX 00000006 EDX 000000B9 ESI 00000000 EDI 00000000
ESP 0123FFF8 EBP 00000000 EIP 40000000 zf 0 nf 0 of 0 cf 1
```


loc_0000000000000000
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
Our proposed code. ☺
loc_0000000000000029
IOError

```
root@kali:~/programs/confidence# python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from miasm2.analysis.binary import Container
>>> from miasm2.analysis.machine import Machine
>>> from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE
>>> with open("confidence2019.bin") as fdesc:
...     cont=Container.from_stream(fdesc)
...
>>> confidencemach=Machine('x86_32')
>>> confidencedis=confidencemach.dis_engine(cont.bin_stream)
>>> myblocks = confidencedis.dis_multiblock(0)
WARNING: not enough bytes in str
WARNING: cannot disasm at 29
WARNING: not enough bytes in str
WARNING: cannot disasm at 29
>>> sym = confidencemach.ira( )
>>> for block in myblocks:
...     sym.add_block(block)
...
[<miasm2.ir.ir.IRBlock object at 0x7f16d22188c0>]
[]
>>> from miasm2.ir.symbexec import SymbolicExecutionEngine
>>> symb = SymbolicExecutionEngine(sym,confidencemach.mn.regs.regs_init)
>>> symbolic_pc = symb.run_at(0, step=True)
```

Get the IRA converter.

Initialize and run the Symbolic Execution Engine.

```
>>> symbolic_pc = symb.run_at(0, step=True)
Instr PUSH       EBX
Assignblk:
ESP = ESP + -0x4
@32[ESP + -0x4] = EBX

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

________________________________________________________________________________
Instr MOV        EBX, 0xB9
Assignblk:
EBX = 0xB9

________________________________________________________________________________
ESP                 = ESP_init + 0xFFFFFFFC
EBX                 = 0xB9
@32[ESP_init + 0xFFFFFFFC] = EBX_init

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

```
EAX                 = EAX_init + ECX_init
cf                  = ((((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF4
7)) & ((EAX_init + ECX_init + 0xFFFFFF47) ^ 0xFFFFFF46)) ^ (EAX_init + ECX_init)
 ^ (EAX_init + ECX_init + 0xFFFFFF47) ^ 0xB9)[31:32]
pf                  = parity((EAX_init + ECX_init) & 0xFF)
zf                  = (EAX_init + ECX_init)?(0x0,0x1)
af                  = ((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF47)
 ^ 0xB9)[4:5]
of                  = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF47
)) & ((EAX_init + ECX_init + 0xFFFFFF47) ^ 0xFFFFFF46))[31:32]
nf                  = (EAX_init + ECX_init)[31:32]
@32[ESP_init + 0xFFFFFFF8] = ECX_init
@32[ESP_init + 0xFFFFFFFC] = EDX_init

Instr POP         EDX
Assignblk:
IRDst = loc_0000000000000029:0x00000029

EAX                 = EAX_init + ECX_init
cf                  = ((((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF4
7)) & ((EAX_init + ECX_init + 0xFFFFFF47) ^ 0xFFFFFF46)) ^ (EAX_init + ECX_init)
 ^ (EAX_init + ECX_init + 0xFFFFFF47) ^ 0xB9)[31:32]
pf                  = parity((EAX_init + ECX_init) & 0xFF)
zf                  = (EAX_init + ECX_init)?(0x0,0x1)
af                  = ((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF47)
 ^ 0xB9)[4:5]
IRDst               = 0x29
of                  = (((EAX_init + ECX_init) ^ (EAX_init + ECX_init + 0xFFFFFF47
)) & ((EAX_init + ECX_init + 0xFFFFFF47) ^ 0xFFFFFF46))[31:32]
nf                  = (EAX_init + ECX_init)[31:32]
@32[ESP_init + 0xFFFFFFF8] = ECX_init
@32[ESP_init + 0xFFFFFFFC] = EDX_init
```

The same conclusion from our previous tests. ☺

TRITON

❑ TRITON

- ✓ It can be downloaded from https://triton.quarkslab.com/
- ✓ Based on Intel Pin instrumentation tool: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

- ✓ 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 https://software.intel.com/sites/landingpage/pintool/downloads/pin-2.14-71313-gcc.4.4.7-linux.tar.gz
  - ✓ 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).

```
1  #!/usr/bin/env python2
2  ## -*- coding: utf-8 -*-
3  ##
4
5  from __future__ import print_function
6  from triton     import TritonContext, ARCH, Instruction, MemoryAccess, CPUSI
   ZE, OPERAND, REG
7
8  import sys
9
10 # We define the code to be handled and symbolic executed
11
12 mycode = [
13
14         (0x400000, b"\x53"),                    # push ebx
15         (0x400001, b"\xbb\xb9\x00\x00\x00"),    # mov ebx, 0xB9
16         (0x400006, b"\x29\xd8"),                # sub eax, ebx
17         (0x400008, b"\x5b"),                    # pop ebx
18         (0x400009, b"\x83\xe8\x55"),            # sub eax, 0x55
19         (0x40000c, b"\x83\xe8\x32"),            # sub eax, 0x32
20         (0x40000f, b"\x01\xc8"),                # add eax, ecx
21         (0x400011, b"\x83\xc0\x50"),            # add eax, 0x50
22         (0x400014, b"\x83\xc0\x37"),            # add eax, 0x37
23         (0x400017, b"\x52"),                    # push edx
24         (0x400018, b"\x51"),                    # push ecx
25         (0x400019, b"\xb9\x49\x00\x00\x00"),    # mov ecx, 0x49
26         (0x40001e, b"\x89\xca"),                # mov edx, ecx
27         (0x400020, b"\x59"),                    # pop ecx
28         (0x400021, b"\x42"),                    # inc edx
29         (0x400022, b"\x83\xc2\x70"),            # add edx, 0x70
30         (0x400025, b"\x4a"),                    # dec edx
31         (0x400026, b"\x01\xd0"),                # add eax, edx
32         (0x400028, b"\x5a"),                    # pop edx
33         (0x400029, b"\xff\xe0"),                # jmp eax
34
35 ]
36
```

```
37
38 if __name__ == '__main__':
39
40     #Set the context for Triton functions
41     context = TritonContext()
42
43     # Set the architecture. In our case, we are using x86 32-bit
44     context.setArchitecture(ARCH.X86)
45
46     for (addr, opcode) in mycode:
47         # Build an instruction object.
48         instruction = Instruction()
49
50         # Setup the opcode
51         instruction.setOpcode(opcode)
52
53         # Setup start  address
54         instruction.setAddress(addr)
55
56         # Process our code
57         context.processing(instruction)
58
59         print('----------------------------------------')
60         print('The current IP: ', instruction)
61         pc = context.getRegisterAst(context.registers.eip).evaluate()
62         print ('The next IP is: ', hex(pc))
63         print('----------------------------------------\n\n')
64
65         # Display each instruction, determine the operation type and show opcode in
   formation
66         print('>>> %s'% instruction)
67
68         print('\n  ----------------------------')
69         print('    Is a memory read?  :', instruction.isMemoryRead())
70         print('    Is a memory write? :', instruction.isMemoryWrite())
71         print('    --------------------------\n')
```

```
72
73          for op_entry in instruction.getOperands():
74               print('    %s' % (op_entry))
75               if op_entry.getType() == OPERAND.MEM:
76                    print('       segment :', op_entry.getSegmentRegister())
77                    print('         base  : %s' % (op_entry.getBaseRegister()))
78                    print('         index : %s' % (op_entry.getIndexRegister()))
79                    print('         disp  : %s' % (op_entry.getDisplacement()))
80                    print('         scale : %s' % (op_entry.getScale()))
81          print('')
82
83
84          # Display each one of the symbolic expressions
85          for expression in instruction.getSymbolicExpressions():
86              print('\t', expression)
87
88          print()
89
90      print()
91      print('Registers information')
92      print('*********************************')
93      for k, v in list(context.getSymbolicRegisters().items()):
94          print(context.getRegister(k), v)
95
96      print()
97      print('Summary Memory information')
98      print('*****************************')
99      for k, v in list(context.getSymbolicMemory().items()):
100         print(hex(k), v)
101
102     print()
103
104     sys.exit(0)
```

```
root@kali:~# rasm2 -a x86 -b 32 "push ebx"
53
root@kali:~# rasm2 -a x86 -b 32 "mov ebx, 0xb9"
bbb9000000
root@kali:~# rasm2 -a x86 -b 32 "sub eax, ebx"
29d8
root@kali:~# rasm2 -a x86 -b 32 "pop ebx"
5b
root@kali:~# rasm2 -a x86 -b 32 "sub eax, 0x55"
83e855
root@kali:~# rasm2 -a x86 -b 32 "sub eax, 0x32"
83e832
root@kali:~# rasm2 -a x86 -b 32 "add eax, ecx"
01c8
root@kali:~# rasm2 -a x86 -b 32 "add eax, 0x50"
83c050
root@kali:~# rasm2 -a x86 -b 32 "add eax, 0x37"
83c037
root@kali:~# rasm2 -a x86 -b 32 "push edx"
52
root@kali:~# rasm2 -a x86 -b 32 "push ecx"
51
root@kali:~# rasm2 -a x86 -b 32 "mov ecx, 0x49"
b949000000
root@kali:~# rasm2 -a x86 -b 32 "mov edx, ecx"
89ca
root@kali:~# rasm2 -a x86 -b 32 "pop ecx"
59
root@kali:~# rasm2 -a x86 -b 32 "inc edx"
42
root@kali:~# rasm2 -a x86 -b 32 "add edx, 0x70"
83c270
root@kali:~# rasm2 -a x86 -b 32 "dec edx"
4a
root@kali:~# rasm2 -a x86 -b 32 "add eax, edx"
01d0
root@kali:~# rasm2 -a x86 -b 32 "pop edx"
5a
root@kali:~# rasm2 -a x86 -b 32 "jmp eax"
ffe0
```

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.

```
root@ubuntu19:~/pin214/source/tools/Triton/src/examples/python# python confidence_sym.py | more
----------------------------------------
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) (bvsub (_ bv0 32) (_ bv4 32))) ; Stack alignment
        (define-fun ref!1 () (_ BitVec 8) ((_ extract 31 24) (_ bv0 32))) ; Byte reference - P
USH operation
        (define-fun ref!2 () (_ BitVec 8) ((_ extract 23 16) (_ bv0 32))) ; Byte reference - P
USH operation
        (define-fun ref!3 () (_ BitVec 8) ((_ extract 15 8) (_ bv0 32))) ; Byte reference - PU
SH operation
        (define-fun ref!4 () (_ BitVec 8) ((_ extract 7 0) (_ bv0 32))) ; Byte reference - PUS
H 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

----------------------------------------
The current IP:  0x400001: mov ebx, 0xb9
The next IP is:  0x400006
----------------------------------------
```

byte by byte ☺

```
>>> 0x400001: mov ebx, 0xb9

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

    ebx:32 bv[31..0]
    0xb9:32 bv[31..0]

        (define-fun ref!7 () (_ BitVec 32) (_ bv185 32)) ; MOV operation
        (define-fun ref!8 () (_ BitVec 32) (_ bv4194310 32)) ; Program Counter

----------------------------------------
The current IP:  0x400006: sub eax, ebx
The next IP is:  0x400008
----------------------------------------

>>> 0x400006: sub eax, ebx

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

    eax:32 bv[31..0]
    ebx:32 bv[31..0]

        (define-fun ref!9 () (_ BitVec 32) (bvsub (_ bv0 32) ref!7)) ; SUB operation
        (define-fun ref!10 () (_ BitVec 1) (ite (= (_ bv16 32) (bvand (_ bv16 32) (bvxor ref!9 (bvxor (_ bv0 32)
ref!7)))) (_ bv1 1) (_ bv0 1))) ; Adjust flag
        (define-fun ref!11 () (_ BitVec 1) ((_ extract 31 31) (bvxor (bvxor (_ bv0 32) (bvxor ref!7 ref!9)) (bvan
d (bvxor (_ bv0 32) ref!9) (bvxor (_ bv0 32) ref!7))))) ; Carry flag
        (define-fun ref!12 () (_ BitVec 1) ((_ extract 31 31) (bvand (bvxor (_ bv0 32) ref!7) (bvxor (_ bv0 32) r
ef!9)))) ; Overflow flag
        (define-fun ref!13 () (_ BitVec 1) (bvxor (bvxor (bvxor (bvxor (bvxor (bvxor (bvxor (bvxor (_ bv1 1) ((_
extract 0 0) (bvlshr ((_ extract 7 0) ref!9) (_ bv0 8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!9) (_ bv1
8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!9) (_ bv2 8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!
9) (_ bv3 8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!9) (_ bv4 8)))) ((_ extract 0 0) (bvlshr ((_ extract
 7 0) ref!9) (_ bv5 8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!9) (_ bv6 8)))) ((_ extract 0 0) (bvlshr (
(_ extract 7 0) ref!9) (_ bv7 8))))) ; Parity flag
        (define-fun ref!14 () (_ BitVec 1) ((_ extract 31 31) ref!9)) ; Sign flag
        (define-fun ref!15 () (_ BitVec 1) (ite (= ref!9 (_ bv0 32)) (_ bv1 1) (_ bv0 1))) ; Zero flag
        (define-fun ref!16 () (_ BitVec 32) (_ bv4194312 32)) ; Program Counter
```

0xb9 == 185 ☺

eax

ALEXANDRE BORGES – MALWARE AND SECURITY RESEARCHER

```
Registers information
******************************
esp:32 bv[31..0] (define-fun ref!112 () (_ BitVec 32) (bvadd ref!79 (_ bv4 32))) ; Stack align
ment
cf:1 bv[0..0] (define-fun ref!105 () (_ BitVec 1) ((_ extract 31 31) (bvxor (bvand ref!52 ref!
96) (bvand (bvxor (bvx
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) (bvxo
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)) ;
POP 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) (bvlshr ((_ extract 7 0) ref!103) (_ bv0 8)))) ((_ extract 0 0) (bvlshr ((_ e
xtract 7 0) ref!103) (
_ bv1 8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!103) (_ bv2 8)))) ((_ extract 0 0) (
bvlshr ((_ extract 7 0
) ref!103) (_ bv3 8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!103) (_ bv4 8)))) ((_ ex
tract 0 0) (bvlshr ((_
 extract 7 0) ref!103) (_ bv5 8)))) ((_ extract 0 0) (bvlshr ((_ extract 7 0) ref!103) (_ bv6
8)))) ((_ extract 0 0)
 (bvlshr ((_ extract 7 0) ref!103) (_ bv7 8))))) ; Parity flag
```

```python
#!/usr/bin/env python2
## -*- coding: utf-8 -*-
##

from __future__ import print_function
from triton     import TritonContext, ARCH, Instruction, MODE

import sys

#Define the code to be emulated

mycode = {

        0x400000: b"\x53",                      # push ebx
        0x400001: b"\xbb\xb9\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()

```

```
39
40 # This function emulates the code.
41 def confidence(pc):
42     while pc in mycode:
43         # Build an instruction
44         instruction = Instruction()
45
46         # Setup the opcode
47         instruction.setOpcode(mycode[pc])
48
49         # Setup start address
50         instruction.setAddress(pc)
51
52         # Process the opcodes
53         context.processing(instruction)
54
55         # Display the instruction
56         print('Curr pc:', instruction)
57
58         # Set the IP to next instruction and update the some registers
59         pc = context.getRegisterAst(context.registers.eip).evaluate()
60         eax = context.getRegisterAst(context.registers.eax).evaluate()
61         ebx = context.getRegisterAst(context.registers.ebx).evaluate()
62         ecx = context.getRegisterAst(context.registers.ecx).evaluate()
63         edx = context.getRegisterAst(context.registers.edx).evaluate()
64         print('Next pc: ', hex(pc))
65         print('Next eax:', hex(eax))
66         print('Next ebx:', hex(ebx))
67         print('Next ecx:', hex(ecx))
68         print('Next edx:', hex(edx))
69         print()
70     return
71
```

```
72 # This function initializes the context memory. EAX and ECX was randomly chosen.
73 def startCtx():
74     context.setConcreteRegisterValue(context.registers.esp, 0x7fffffff)
75     context.setConcreteRegisterValue(context.registers.ebp, 0x7fffffff)
76     context.setConcreteRegisterValue(context.registers.eax, 0x2)
77     context.setConcreteRegisterValue(context.registers.ebx, 0x0)
78     context.setConcreteRegisterValue(context.registers.ecx, 0x7)
79     context.setConcreteRegisterValue(context.registers.edx, 0x0)
80     return
81
82 if __name__ == '__main__':
83     # Set the architecture. In our case, we have chosen x86 32-bit.
84     context.setArchitecture(ARCH.X86)
85
86     # Align the memory
87     context.enableMode(MODE.ALIGNED_MEMORY, True)
88
89     # Define the entry point address
90     entrypoint = 0x400000
91
92     # Set the memory context
93     startCtx()
94
95     # Run the emulation
96     confidence(entrypoint)
97
98     sys.exit(0)
```

```
root@ubuntu19:~/pin214/source/tools/Triton/src/examples/python# python confidence_sym_2.py
Curr pc: 0x400000: push ebx
Next pc:  0x400001
Next eax: 0x2
Next ebx: 0x0
Next ecx: 0x7
Next edx: 0x0

Curr pc: 0x400001: mov ebx, 0xb9
Next pc:  0x400006
Next eax: 0x2
Next ebx: 0xb9
Next ecx: 0x7
Next edx: 0x0

Curr pc: 0x400006: sub eax, ebx
Next pc:  0x400008
Next eax: 0xffffff49
Next ebx: 0xb9
Next ecx: 0x7
Next edx: 0x0


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

Curr ip: 0x400029: jmp eax
Next ip: 0x9
Next eax: 0x9
Next ebx: 0x0
Next ecx: 0x7
Next edx: 0x0
```

RADARE2 + MIASM

```
root@kali:~/programs/confidence# r2 -b 32 confidence2019.bin
 -- In soviet russia, radare2 debugs you!
[0x00000000]> aaa

[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] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00000000]> ec comment yellow

[0x00000000]> e asm.emu=true

[0x00000000]> pdf

┌ (fcn) fcn.00000000 41
│   fcn.00000000 ();
│           0x00000000      53              push ebx
│           0x00000001      bbb9000000      mov ebx, 0xb9
│           0x00000006      29d8            sub eax, ebx
sf=0x1 -> 0xb9bb ; zf=0x0 ; pf=0x1 -> 0xb9bb ; cf=0x1 -> 0xb9bb
│           0x00000008      5b              pop ebx
│           0x00000009      83e855          sub eax, 0x55
=0x1 -> 0xb9bb ; zf=0x0 ; pf=0x0 ; cf=0x0
│           0x0000000c      83e832          sub eax, 0x32
=0x1 -> 0xb9bb ; zf=0x0 ; pf=0x1 -> 0xb9bb ; cf=0x0
│           0x0000000f      01c8            add eax, ecx
> 0xb9bb ; zf=0x0 ; cf=0x0 ; pf=0x1 -> 0xb9bb
│           0x00000011      83c050          add eax, 0x50
=0x1 -> 0xb9bb ; zf=0x0 ; cf=0x0 ; pf=0x0
│           0x00000014      83c037          add eax, 0x37
=0x1 -> 0xb9bb ; zf=0x0 ; cf=0x0 ; pf=0x1 -> 0xb9bb
│           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
0 ; pf=0x0
```

ESIL comments, which came from MIASM and are converted to R2

```
; esp=0xfffffffffffffffc
; 185 ; ebx=0xb9
; eax=0xffffffffffffff47 ; of=0x0 ;

; ebx=0xffffffff ; esp=0x100000000
; 'U' ; eax=0xfffffef2 ; of=0x0 ; sf

; '2' ; eax=0xfffffec0 ; of=0x0 ; sf

; eax=0xfffffec0 ; of=0x0 ; sf=0x1 -

; 'P' ; eax=0xffffff10 ; of=0x0 ; sf

; '7' ; eax=0xffffff47 ; of=0x0 ; sf

; esp=0xfffffffffffffffc
; esp=0xfffffff8
; 'I' ; 73 ; ecx=0x49
; edx=0x49
; ecx=0xffffffff ; esp=0xfffffffc
; edx=0x4a ; of=0x0 ; sf=0x0 ; zf=0x
```

```
[0x00000000]> aer eax=0x7
[0x00000000]> aer ecx=0x2
[0x00000000]> e io.cache = true
[0x00000000]> aes
[0x00000000]> aer
oeax = 0x00000000
eax = 0x00000007
ebx = 0x00000000
ecx = 0x00000002
edx = 0x00000000
esi = 0x00000000
edi = 0x00000000
esp = 0xfffffffc
ebp = 0x00000000
eip = 0x00000001
eflags = 0x00000000
```

- ✓ aer: handle ESIL registers (set and show)
- ✓ aes: perform emulated debugger step
- ✓ aecu: continue until address

```
[0x00000000]> e asm.emu=true
[0x00000000]> aecu 0x00000028
[0x00000000]> aer
oeax = 0x00000000
eax = 0x00000009
ebx = 0x00000000
ecx = 0x00000002
edx = 0x000000b9
esi = 0x00000000
edi = 0x00000000
esp = 0xfffffffc
ebp = 0x00000000
eip = 0x00000028
eflags = 0x00000005
```

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?

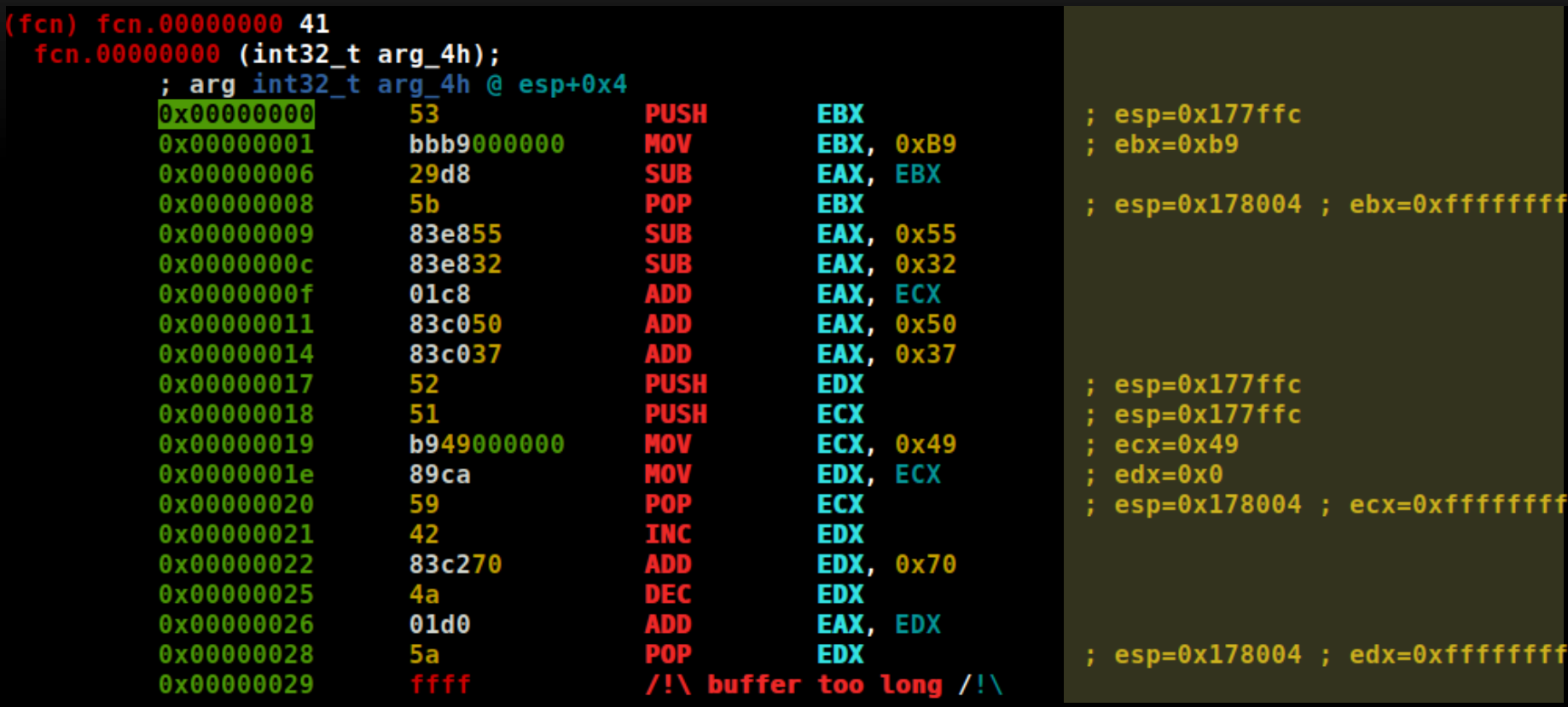
- ✓ apt-get install docker
- ✓ git clone https://github.com/radare/radare2.git
- ✓ cd radare2/
- ✓ sys/install.sh
- ✓ Install MIASM
- ✓ pip install cffi
- ✓ pip install jinja2
- ✓ docker pull guedou/r2m2
- ✓ docker run --rm -it -e 'R2M2_ARCH=x86_32' guedou/r2m2 bash

- ✓ [r2m2@fd5662d151e4 ~]$ pwd

- ✓ (another terminal) docker ps -a
- ✓ (another terminal) docker cp /root/confidence2019.bin fd5662d151e4:/home/r2m2/confidence2019.bin

- ✓ [r2m2@fd5662d151e4 ~]$ export R2M2_ARCH=x86_32
- ✓ [r2m2@fd5662d151e4 ~]$ r2 -A -b 32 -a r2m2 confidence2019.bin

```
[r2m2@5aab993be8b4 ~]$ r2 -A -b 32 -a r2m2 confidence2019.bin
[/home/r2m2/miasm/miasm/expression/expression.py:924: UserWarning: DEPRE
CATION WARNING: use exprmem.ptr instead of exprmem.arg
  warnings.warn('DEPRECATION WARNING: use exprmem.ptr instead of exprmem
.arg')
[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.
 -- In soviet russia, radare2 debugs you!
[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 ; esp=0x177ffc
0x00000001 bbb9000000 MOV EBX, 0xB9 ; ebx=0xb9
0x00000006 29d8 SUB EAX, EBX
0x00000008 5b POP EBX ; esp=0x178004 ; ebx=0xffffffff
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 ; esp=0x177ffc
0x00000018 51 PUSH ECX ; esp=0x177ffc
0x00000019 b949000000 MOV ECX, 0x49 ; ecx=0x49
0x0000001e 89ca MOV EDX, ECX ; edx=0x0
0x00000020 59 POP ECX ; esp=0x178004 ; ecx=0xffffffff
0x00000021 42 INC EDX
0x00000022 83c270 ADD EDX, 0x70
0x00000025 4a DEC EDX
0x00000026 01d0 ADD EAX, EDX
0x00000028 5a POP EDX ; esp=0x178004 ; edx=0xffffffff
0x00000029 ffff /!\ buffer too long /!\

ANTI-VM

- ✓ 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:

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

  - ✓ Using WMI to query the Win32_BIOS management class to interact with attributes from the physical machine.

- ✓ 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 biosClass =
            new ManagementClass("Win32_BIOS");
            ManagementObjectCollection biosc =
                biosClass.GetInstances();
            ManagementObjectCollection.ManagementObjectEnumerator
                biosEnumerator =
                biosc.GetEnumerator();
            while (biosEnumerator.MoveNext())
            {
                ManagementObject biosc1 =
                    (ManagementObject)biosEnumerator.Current;
                Console.WriteLine(
                    "Attributes:\n\n" + "Version:\t " + biosc1["version"].ToString( ));
                Console.WriteLine(
                    "SerialNumber:\t " + biosc1["SerialNumber"].ToString());
                Console.WriteLine(
                    "OperatingSystem:\t " + biosc1["TargetOperatingSystem"].ToString());
                Console.WriteLine(
                    "Manufacturer:\t " + biosc1["Manufacturer"].ToString());
            }
            //return 0;
        }
    }
}
```

- ✓ 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:

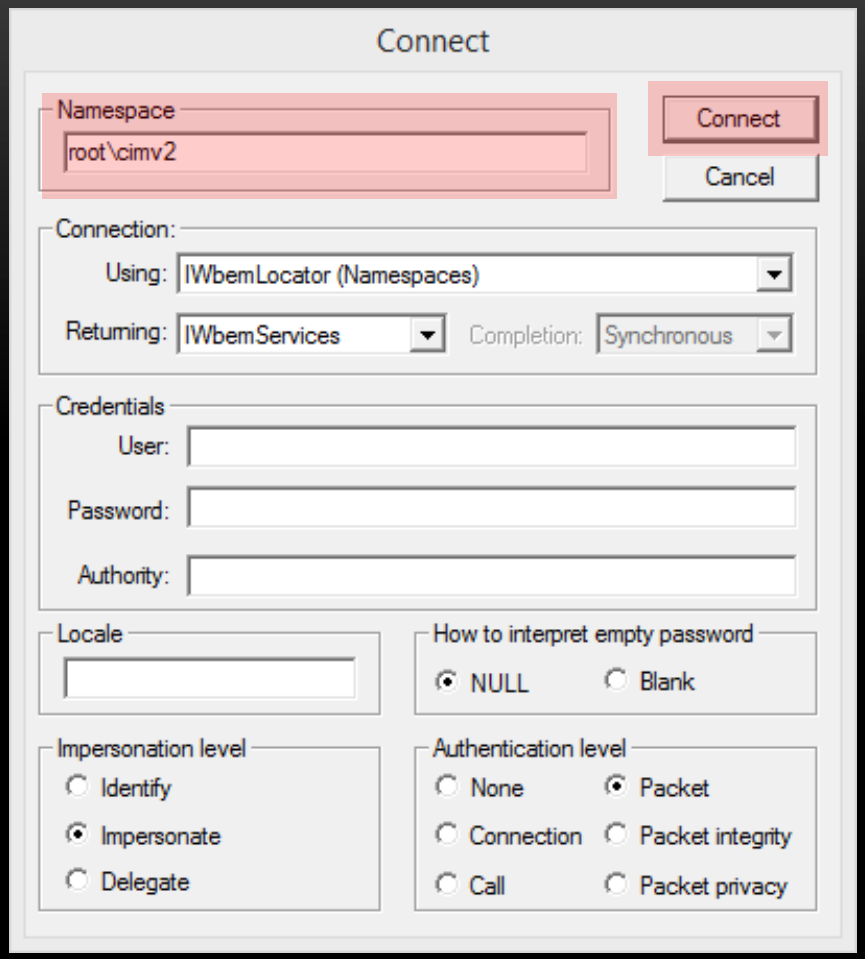
```
C:\> Test_VM.exe
Attributes:
Version:        DELL   - 6222004
SerialNumber:   D5965S1
OperatingSystem:        0
Manufacturer:   Dell Inc.
```

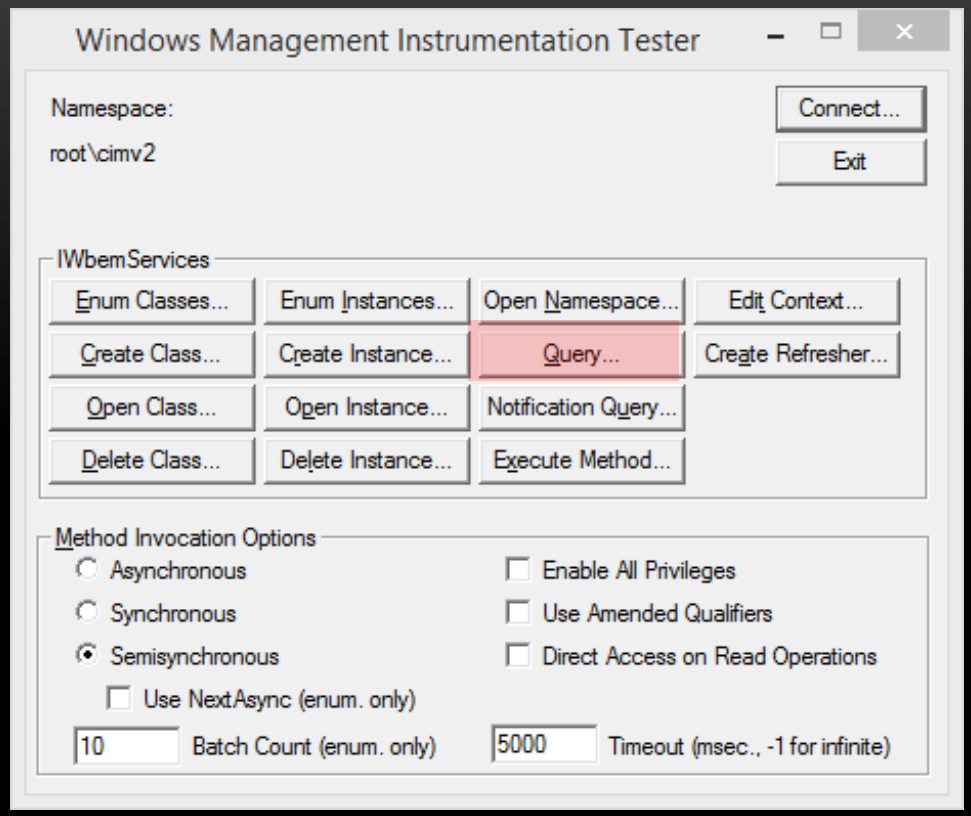
❑ Guest virtual machine:

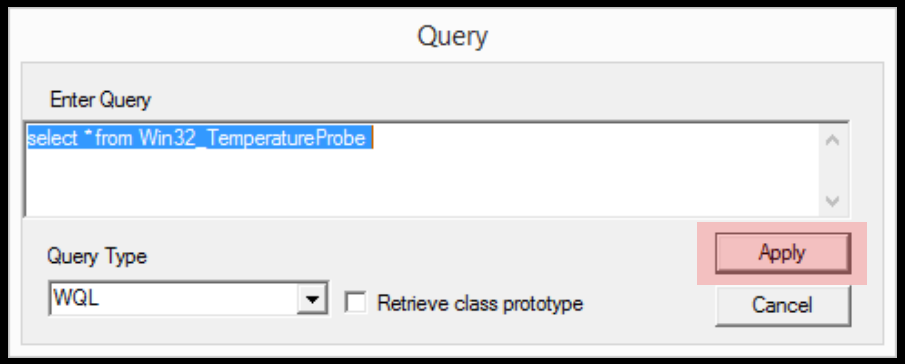
```
E:\> Test_VM.exe
Attributes:
Version:        LENOVO - 6040000
SerialNumber:   VMware-56 4d 8d c3 a7 c7 e5
2b-39 d6 cc 93 bf 90 28 2d
OperatingSystem:        0
Manufacturer:   Phoenix Technologies LTD
```

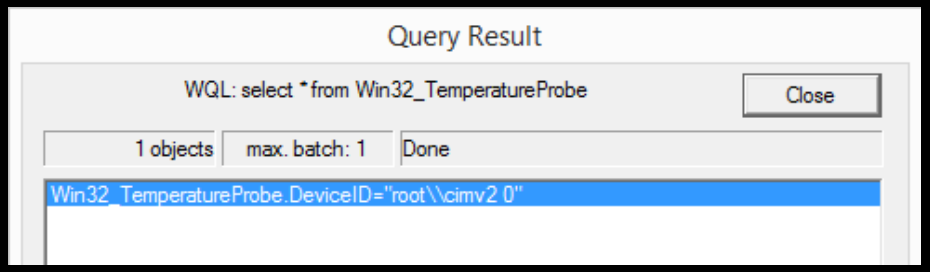
```
namespace TestVM_3
{
    class Program
    {
        static void Main(string[] args)
        {
            ManagementClass tempClass =
            new ManagementClass("Win32_TemperatureProbe");
            ManagementObjectCollection tempinstance =
                tempClass.GetInstances() ;
     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

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

                }
            }
        }
    }
}
```

```
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
```

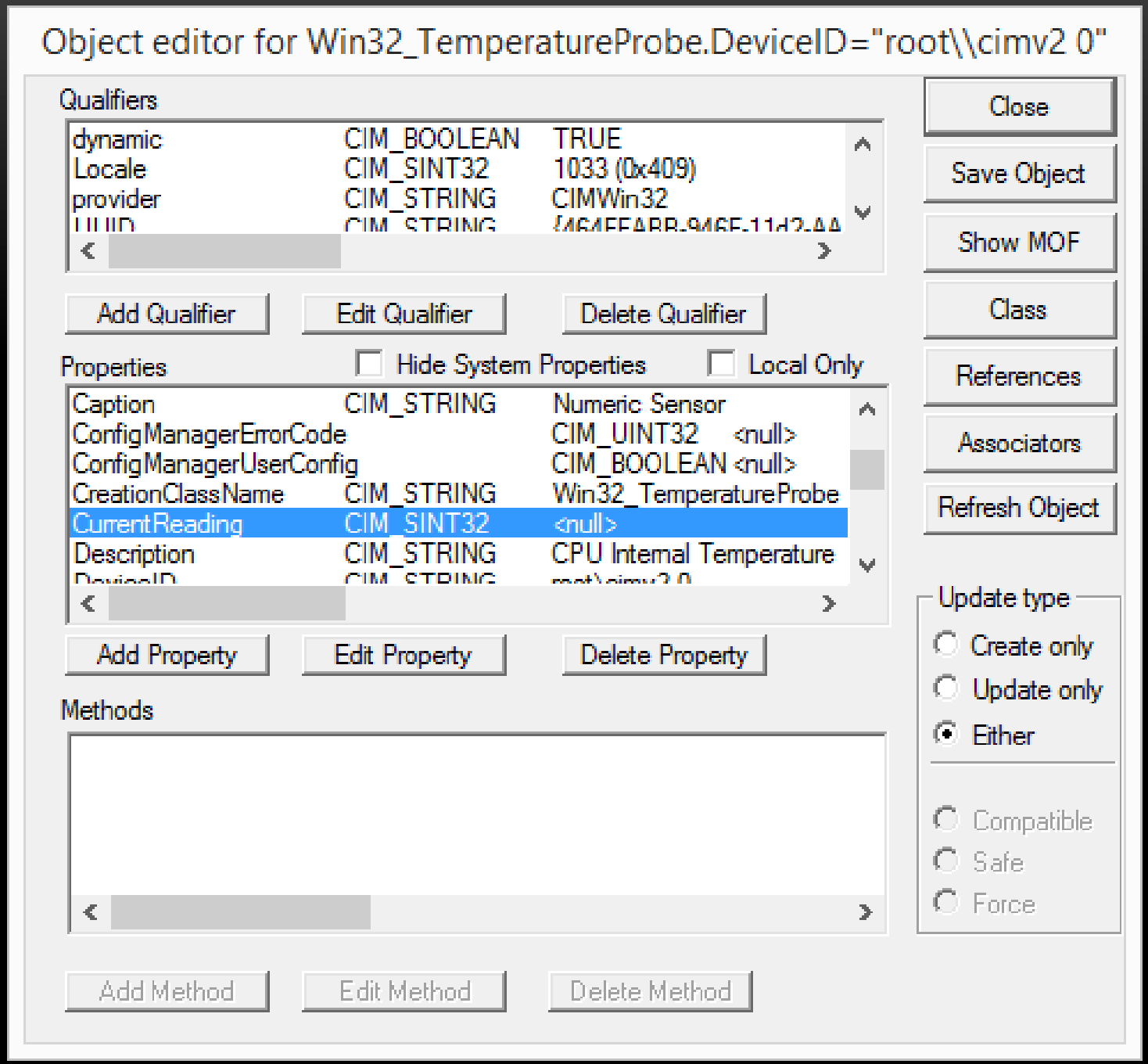






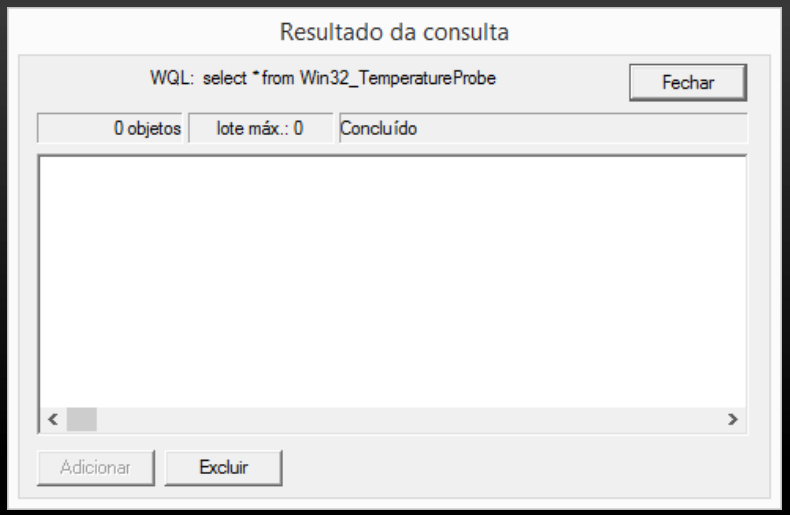


Double-click the result....

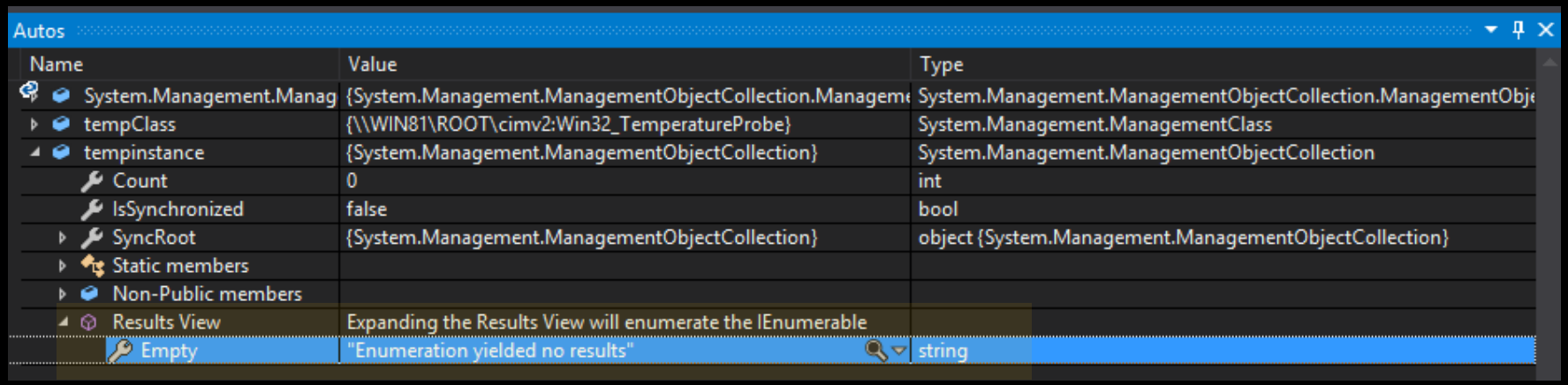

Object editor for Win32_TemperatureProbe.DeviceID="root\\cimv2 0"
Qualifiers
dynamic CIM_BOOLEAN TRUE
Locale CIM_SINT32 1033 (0x409)
provider CIM_STRING CIMWin32
Add Qualifier
Edit Qualifier
Delete Qualifier
Properties
Hide System Properties
Local Only
Caption CIM_STRING Numeric Sensor
ConfigManagerErrorCode CIM_UINT32 <null>
ConfigManagerUserConfig CIM_BOOLEAN <null>
CreationClassName CIM_STRING Win32_TemperatureProbe
CurrentReading CIM_SINT32 <null>
Description CIM_STRING CPU Internal Temperature
Add Property
Edit Property
Delete Property
Methods
Add Method
Edit Method
Delete Method
Close
Save Object
Show MOF
Class
References
Associators
Refresh Object
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.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!");
        }
    }
}
```

[26]	{System.Management.PropertyData}	object {System.Management.PropertyData}
[27]	{System.Management.PropertyData}	object {System.Management.PropertyData}
IsArray	false	bool
IsLocal	true	bool
Name	"Status"	string
Origin	"CIM_ManagedSystemElement"	string
Qualifiers	{System.Management.QualifierDataCollection}	System.Management.QualifierDataCollection
Type	String	System.Management.CimType
Value	"OK"	object {string}
Non-Public membe		



- ✓ 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:

✓CONFidence staff, which since the beginning has been very kind and professional.

✓You, who reserved some time attend my talk.

✓Remember: the best of this life are people. ☺

- ✓ Malware and Security Researcher.
- ✓ Speaker at DEF CON USA 2018
- ✓ Speaker at DEF CON China 2019
- ✓ Speaker at HITB2019 Amsterdam
- ✓ 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

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