INTRODUCTION TO ADVANCED THREATS

BHACK 2020 LIVE

BLACKSTORM SECURITY
Providing Skills for Hackers
Agenda:

- Introduction
- Reversing
- Anti-Reversing
- De-obfuscation

Cyber Security Researcher
Speaker at DEF CON USA 2019
Speaker at DEF CON USA 2018
Speaker at DEF CON CHINA 2019
Speaker at DC2711 (Johannesburg)
Speaker at NO HAT 2019 (Italy)
Speaker at HITB 2019 (Amsterdam)
Speaker at CONFidence 2019 (Poland)
Speaker at DevCon 2020
Speaker at SANS 2020
Speaker at DevOpsDays BH 2019
Speaker at H2HC 2016/2015
Speaker at BHACK 2019/2018
Researcher on Android/iOS Reversing, Rootkits and Digital Forensics.
Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response
Last conferences:

- DEF CON USA 2019 (Las Vegas / USA)
- CONFidence Conference 2019 (Krakow / Poland)
- DEF CON China 2019 (Beijing / China)
- HITB Security Conference 2019 (Amsterdam)
- BSIDES Sao Paulo 2019 (Sao Paulo / Brazil)
- DEF CON USA 2018 (Las Vegas / USA)
- DEVCON 2020 (ONLINE)
- SANS 2020 (ONLINE)
- BHACK 2019 (Belo Horizonte/Brazil)
- DC2711 (Johannesburg/South Africa)
- NO HAT Conference 2019 (Bergamo/Italy)

Malwoverview Tool: https://github.com/alexandreborges/malwoverview
INTRODUCTION
During reverse engineering of a malware sample, we need to understand few aspects on the threat:

- Is the malware packed?
- Are DLLs and functions resolved dynamically?
- Are strings encrypted?
- Are there any anti-forensic techniques such as anti-vm, anti-debugging or anti-disassembly?
- Is there any obfuscation technique being used?

Unpacking malware is usually easy, but you might find sophisticated packers...
INTRODUCTION

Advanced threads are different from any daily malware because:

- They don’t use common packers.
- Most of the time, they bring malicious device drives (rootkits).
- Sometimes they try to compromise the platform (bootkits).
- They can use 0-days to exploit the infrastructure and systems.
- They bypass most of defenses and run under the radar.
- C2 transmit beacons and data once per week with short duration.
- They might implement tricks to prevent any memory acquisition.
- Most certainly, there’ll be anti-forensic techniques.
- It’s hard and take so much time to reverse it.

- If you have luck, so you’ll have the opportunity to analyze them.
There’re many packers that we know about to unpack them or, at least, how to manage them (but it can be hard...):

- **Native code**: ASPack, Armadillo, Petite, FSG, UPX, MPRESS, NSPack, PECompact, WinUnpack and so on...

- **.NET packers and obfuscators**: .NET Reactor, Salamander .NET Obfuscator, Dotfuscator, Smart Assembly, CryptoObfuscator for .NET, Agile, ArmDot, babelfor.NET, Eazfuscator.NET, Spice.Net, Skater.NET, VM Protect 3.40+ and so on...

- **Android Packers/Obfuscators**: DexGuard, DexProtect, DevGuard, Arxan, ApkGyard, and so on...
INTRODUCTION

- .NET packers use similar tricks of native code:
  - Control flow obfuscation and dead/junk code insertion.
  - Renaming: methods signatures, fields, methods implementation, namespaces, metadata and external references.
  - Re-encoding: changing printable to unprintable characters
  - Simple encryption of methods and strings.
  - Cross reference obfuscation.

- To native code, there’re well known memory APIs: `VirtualAlloc/Ex()`, `HeapCreate() / RtlCreateHeap()`, `HeapReAlloc()`, `GlobalAlloc()`, `RtlAllocateHeap()`
INTRODUCTION

- Most packed binaries can be unpacked using debuggers, breakpoints and dumping unpacked content from memory.

- Even when a binary uses **customized packing techniques**, it is still possible:
  - dumping the unpacked code from memory using Volatility.
  - fixing the ImageAddress field using few lines in Python its respective IAT using `impscan plugin` to analyze it in IDA Pro:
    - `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`
REVERSING
➢ INTRODUCTION

❖ As we’ve mentioned previously, strings are one of first reference.
❖ However, they are all encrypted and writing YARA rules using them could not be so interesting 😊
Setup an exception framework

```
push ebp
mov ebp, esp
push 0xFFFFFFFFh
push offset SEH_408B90
mov eax, large fs:0 ; Remember:

; FS --> TEB --> TIB --> SEH
;
; push Exception Handler (0xFFFFFFFF to end of handler list)
; push next record
; mov fs:[0], esp

push eax
sub esp, 110h
mov eax, _security_cookie
xor eax, ebp
mov [ebp+var_14], eax
push ebx
push esi
push edi
push eax
lea eax, [ebp+var_C]
mov large fs:0, eax
mov [ebp+var_10], esp
push 256 ; Size
lea eax, [ebp+Dst]
push 0 ; Val
push eax ; Dst
call memset ; void *memset(void *dest, int c, size_t count);
; Sets buffers to a specified character.
add esp, 0Ch
```

DLLs seem to be “obfuscated” 😊

HTTP://WWW.BLACKSTORMSECURITY.COM
Old anti-debugger tricks... 😊
Dynamic DLL name resolution being executed before resolving the function address.
INTRODUCTION

Jump table

DLL name (obfuscated format)
INTRODUCTION

```assembly
loc_4053E1:
  cmp    eax, 4
  jb     short loc_4053F9
  cmp    [eax+4DAF44h], dl
  jnz    short loc_4053F5
  mov    ecx, 1
  jmp    short loc_4053F9

loc_4053F5:
  cmp    ecx, edx
  jz     short loc_4053FF

loc_4053F9:
  ; CODE XREF: ab_decode_dll_names+6C↑j
  mov    byte ptr var_1[eax], dl

loc_4053FF:
  ; CODE XREF: ab_decode_dll_names+77↑j
  inc    eax
  cmp    eax, 14h
  jb     short loc_4053E1

loc_405405:
  ; CODE XREF: ab_decode_dll_names+F↑j
  mov    eax, 4DAF48h
  retn
```
Each offset takes us to a different switch case, which is a different DLL name resolution function.
➢ As expected, there’re many calls to the same function for “decrypting” the DLL names 😊
INTRODUCTION

```python
from binascii import *

var_1 = ['50', '73', 'EA', '50']
var_2 = ['1B', '7E', '0C', '62']
var_3 = ['07', '6B', '58', '6A']
var_4 = ['44', '20', '4C', '20']
var_5 = ['20', '19', 'E2', '17']

mylist = var_1 + var_2 + var_3 + var_4 + var_5

def mydecrypt(hexdata):
    max = len(hexdata) - 1
    counter = max
    output = ''

    while(True):
        hexdata[counter] = ord(unhexlify(hexdata[counter])) ^ ord(unhexlify(hexdata[counter - 1]))
        counter -= 1
        if counter == 0:
            break
    return hexdata

final = mydecrypt(mylist)
for x in range(0,4):
    final[x] = 0
final1 = ''.join([chr(w) for w in final])
print("The output is %s" % final1)
```

> output: kernel32.dll
Of course, we could try to improve and automatize the de-obfuscation of all functions names by using:

- **IDA Python**: using IDA Python you can de-obfuscated function names and save them into the idb.
- **IDC**: it’s a bit more complicated, but very powerful.

If your time is short, so you could try emulation tools such as Floss ([https://github.com/fireeye/flare-floss](https://github.com/fireeye/flare-floss)) to decode possible obfuscated strings and create an IDA script to decorate the reversed code:

- `floss --no-static-strings -x malware.bin --ida=floss_ida.py`
### INTRODUCTION

```
E:\malware_samples>floss --no-static-strings -x malware.bin --ida=floss_ida.py
...
```

Decoding function at 0x405380 (decoded 38 strings)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Called At</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4DAF48</td>
<td>0x402D6F</td>
<td>Kernel32.dll</td>
</tr>
<tr>
<td>0x4DAF5C</td>
<td>0x402D8D</td>
<td>CloseHandle</td>
</tr>
<tr>
<td>0x4DAF70</td>
<td>0x402DAE</td>
<td>CreateFileA</td>
</tr>
<tr>
<td>0x4DAF84</td>
<td>0x402DC9</td>
<td>CreateMutexA</td>
</tr>
<tr>
<td>0x4DAFAC</td>
<td>0x402DE4</td>
<td>CreateToolhelp32Snapshot</td>
</tr>
<tr>
<td>0x4DAFCC</td>
<td>0x402DFF</td>
<td>DeviceIoControl</td>
</tr>
<tr>
<td>0x4DAFE4</td>
<td>0x402E1A</td>
<td>GetCurrentThread</td>
</tr>
<tr>
<td>0x4DAFFC</td>
<td>0x402E35</td>
<td>GetLongPathNameA</td>
</tr>
<tr>
<td>0x4DB014</td>
<td>0x402E50</td>
<td>GetModuleFileNameA</td>
</tr>
<tr>
<td>0x4DB030</td>
<td>0x402E6B</td>
<td>GetNativeSystemInfo</td>
</tr>
<tr>
<td>0x4DB04C</td>
<td>0x402E86</td>
<td>GetProcessHeap</td>
</tr>
<tr>
<td>0x4DB064</td>
<td>0x402E1A</td>
<td>GetSystemInfo</td>
</tr>
<tr>
<td>0x4DB07C</td>
<td>0x402EBC</td>
<td>GetThreadContext</td>
</tr>
<tr>
<td>0x4DB094</td>
<td>0x402ED7</td>
<td>HeapAlloc</td>
</tr>
<tr>
<td>0x4DB0A8</td>
<td>0x402EF2</td>
<td>HeapFree</td>
</tr>
<tr>
<td>0x4DAF98</td>
<td>0x402F0D</td>
<td>HeapReAlloc</td>
</tr>
<tr>
<td>0x4DB0B8</td>
<td>0x402F28</td>
<td>IsBadReadPtr</td>
</tr>
<tr>
<td>0x4DB0CC</td>
<td>0x402F43</td>
<td>Module32First</td>
</tr>
<tr>
<td>0x4DB0E4</td>
<td>0x402F5E</td>
<td>Module32Next</td>
</tr>
</tbody>
</table>

HTTP://WWW.BLACKSTORMSECURITY.COM
Same result, of course. 😊
INTRODUCTION

- IDA Python is ALWAYS very handy and we can use IDA Pro SDK to write plugins for:
  - extending the IDA Pro functionalities
  - analyzing piece of obsfuscated code and data flow
  - automatizing unpacking of strange malicious files
  - decoding and loading encrypted / modified MBRs

- 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 2017/2019 (File ➔ New ➔ Create Project ➔ Visual C++ ➔ Windows Desktop ➔ Dynamic-Link Library (DLL)).

Change few project properties as shown in this slide and next ones.
**INTRODUCTION**

- Include the “\_NT\_;\_IDP\_” in Processor Definitions and change Runtime Library to “Multi-threaded” (MT).

---

**Warning!**
It is NOT /MTd
ida.lib: C:\Program Files (x86)\IDA 6.95\idasdk695\lib\x86_win_vc_32
Needed headers. 😊

```cpp
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <allins.hpp>
#include <strlist.hpp>
#include <search.hpp>
```

Initialization function.

```cpp
int IDAP_init() {
    return PLUGIN_KEEP;
}
```

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

Clean-up tasks.

```cpp
void IDAP_term(void) {
}
```

This is called when users activates the plugin.

```cpp
void IDAP_run(int arg) {
    msg("Blackstorm Security: basic plugin example :)\n\n");
    char blackstorm[MAXSTR];
    string_info_t strinfo;
    char s[] = "[a-zA-Z0-9_]\+[.]\{1,\}([a-zA-Z0-9_]\+[.]\{1,\}[a-zA-Z0-9]\{2,\}"
    auto last = BADADDR;
    auto ea = 0;
    auto urlcount = 1;
}"
```

Simple (and incomplete) URL regex. 😊
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 structure.

Plugin will be activated by combination ALT-C. 😊
URLs found within this malicious driver. 😊

ALT + C
ANTI-REVERSING
Advanced threats don’t use standard tricks and there’re many lots of facts about them:

- They use similar techniques from modern packers such as Themida, Arxan, Agile .NET, Tigress, Obfuscator-LLVM and so on.
- Most of them are focused on 64-bit code.
- Obviously, almost all functions are removed from IAT. Remember that Themida packer usually keeps only TlsSetValue().
- String encryption is also a common technique.
ANTI-REVERSING

- Advanced threats are concerned to integrity:
  - 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 completely decoded in the memory*.
  - There could be *checksum functions* verifing the integrity of the code itself. Therefore, any attempt of changing the code may break it up.
  - Additionally, advanced threats may use *watermark* to control the “ownership”. This is the same technique used to program copyright.
Many additional techniques are used:

- There are also fake push instructions.
- There are many dead and useless pieces of code.
- There is some code reordering using unconditional jumps.
- All obfuscators use code flattening.
- Packers have few anti-debugger and anti-vm tricks. Weird anti-vm methods based on temperature, for example.
ANTI-REVERSING

RVA → RVA + process base address and other tasks.

Instructions are stored in an encrypted format.

A, B, C, ... are handlers such as handler_add, handler_sub, handler_push...

Opcodes from a custom instruction set.

Instructions are stored in an encrypted format.
- opcode 1
- opcode 2
- opcode 3
- opcode 4
- opcode 5
- opcode 6
- opcode 7

- handler 1
- handler 2
- handler 3
- handler 4
- handler 5
- handler 6
- handler 7

function pointer table (likely encrypted)
ANTI-REVERSING

Remember: obfuscating is transforming a code from A to B by using different techniques (including virtualization).

What’re the transition points from native code to virtualized code and vice-versa?

Prologues and epilogues from each function could not be 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). 😞
ANTI-REVERSING

- In few cases the VM handlers come from data blocks.

- Original code section could be “splitted” and “scattered” around the program. In this case, 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.
ANTI-REVERSING

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

- API names could be hashed (as used in shellcodes). 😊

- Custom packers usually don’t virtualize all native (x86/x64) instructions.

- There’s a mix between virtualized, native instructions and data after the packing procedure.
ANTI-REVERSING

- Native APIs could be redirected to stub code (proxy), 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.
ANTI-REVERSING

- By the way, how many virtualized instructions exist in the binary?

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

- Try to classify virtualized instructions in groups according to operands and their purpose such as memory access, conditional/unconditional jumps, arithmetic, general...

- Try to understand the size of virtualized instructions which we might fit into a structure that represents encryption key, data, RVA (location), opcode (type) and so on.
ANTI-REVERSING

- Pay attention to instruction’s stem to put similar classes of instructions together as, for example, jump instructions, direct calls, indirect calls and so on.

- Find the transition instructions from native mode to virtualized mode and vice versa.

- Find similarity between virtualized instructions and x86 instructions.

- x86 instructions are also kept encrypted and compressed together with the virtualized instructions.
ANTI-REVERSING

- **Constant unfolding:** technique used by obfuscators to replace a constant by a bunch of code that produces the same resulting constant’s value.

- **Pattern-based obfuscation:** exchange of one instruction by a set of equivalent instructions.

- **Abusing inline functions.**

- **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.
ANTI-REVERSING

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

- **Control indirection 1**: call instruction $\rightarrow$ stack pointer update $\rightarrow$ return skipping some junk code after the call instruction (RET x).

- **Control indirection 2**: malware trigger an exception $\rightarrow$ registered exception is called $\rightarrow$ new branch of instructions.

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

- **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.

- **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;
}
```
; 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

; __unwind {
push rbp
mov rbp, rsp
sub rsp, 10h
mov [rbp+var_4], 0
jmp short loc_1160

loc_1160:
cmp [rbp+var_4], 1Dh
jle short loc_1146

loc_1146:
omv eax, [rbp+var_4]
omv esi, eax
lea rdi, format ; "%d\n"
omv eax, 0
call _printf
add [rbp+var_4], 1
mov eax, 0
leave
ret
; } // starts at 1135
main endp
ANTI-REVERSING

Disadvantages:
✓ Loss of performance
✓ Easy to identify the Code flattening

cc = 1
cc != 0
switch(cc)
cc = 1
cc = 2
cc = 3

aborges = 0
cc = 2
break
aborges < 30
cc = 0
cc = 3
break

printf
aborges++
cc = 2
break

loading libs
c

aborges = 0
cc = 2
break
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`
ANTI-REVERSING

- 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

- Where:
  - fla: Control Flow Flattening
  - sub: Instruction Substitution
  - bcf: Opaque Predicate
ANTI-REVERSING

- A better option would be using Tigress.
- Download Tigress binary from https://tigress.wtf/download.html
- Install Tigress is pretty easy:
  - unzip tigress-3.1-bin.zip
  - Export the TIGRESS_HOME environment variable:
    - export TIGRESS_HOME=/root/Downloads/tigress/3.1
  - Add the Tigress installation directory to the PATH variable:
    - export PATH=$PATH:/root/Downloads/tigress/3.1
#include <stdio.h>
#include "/root/Downloads/tigress/3.1/tigress.h"

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

    return 0;
}
ANTI-REVERSING

- To transform a C source using Tigress:
  
  - `tigress --Environment=x86_64:Linux:Gcc:4.6 --Transform=Flatten --Functions=main --out=aleborges_obfuscated.c aleborges_trigess.c`

- There’re many notes about the command above:
  
  - We should **pick up one or more functions to be transformed**. Of course, I’ve chosen the `main()` only for educational purposes.

  - The argument for Environment must be according to your environment (`x86_64:Linux:Gcc:4.6`, `x86_64:Daruwin:Clang:5.1`, `armv7:Linux:Gcc:4.6`, `armv8:Linux:Gcc:4.6`)
ANTI-REVERSING

Additional notes:

- We could use **multiple transformations** (specifying the --Transform option and --Function option multiple times) such as:
  - Opaque Predicate (InitOpaque)
  - Virtualization Obfuscation (Virtualize)
  - Split and Merge
  - Encode Literals
  - Encode Branches
  - AntiTaintAnalysis
  - ...and much more...
```c
int main(int _formal_argc , char **_formal_argv , char **_formal_envp )
{
    int aborges ;
    int _BARRIER_0 ;
    unsigned long _l_main_next ;

    {
        megaInit();
        _global_argc = _formal_argc;
        _global_argv = _formal_argv;
        _global_envp = _formal_envp;
        _BARRIER_0 = 1;
        _l_main_next = 2UL;
    }
    while (1) {
        switch (_l_main_next) {
            case 4: ;
            return (0);
            break;
            case 3:
            printf((char const /*__restrict */)%d\n", aborges);
        }
```
ANTI-REVERSING

```c
aborges ++;
{
    _main_next = 0UL;
}
break;
case 0: 
if (aborges < 30) {
    {
        _main_next = 3UL;
    }
} else {
    {
        _main_next = 4UL;
    }
} break;
case 2: 
aborges = 0;
{
    _main_next = 0UL;
} break;
}
}

void megaInit(void)
{
```

Program obfuscated with Tigress (remaining part)
ANTI-REVERSING

Prologue and initial assignment

Main dispatcher
ANTI-REVERSING
Simple opaque predicate and anti-disassembly technique

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

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

.loc_40100D:  ; CODE XREF: .text:00401005j
  jmp near ptr 0D0A8837h
```
ANTI-REVERSING

Call stack manipulation:

- Do you know what’s happening here? 😊
ANTI-REVERSING

If it’s running on Windows 8, so die immediately. If it’s running on Window 7, so there’s a chance with Exception Handlers below. 😊

push 17h ; ProcessorFeature
call IsProcessorFeaturePresent

test eax, eax
jz short loc_53CE01
push 5
pop ecx

int 29h ; Win8: RtlFailFast(ecx)

loc_53CE01:

push esi
push 1
mov esi, 0C0000417h
push esi
push 2

The anti-debugging techniques are within this function, which set up exception handlers. The easiest one is IsDebuggerPresent( ), of course.

call sub_53CC17

add esp, 0Ch
push esi ; uExitCode
call ds:GetCurrentProcess
call ds:TerminateProcess

We usually avoid using ExitProcess( ) or TerminateProcess( ) because possible leakage, but malware’s authors don’t have this concern. 😊

Few lines of code could have several anti-debugging tricks. Maybe some non-recommended functions too 😊
ANTI-REVERSING

```
.push edi
mov edi, [ebp+lpLibFileName]  ; If GetModuleHandleW() fails, so jump to
.test eax, eax
jnz short loc_40C1D1
push offset aKernel32_dll_0 ; "kernel32.dll"
call ds:GetModuleHandleW
.test eax, eax
jz short ab_wrapper_GetSystemDirectoryW
.push offset aSetdefaultdllid ; "SetDefaultDllDirectories"
push eax                ; hModule_GetProcAddress
call ds:GetProcAddress
mov esi, eax
.push esi                ; Ptr
.call ds:EncodePointer
mov Ptr, eax
jmp short loc_40C1DA

loc_40C1D1:
.push eax
.call ds:DecodePointer
mov esi, eax

loc_40C1DA:
.test esi, esi
jz short ab_wrapper_GetSystemDirectoryW
.push 800h ; dwFlags
push 0     ; hFile
push edi   ; lpLibFileName
call ab_wrapper_LoadLibraryExW
jmp short loc_40C251

; CODE XREF: sub_40C185+1F↑
void *enc_func = EncodePointer(SetDefaultDllDirectories);
int (*pointer_fn)(int) = (int (*)(int))DecodePointer(enc_func);
pointer_fn("any_argument");
```

Specifies a default set of directories to search when the calling process loads a
DLL. Is it the malware concerned to a DLL
pre-loading attack? 😊

Malware is protecting their function
pointers of being hooked. Ironic. 😊

Windows7 only has this function if the KB2533623 is
applied, so GetSystemDirectoryW() is a “fallback”.

Looking for DLLs in C:\Windows\system32

Protected by an EH from C++ (uses CPPEH_RECORD struct)
This declaration has been defined by using “Y” in IDA Pro. The ServiceMain() (an exported function, in this case) is responsible for registering a service control handler and updating the Service Control Manager about its status. Therefore, the ServiceMain() is only a placeholder to the local service to be called, which will be started by the SCM in a separate thread.

Each service registers a control handler function that will be invoked by the SCM when control requests are received from the service.

This function is a wrapper to the SetServiceStatus(), which tells to the SCM of the service’s status. Additionally, the _SERVICE_STATUS structure, which hold the type of service (SERVICE_WIN32_OWN_PROCESS – 0x10, in this case) is set up within this wrapper, before calling the SetServiceStatus().

This function is a wrapper to CreateEventA() and WaitForSingleObject(). Events can be signaled or non-signaled. In this case, it’s set to non-signaled, so thread(s) will wait for it got signaled (the operation has completed). Events can be auto-reset (only one of threads waiting for this event becomes schedulable) or manual-reset (all threads waiting for this event becomes schedulable). Our event here is auto-reset and returns to non-signaled state automatically. At end, the thread (_beginthreadex()) calls the wrapper (sub_403990), which calls the WaitForSingleObject() and runs until the service being stopped.
Are there only these anti-forensic techniques? No.... ☺

- Instruction substitution
- Garbage insertion
- Self-modifying code (like shellcode)
- Method renaming
- Cryptography
- Variable reordering
- Instruction reordering
- Randomized Stack frame
- Inheritance manipulation
- And more...much more ☺
DEOBFUSCATION
METASM
How to reverse the obfuscation and, from stage 4, to return to the stage 1?
DEOBFUSCATION

- 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
DEOBFUSCATION

- root@kali:~/github# git clone https://github.com/jjyg/metasm.git

- Include the following line into `.bashrc` file to indicate the Metasm directory installation:
  - `export RUBYLIB=$RUBYLIB:~/github/metasm`

- Test metasm:
  - `ruby -r metasm -e 'p Metasm::VERSION'`

- You should see a number in the output. It’s done 😊
**DEOBFUSCATION**

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

mycode = Metasm::Shellcode.assemble(Metasm::Ta32.new, <<E0B)

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
E0B
```

- Based on metasm.rb file and Bruce Dang’s code.
- This instruction was inserted to make the eax register evaluation easier. 😊
#### DEOBFUSCATION

```ruby
addrstart = 0
asmcode = mycode.init_disassembler
asmcode.disassemble(addrstart)
conference_di = asmcode.di_at(addrstart)
conference = conference_di.block
puts "\n<!!!> Blackstorm Security 2020:\n"
puts conference.list
conference.list.each { |aborges|
  puts "\n<!!!> #{aborges.instruction}"
  back = aborges.backtrace_binding()
  v = back.values
  k = back.keys
  j = k.zip(v)
  puts "Our data flow follows below:\n"
  j.each do |mykeys, myvalues|
    puts "Processing: #{mykeys} ==> #{myvalues}"
  end
  if aborges.opcode.props[:setup]
    puts "\nOur control flow follows below:\n"
    puts "### #{asmcode.get_xrefs_x(aborges)}"
  end
}
```

- Initialize and disassemble code since beginning (start).
- List the assemble code.
- Initialize the backtracking engine.
- Determines which is the final instruction to walk back from there.
DEOBFUSCATION

- Backtracing from the last instruction
- Logs the sequence of backtraced instructions.
- Shows only the effective instructions, which really can alter the final result.
DEOBFUSCATION

Remember: this is our obfuscated code. 😊
DEOBFUSCATION

```c
<!!!> push ebx
Our data flow follows below:
Processing: esp ➞ esp-4
Processing: dword ptr [esp] ➞ ebx

<!!!> mov ebx, 0b9h
Our data flow follows below:
Processing: ebx ➞ 0b9h

<!!!> sub eax, ebx
Our data flow follows below:
Processing: eax ➞ eax-ebx
Processing: eflag_z ➞ (((eax&$fffffffh)-(ebx&$fffffffh))&$fffffffh)==0
Processing: eflag_s ➞ (((eax&$fffffffh)-(ebx&$fffffffh))&$fffffffh)>>1fh)≠0
Processing: eflag_c ➞ (eax&$fffffffh)<(ebx&$fffffffh)
Processing: eflag_o ➞ (((eax&$fffffffh)>>1fh)≠0)==(!(((ebx&$fffffffh)>>1fh)≠0))&((eax&$fffffffh)>>1fh)≠0)≠(((eax&$fffffffh)-(ebx&$fffffffh))&$fffffffh)>>1fh)≠0)

<!!!> pop ebx
Our data flow follows below:
Processing: esp ➞ esp+4
Processing: ebx ➞ dword ptr [esp]

<!!!> sub eax, 55h
Our data flow follows below:
Processing: eax ➞ eax-55h
```

HTTP://WWW.BLACKSTORMSECURITY.COM

ALexandre Borges – Malware and Security Researcher
Our control flow follows below:

```plaintext
>>> [Expression[:eax]]

[+] final output: eax

[Start] Here is the sequence of expression evaluations eax from $0 \times 29$

[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
```
DEOBFUSCATION

The effective instructions are:

1. `mov ebx, 0b9h`
2. `6 sub eax, ebx`
3. `9 sub eax, 55h`
4. `0ch sub eax, 32h`
5. `0fh add eax, ecx`
6. `11h add eax, 50h`
7. `14h add eax, 37h`
8. `19h mov ecx, 49h`
9. `1eh mov edx, ecx`
10. `21h inc edx`
11. `22h add edx, 70h`
12. `25h dec edx`
13. `26h add eax, edx`

- Output originated from backtracing_log.select command (in reverse)

Game over 😊
MIASM
DEOBFUSCATION

- 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 (python 2.7.x):
  - `git clone https://github.com/serpilliere/elfesteem.git elfesteem`
  - `cd elfesteem/`
DEOBFUSCATION

- 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
DEOBFUSCATION

- pip install llvmlite
- pip install future
- apt-get install z3
- apt-get install python-pycparser
- pip install pyparsing
- cd ..
- git clone https://github.com/cea-sec/miasm.git
- cd miasm
- python setup.py build
- python setup.py install
- cd test/
- python test_all.py
DEOBFUSCATION

root@kali:~/github/miasm/test# python2.7 test_all.py

[LLVM] Python'llvm-lite' module is required for llvm tests

[Z3] Z3 and its python binding are necessary for TranslatorZ3.

TEST/ARCH msp430/arch.py
TEST/ARCH ppc32/arch.py
TEST/ARCH x86/arch.py
TEST/ARCH aarch64/arch.py
DONE msp430/arch.py 0s
TEST/ARCH x86/sem.py python
DONE ppc32/arch.py 0s
TEST/ARCH x86/unit/mn_strings.py python
DONE x86/sem.py python 1s
TEST/ARCH x86/unit/mn_stack.py gcc
DONE x86/unit/mn_strings.py python 1s
TEST/ARCH x86/unit/mn_daa.py gcc
DONE x86/unit/mn_daa.py gcc 3s
TEST/ARCH x86/unit/mn_das.py gcc
DONE x86/unit/mn_stack.py gcc 3s
TEST/ARCH x86/unit/mn_int.py gcc
DONE x86/unit/mn_int.py gcc 1s
TEST/ARCH x86/unit/mn_pshufb.py gcc
DONE x86/unit/mn_pshufb.py gcc 1s
TEST/ARCH x86/unit/mn_psrl_psll.py gcc
Before proceeding with MIASM, we need to create a binary containing our code, so we need an assembler and Keystone is great.

- **Keystone Engine** acts as 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`
DEOBFUSCATION

- 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 -2 /root/.bashrc

export RUBYLIB=$RUBYLIB:~/programs/metasm
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
#include <stdio.h>
#include <keystone/keystone.h>

#define ABORGES "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 has occurred while calling ks_open(), quit\n");
        return -1;
    }

    if (ks_asm(keyeng, ABORGES, 0, &encode, &size, &count)) {
        printf("ERROR: A fail has occurred while calling ks_asm() with count = %lu, error code = %s\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;
}
➢ DEOBFUSCATION

```bash
root@kali:/conference# more Makefile

.PHONY: all clean

KEYSTONE_LDFLAGS = -lkeystone -lstdc++ -lm

all:
    ${CC} -o conference2020 conference2020.c ${KEYSTONE_LDFLAGS}

clean:
    rm -rf *.o conference2020

root@kali:/conference# make

root@kali:/conference# cc -o conference2020 conference2020.c -lkeystone -lstdc++ -lm

root@kali:/conference# ./conference2020

root@kali:/conference# ./conference2020 | xxd -r -p - > conference2020.bin

root@kali:/conference# hexdump -C conference2020.bin

00000000  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:/conference#
```

HTTP://WWW.BLACKSTORMSECURITY.COM
DEOBFUSCATION

IDa Pro confirms: it’s our content 😊
DEOBFUSCATION

```python
from colorama import init, Fore, Back, Style
from miasm.analysis.binary import Container
from miasm.analysis.machine import Machine
from miasm.jitter.csts import PAGE_READ, PAGE_WRITE

with open("/root/conference/conference2020.bin") as fdesc:
    cont=Container.from_stream(fdesc)

offset = 0
conferencemach=Machine('x86_32')
conferencedis=conferencemach.dis_engine(cont.bin_stream)
asmcfg = conferencedis.dis_multiblock(offset)
ira = conferencemach.ira(conferencedis.loc_db)

ircfg =ira.new_ircfg_from_asmcfg(asmcfg)
open('out.dot', 'w').write(ircfg.dot())

from miasm.ir.symbexec import SymbolicExecutionEngine
symb = SymbolicExecutionEngine(ira,conferencemach.mn.regs.regs_init)
symbolic_pc = symb.run_at(ircfg, 0, step=True)
print(Fore.YELLOW + "\nThe final value of EAX register is: %s" %
symb.symbols[conferencemach.mn.regs.EAX])
```

Open the binary file generated through Keystone. The Container provides the byte source to the disasm engine.

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

Initialize and run the Symbolic Execution Engine, setting all registers to an initial value.

Print the final value of EAX.
DEOBFUSCATION

```
root@kali:~# python conference_symbolic.py | more

[WARNING ]: not enough bytes in str
[WARNING ]: cannot disasm at 29
[WARNING ]: not enough bytes in str
[WARNING ]: cannot disasm at 29

Instr PUSH EBX

Assignblk:
ESP = ESP + -0x4
q32[ESP + -0x4] = EBX

----------------------------------------

R12 = R12_init
MM2 = MM2_init
FS = FS_init
XMM8 = XMM8_init
AL = AL_init
float_c3 = float_c3_init
ST(0) = ST(0)_init
EAX = EAX_init
cf = cf_init
R8W = R8W_init
float_st6 = float_st6_init
MM1 = MM1_init
```
DEOBFUSCATION

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

R12 = R12_init
MM2 = MM2_init
FS = FS_init
XMM8 = XMM8_init
AL = AL_init
float_c3 = float_c3_init
ST(0) = ST(0)_init
EAX = EAX_init
cf = cf_init
R8W = R8W_init
float_st6 = float_st6_init
MM1 = MM1_init
pf = pf_init
R15B = R15B_init
XMM10 = XMM10_init
SP = SP_init
zf = zf_init
XMM15 = XMM15_init
BL = BL_init
➢ DEOBFUSCATION

DR3 = DR3_init
R14 = R14_init
XMM11 = XMM11_init
CX = CX_init
R13B = R13B_init
DR2 = DR2_init
RCX = RCX_init
i_d = i_d_init
XMM1 = XMM1_init
RSI = RSI_init
R8 = R8_init
DI = DI_init
RAX = RAX_init
float_st4 = float_st4_init
XMM6 = XMM6_init
R15 = R15_init
MM7 = MM7_init
SPL = SPL_init

\[\text{The final value of EAX register is: } EAX\_init + \text{ ECX\_init} \]

Finally.... 😊

- I’ve skipped a very long output, which shows all instruction being execute. We are interested in the final value of EAX. 😊

- If you want, view the graph:
  - apt-get install graphviz
  - apt-get install xdot
  - xdot out.dot
Closing thoughts and Acknowledgments...

- Honestly, I didn’t even scratch the surface of this topic....

- There’re tons of obfuscation techniques and de-obfuscation/reversing techniques to explain... it would take one or two entire weeks...and probably you wouldn’t to know about it... 😊

- I’d like to thank BHACK for the event and, in special, my friends Ewerson (crashbr) and Rafael. 😊

- And, of course, I’d like to thank you (the audience) for attending my talk.
THANK YOU FOR ATTENDING MY TALK 😊

- Security Researcher
- Speaker at SANS 2020
- Speaker at DEF CON USA 2019
- Speaker at DEF CON USA 2018
- Speaker at DEF CON CHINA 2019
- Speaker at DC2711 (Johannesburg)
- Speaker at NO HAT 2019 (Italy)
- Speaker at HITB 2019 (Amsterdam)
- Speaker at CONFidence 2019 (Poland)
- Speaker at DevOpsDays BH 2019
- Speaker at H2HC 2016/2015
- Speaker at BHACK 2018
- Researcher on Android/iOS Reversing, Rootkits and Digital Forensics.
- Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response

- Twitter: @ale_sp_brazil
  @blackstormsecbr

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

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

- E-mail: alexandreborges@blackstormsecurity.com