Understanding Alcatraz ~ Obfuscator Analysis [EN]

0xreverse.com/understanding-alcatraz-obfuscator-analysis-en

Introduction

Binary-to-binary (bin2bin) obfuscators, which are frequently preferred by both malware developers and users seeking to protect their source code, are making the work of malware analysts and reverse engineering experts increasingly difficult with their advanced techniques, complicating the overall analysis process. In this article, the processes of one such tool, the “Alcatraz” obfuscator, are analyzed.

Alcatraz Obfuscator

Alcatraz is an obfuscator capable of obfuscating x64-based PE files, and it stands out among open-source binary-to-binary obfuscator solutions with the techniques it incorporates.

- Obfuscation of immediate moves
- Control flow flattening
- ADD mutation
- Entry-point obfuscation
- Lea obfuscation
- Anti disassembly

If we are not analyzing directly on a sample (such as a crackme, malware, etc.), we can more easily observe the changes made by the target obfuscator using a file we have prepared ourselves. For this purpose, I created a test application prior to the analysis.

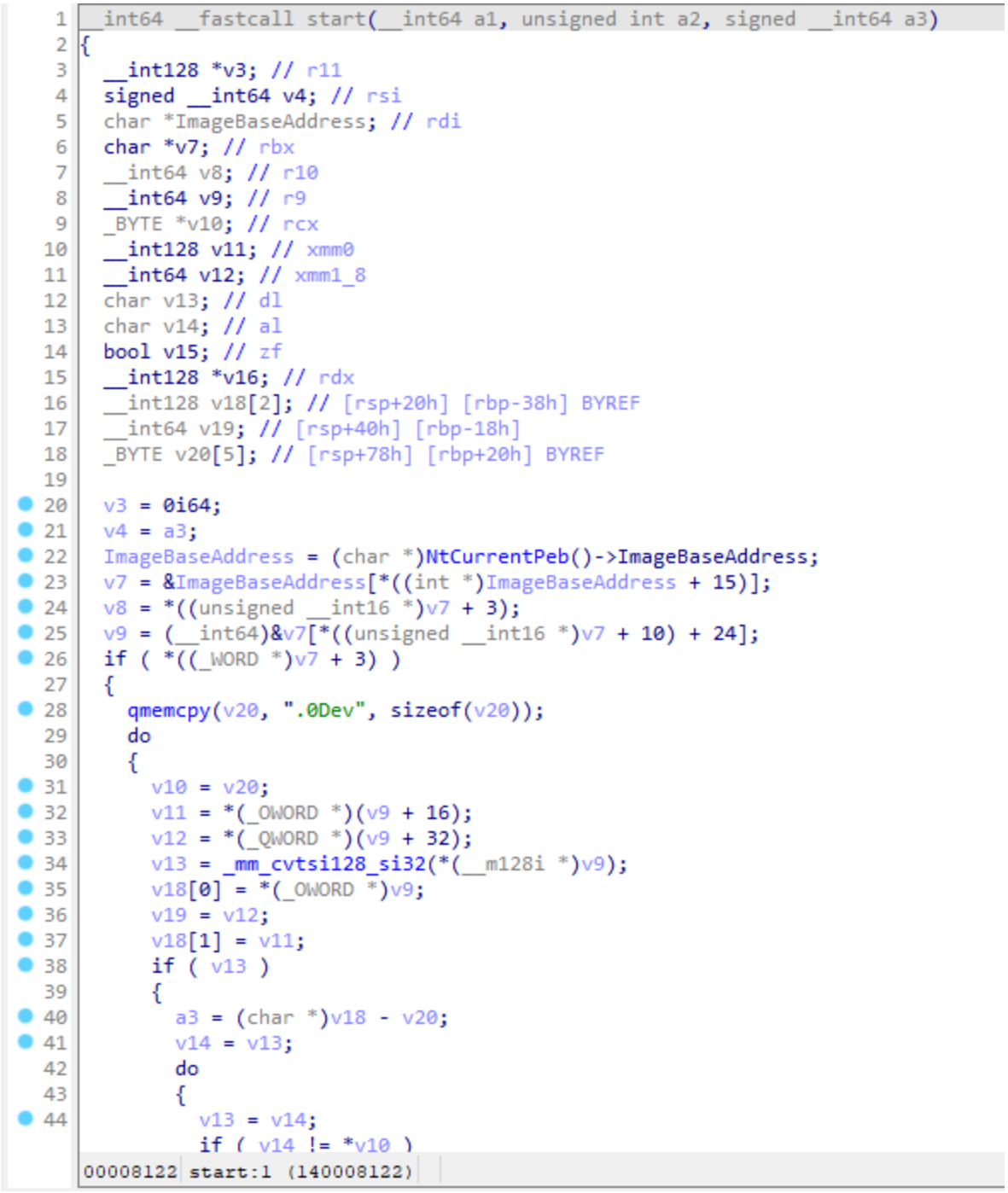
```
#include <stdio.h>

char* func1() {
    return "0xreverse.com";
}

int func2() {
    return 24;
}

int main()
{
    printf("%s\n", func1());
    printf("%d\n", func2());
    getchar();
    return 1;
}
```

I obfuscated this application, which I compiled in Release (x64) mode, using Alcatraz with all features enabled. In the remainder of the article, I will refer to this test application as the “sample.”

Analysis

Entry-Point Obfuscation

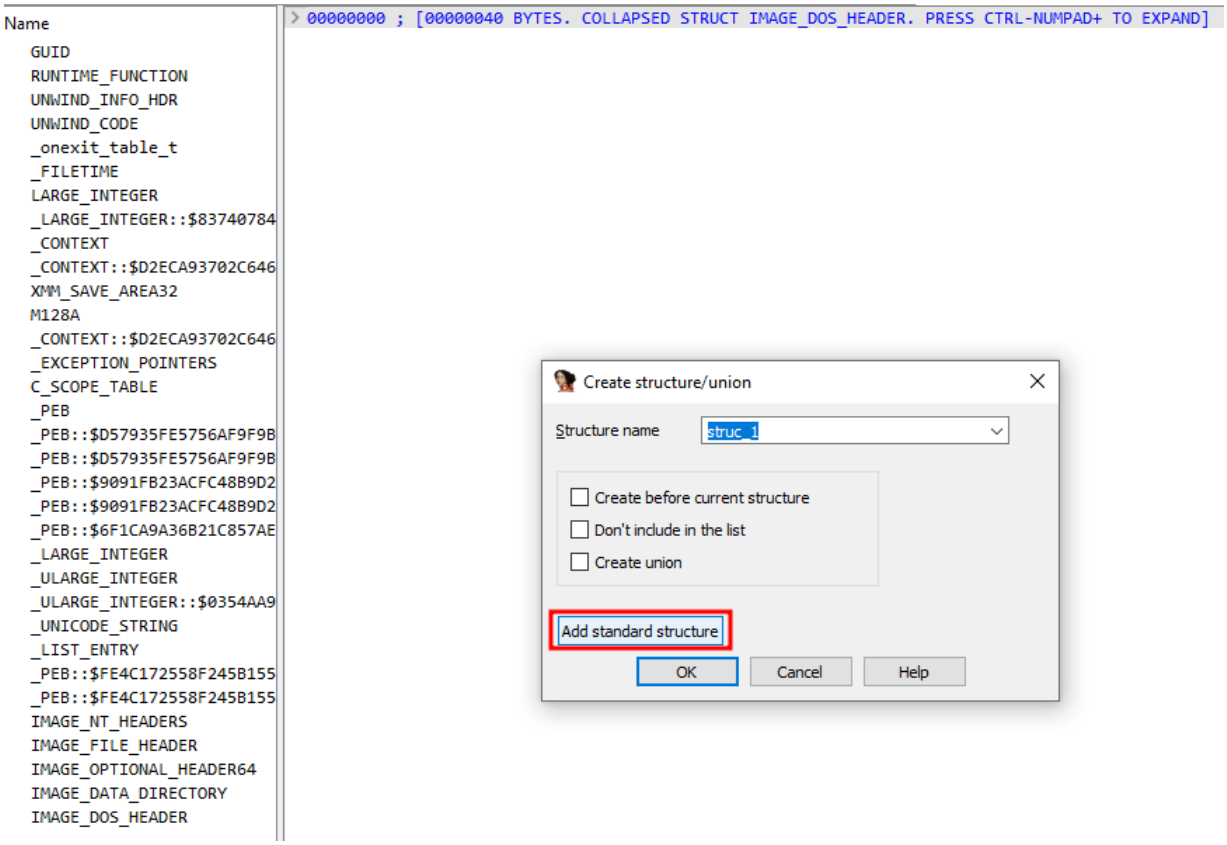
When I opened the sample on IDA, I saw that the first function (Entry-Point) was doing something with PE Headers.



```
__int64 __fastcall start(__int64 a1, unsigned int a2, signed __int64 a3)
{
  __int128 *v3; // r11
  signed __int64 v4; // rsi
  char *ImageBaseAddress; // rdi
  char *v7; // rbx
  __int64 v8; // r10
  __int64 v9; // r9
  _BYTE *v10; // rcx
  __int128 v11; // xmm0
  __int64 v12; // xmm1_8
  char v13; // dl
  char v14; // al
  bool v15; // zf
  __int128 *v16; // rdx
  __int128 v18[2]; // [rsp+20h] [rbp-38h] BYREF
  __int64 v19; // [rsp+40h] [rbp-18h]
  _BYTE v20[5]; // [rsp+78h] [rbp+20h] BYREF

  v3 = 0i64;
  v4 = a3;
  ImageBaseAddress = (char *)NtCurrentPeb()->ImageBaseAddress;
  v7 = &ImageBaseAddress[*((int *)ImageBaseAddress + 15)];
  v8 = *((unsigned __int16 *)v7 + 3);
  v9 = (__int64)&v7[*((unsigned __int16 *)v7 + 10) + 24];
  if ( *((_WORD *)v7 + 3) )
  {
    qmemcpy(v20, ".0Dev", sizeof(v20));
    do
    {
      v10 = v20;
      v11 = *(_OWORD *)(v9 + 16);
      v12 = *(_QWORD *)(v9 + 32);
      v13 = _mm_cvtsi128_si32(*(__m128i *)v9);
      v18[0] = *(_OWORD *)v9;
      v19 = v12;
      v18[1] = v11;
      if ( v13 )
      {
        a3 = (char *)v18 - v20;
        v14 = v13;
        do
        {
          v13 = v14;
          if ( v14 != *v10 )
00008122 start:1 (140008122)
```

```
ImageBaseAddress = (char *)NtCurrentPeb()->ImageBaseAddress;
v7 = &ImageBaseAddress[*((int *)ImageBaseAddress + 15)];
v8 = *((unsigned __int16 *)v7 + 3);
v9 = (__int64)&v7[*((unsigned __int16 *)v7 + 10) + 24];
```

Since IDA does not recognize the structures, it represents them using pointer operations. To improve this process, I imported the MS SDK (Windows 7 x64) types from the Type Libraries section (Shift + F11 or View > Open Subviews). To modify the types of variables, I added the necessary types as “standard structures” via the Structures section (Shift + F9) by pressing the Ins key. After adding the structures, I converted the relevant variables in the code to appropriate types.




```
dosHeader = (IMAGE_DOS_HEADER *)NtCurrentPeb()->ImageBaseAddress;
ntHeader = (IMAGE_NT_HEADERS *)((char *)dosHeader + dosHeader->e_lfanew);
NumberOfSections = ntHeader->FileHeader.NumberOfSections;
firstSectionHeader = (IMAGE_SECTION_HEADER *)((char *)&ntHeader->OptionalHeader
                                              + ntHeader-
>FileHeader.SizeOfOptionalHeader);
if (ntHeader->FileHeader.NumberOfSections)
{
    qmemcpy(obfuscatorSection, ".0Dev", sizeof(obfuscatorSection));
    do
    {
      v10 = obfuscatorSection;
      v11 = *(_OWORD *)&firstSectionHeader->SizeOfRawData;
      v12 = *(_QWORD *)&firstSectionHeader->NumberOfRelocations;
      hasSectionName = _mm_cvtsi128_si32(*(__m128i *)firstSectionHeader->Name);
      *(_OWORD *)selectedSectionName.Name = *(_OWORD *)firstSectionHeader-
>Name;
      *(_QWORD *)&selectedSectionName.NumberOfRelocations = v12;
      *(_OWORD *)&selectedSectionName.SizeOfRawData = v11;
      if ( hasSectionName )
      {
        a3 = (char *)&selectedSectionName - obfuscatorSection;
        v14 = hasSectionName;
//...
```

After these operations, the start function has become easier to understand. At first, the function tries to find the section whose name is 0Dev. Then it calculates the Original Entry-Point by performing mathematical operations on some values in the header of this section.

```
return ((__int64 (__fastcall *)(_QWORD, signed __int64, signed __int64,
IMAGE_SECTION_HEADER *))((char *)dosHeader + (unsigned
int)__ROR4__(LODWORD(ntHeader->OptionalHeader.SizeOfStackCommit) ^ *(_DWORD *)
((char *)&dosHeader->e_magic + v3->VirtualAddress), ntHeader-
>FileHeader.TimeDateStamp)))(
       a2,
       v4,
       a3,
       firstSectionHeader);
```

```
__int64 __fastcall start(__int64 a1, unsigned int a2, signed __int64 a3)
{
  IMAGE_SECTION_HEADER *v3; // r11
  signed __int64 v4; // rsi
  IMAGE_DOS_HEADER *dosHeader; // rdi
  IMAGE_NT_HEADERS *ntHeader; // rbx
  __int64 NumberOfSections; // r10
  IMAGE_SECTION_HEADER *firstSectionHeader; // r9
  _BYTE *v10; // rcx
  __int128 v11; // xmm0
  __int64 v12; // xmm1_8
  char hasSectionName; // dl
  char v14; // al
  bool v15; // zf
  IMAGE_SECTION_HEADER *p_selectedSectionName; // rdx
  IMAGE_SECTION_HEADER selectedSectionName; // [rsp+20h] [rbp-38h] BYREF
  _BYTE obfuscatorSection[5]; // [rsp+78h] [rbp+20h] BYREF

  v3 = 0i64;
  v4 = a3;
  dosHeader = (IMAGE_DOS_HEADER *)NtCurrentPeb()->ImageBaseAddress;
  ntHeader = (IMAGE_NT_HEADERS *)((char *)dosHeader + dosHeader->e_lfanew);
  NumberOfSections = ntHeader->FileHeader.NumberOfSections;
  firstSectionHeader = (IMAGE_SECTION_HEADER *)((char *)&ntHeader->OptionalHeader
                                                + ntHeader->FileHeader.SizeOfOptionalHeader);
  if ( ntHeader->FileHeader.NumberOfSections )
  {
    qmemcpy(obfuscatorSection, ".0Dev", sizeof(obfuscatorSection));
    do
    {
      v10 = obfuscatorSection;
      v11 = *(_OWORD *)&firstSectionHeader->SizeOfRawData;
      v12 = *(_QWORD *)&firstSectionHeader->NumberOfRelocations;
      hasSectionName = _mm_cvtsi128_si32(*(__m128i *)firstSectionHeader->Name);
      *(_OWORD *)selectedSectionName.Name = *(_OWORD *)firstSectionHeader->Name;
      *(_QWORD *)&selectedSectionName.NumberOfRelocations = v12;
      *(_OWORD *)&selectedSectionName.SizeOfRawData = v11;
      if ( hasSectionName )
      {
        a3 = (char *)&selectedSectionName - obfuscatorSection;
        v14 = hasSectionName;
        do
        {
          hasSectionName = v14;
          if ( v14 != *v10 )
            break;
          v14 = (v10++)[a3 + 1];
          hasSectionName = v14;
        }
        while ( v14 );
      }
      v15 = hasSectionName == *v10;
      p_selectedSectionName = &selectedSectionName;
      if ( !v15 )
        p_selectedSectionName = v3;
      ++firstSectionHeader;
```

We can also verify this process through the source code:

```
__declspec(safebuffers) int obfuscator::custom_main(int argc, char* argv[]) {
    auto peb = (uint64_t)__readgsqword(0x60); // peb
    auto base = *(uint64_t*)(peb + 0x10); // peb + 0x10 = DOS Header (MZ)
    PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)(base +
((PIMAGE_DOS_HEADER)base)->e_lfanew);

    PIMAGE_SECTION_HEADER section = nullptr;
    auto first = IMAGE_FIRST_SECTION(nt);
    for (int i = 0; i < nt->FileHeader.NumberOfSections; i++) {
        auto currsec = first[i];

        char dev[5] = { '.', '0', 'D', 'e', 'v' };
        if (!_strcmp((char*)currsec.Name, dev)) {
            section = &currsec;
        }
    }

    uint32_t real_entry = *(uint32_t*)(base + section->VirtualAddress);
    real_entry ^= nt->OptionalHeader.SizeOfStackCommit;
    real_entry = _rotr(real_entry, nt->FileHeader.TimeDateStamp);
    return reinterpret_cast<int(*)(int, char**)>(base + real_entry)(argc,
argv);
}
```

Writing OEP Finder with Qiling Framework

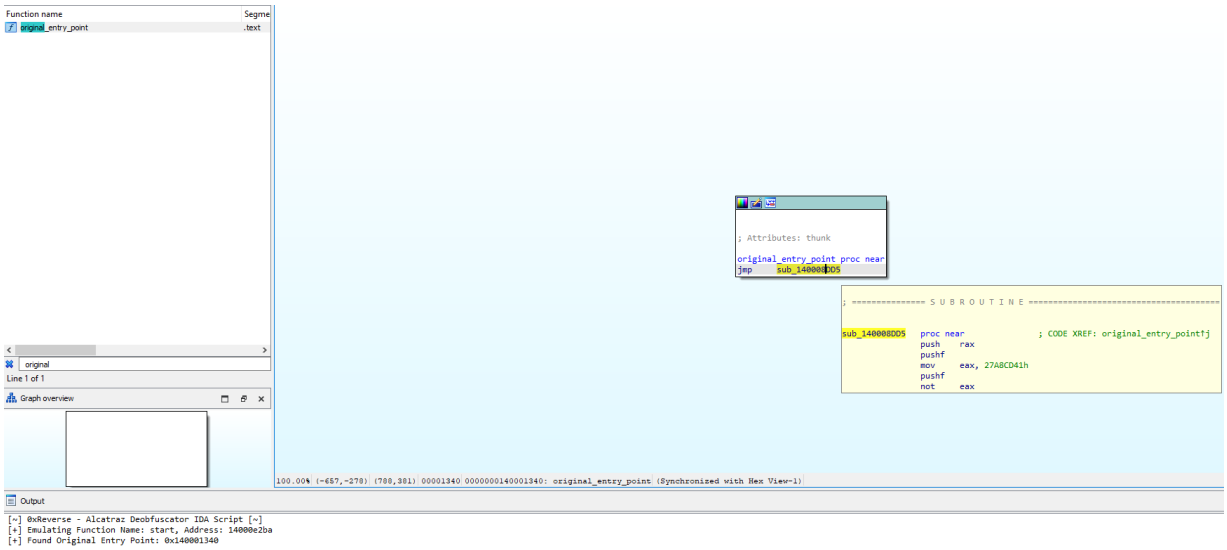
There are many possible approaches to performing all these steps (e.g., calculating them statically on the binary using PE parser libraries like LIEF). However, I would like to demonstrate how this entire process can be handled using the Qiling Framework. By using IDAPython APIs, we can retrieve the entry point of the sample as well as the start and end addresses of the function at that location.

```
from qiling import Qiling
from qiling.const import QL_VERBOSE

# Qiling Version: 1.4.8 f4464b95
# IDA Pro Version 7.7

ROOTFS_PATH = r"ROOTFS_PATH"

def find_oep(sample_path, func) -> int:
    # I added the log_devices parameter because I got the following error.
    # TypeError: unexpected logging device type: IDAPythonStdOut
    ql = Qiling([sample_path], rootfs=ROOTFS_PATH, verbose=QL_VERBOSE.OFF,
log_devices=[])
    # To prevent Qiling from continuing emulation, the RAX value
    # must be taken one instruction before the jmp opcode is executed.
    ql.run(begin=func.start_ea, end=func.end_ea - 0x4) # - jmp opcode size
    original_entry_point = ql.arch.regs.read("RAX")
    print(f"[+] Found Original Entry Point: 0x{original_entry_point:x}")
    return original_entry_point


def main():
    print("[~] 0xReverse - Alcatraz Deobfuscator IDA Script [~]")
    sample_path = idaapi.get_input_file_path()
    # IDAPython CheatSheet
    # https://gist.github.com/icecr4ck/7a7af3277787c794c66965517199fc9c
    info = idaapi.get_inf_structure()
    entrypoint = info.start_ea  # EntryPoint start address
    if func := ida_funcs.get_func(entrypoint):
        function_name = idc.get_func_name(entrypoint)
        print(f"[+] Emulating Function Name: {function_name}, Address:
{entrypoint:x}")
        original_entry_point = find_oep(sample_path, func)
        # Rename OEP address to original_entry_point
        idc.set_name(original_entry_point, "original_entry_point",
idc.SN_NOWARN)
    else:
        print(f"[-] This address is not a function")
        exit(-1)

main()
```

This is the output I got after running this script with `File > Script File... (Alt + F7)`:

```
[~] 0xReverse - Alcatraz Deobfuscator IDA Script [~]
[+] Emulating Function Name: start, Address: 14000e2ba
[+] Found Original Entry Point: 0x140001340
```

Since we renamed the function at the target address to `original_entry_point`, instead of using the "Jump to Address (G)" option, we can directly search for the function name in the Functions section to view the actual entry-point location.



LEA Obfuscation

```
lea rdx, cs:7FF7D996FE33h
pushf
sub rdx, 421FDBD3h
popf
lea rcx, cs:7FF7FD145F19h
pushf
sub rcx, 659D3CA9h
popf
call loc_7FF797778E4A
```

The Load Effective Address (LEA) instruction loads the target address into the specified register. The sub instruction, which is wrapped with pushf and popf, reveals the actual address by subtracting a specific value from the address stored in the rdx register.

> The reason for wrapping with pushf and popf is to ensure that the RFLAGS on the CPU are not affected by the sub operation. With pushf, the RFLAGS are saved to memory, and then with popf, the preserved RFLAGS are restored.

Hence the representation of the above process:

$$RDX = 0x7FF7D996FE33 - 0x421FDBD3$$

$$RCX = 0x7FF7FD145F19 - 0x659D3CA9$$

You can see how it calculates this in the lea.cpp file in the Alcatraz Codebase:

```
auto rand_add_val = distribution(generator);
instruction->location_of_data += rand_add_val;
assm.pushf();
assm.sub(x86_register_map->second, rand_add_val);
assm.popf();
void* fn;
auto err = rt.add(&fn, &code);
```

Display on the debugger:

```
RAX 00007FFA5A3107A8 ↳ ucrtbase.dll:00007FFA5A3107A8
RBX 000002C5039E72D0 ↳ debug028:000002C5039E72D0
RCX 00007FF797772270 ↳ "%s\n"
RDX 00007FF797772260 ↳ "0xreverse.com"
RSI 0000000000000000 ↳
RDI 000002C5039EC5D0 ↳ debug028:000002C5039EC5D0
RBP 0000000000000000 ↳
```

Anti-Disassembly

According to the description here, it baffles **linear disassemblers** that it converts an instruction starting with `0xFF` into `0xEB 0xFF`. Since I plan to explain this technique (Impossible Disassembly) and other anti-* techniques in a different blog post, I won't go into detail here. We can clearly see how it does this from the source code.

```
bool obfuscator::obfuscate_ff(std::vector<obfuscator::function_t>::iterator&
function, std::vector<obfuscator::instruction_t>::iterator& instruction) {

    instruction_t conditional_jmp{}; conditional_jmp.load(function->func_id, {
0xEB });
    conditional_jmp.isjmpcall = false;
    conditional_jmp.has_relative = false;
    instruction = function->instructions.insert(instruction, conditional_jmp);
    instruction++;

    return true;
}
```

```
.0Dev:00007FF797778417 call    loc_7FF797778E4A
.0Dev:00007FF79777841C
.0Dev:00007FF79777841C loc_7FF79777841C:                       ; CODE XREF: .0Dev:loc_7FF79777841C↑j
.0Dev:00007FF79777841C jmp     short near ptr loc_7FF79777841C+1
.0Dev:00007FF79777841E ; ---------------------------------------------------------------------------
.0Dev:00007FF79777841E adc     eax, 0FFFF9D55h
.0Dev:00007FF797778423 mov     eax, 0F74D200Dh
```

MOV & ADD Obfuscation

As I mentioned in LEA obfuscation, it does some operations wrapped in pushf and popf.

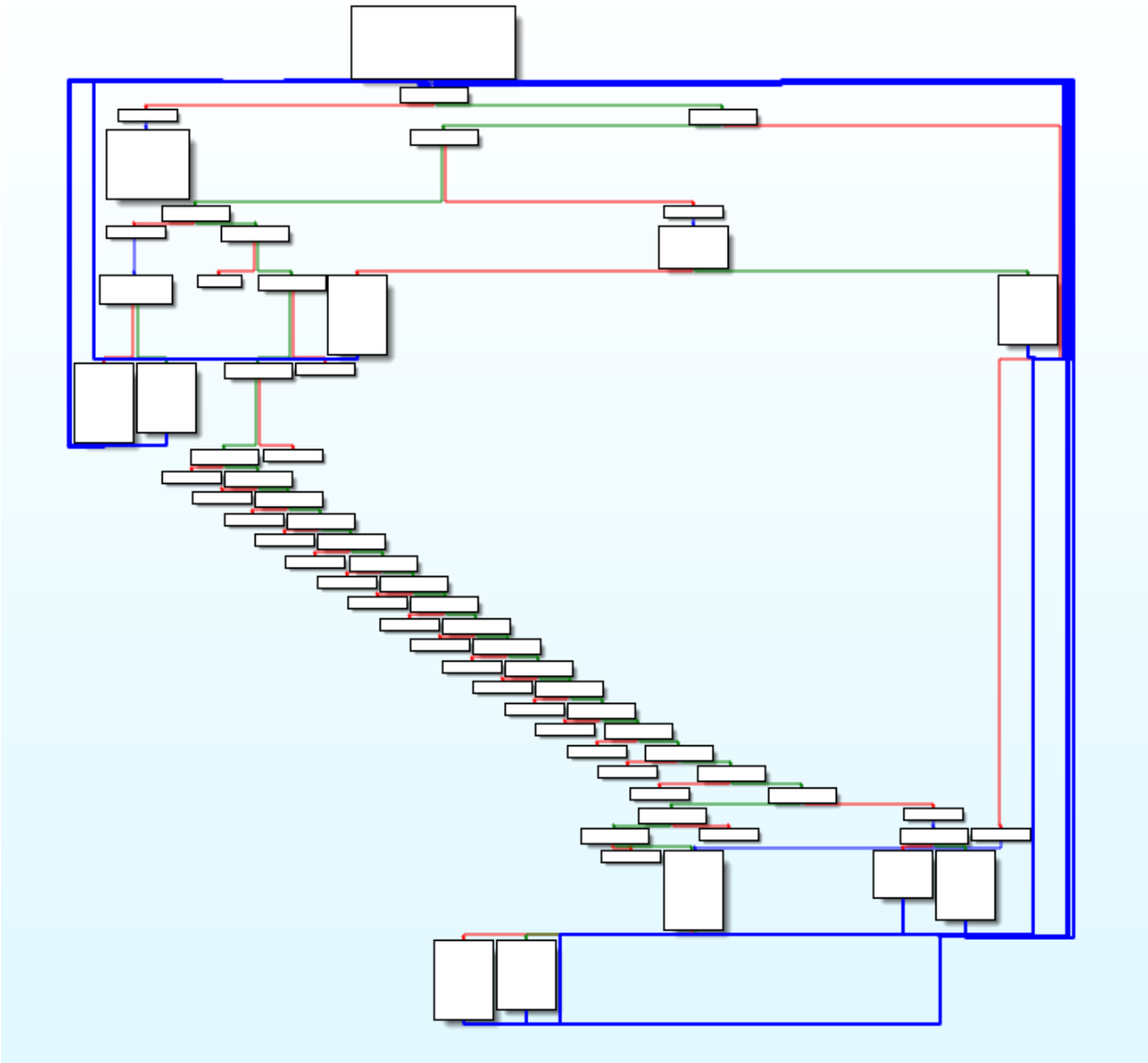
```
mov     edx, 0AA045890h
pushf
not     edx
add     edx, 0E6CAF175h
xor     edx, 0B7625898h
rol     edx, 0C4h
popf
pushf
not     edx
add     edx, 811E9B28h
xor     edx, 0C6E2935Fh
rol     edx, 0Fh
popf
```

Control Flow Obfuscation

The part of the function that makes it harder to read the flow by placing the normal flow in the function in reconstructed blocks. This is the normal flow of the main function:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near
sub     rsp, 28h
lea     rdx, a0xreverseCom ; "0xreverse.com"
lea     rcx, Format     ; "%s\n"
call    sub_140001010
mov     edx, 18h
lea     rcx, aD         ; "%d\n"
call    sub_140001010
call    cs:getchar
mov     eax, 1
add     rsp, 28h
retn
main endp
```

It translates into this (graph view):



Pseudo-Code representation:

```
switch ( v13 )
{
  case 13:
    __asm { popf }
    return test(a1, a2, a3, a4);
  case 7:
    __asm { popf }
    return sub_7FF79777C9A1(a1, a2, a3, a4);
  case 14:
    __asm { popf }
    return sub_7FF79777CD5F(a1, a2, a3, a4);
  case 11:
    __asm { popf }
    return sub_7FF79777CC19(a1, a2, a3, a4);
  case 25:
    __asm { popf }
    sub_7FF79777D219(a1, a2, a3, a4);
  case 8:
    __asm { popf }
    return sub_7FF79777CA68(a1, a2, a3, a4);
  case 16:
    __asm { popf }
    return sub_7FF79777CE28(a1, a2, a3, a4, a5, a6, a7, a8, a9);
  case 23:
    __asm { popf }
    return sub_7FF79777D129(a1, a2, a3, a4);
  case 9:
    __asm { popf }
    JUMPOUT(0x7FF79777CB00i64);
  case 18:
    __asm { popf }
    JUMPOUT(0x7FF79777CF32i64);
  case 24:
    __asm { popf }
    sub_7FF79777D1CB(a1, a2, a3, a4);
  case 17:
    __asm { popf }
    JUMPOUT(0x7FF79777CE88i64);
  case 5:
    __asm { popf }
    return sub_7FF79777C89C(a1, a2, a3, a4);
  case 22:
    __asm { popf }
    return sub_7FF79777D0AE(a1, a2, a3, a4);
  case 10:
    __asm { popf }
    return sub_7FF79777CB59(a1, a2, a3, a4);
  case 20:
    __asm { popf }
    JUMPOUT(0x7FF79777D036i64);
  case 15:
    __asm { popf }
    return sub_7FF79777CDF0(a1, a2, a3, a4);
  case 19:
```

Writing MOV Calculator Script with IDAPython

In Section 3.4, it is necessary to write a calculator for the MOV Obfuscation operation I mentioned. The functions that will be used in this script will also be needed for Control Flow Unflattening.

In the main function of the script, the `analyze_mov_obfuscation` function is called by getting the selected address on IDA.

```
def main():
   print("[~] 0xReverse - Alcatraz Deobfuscator IDA Script [~]")
   # Select a mutated function on IDA Screen
   start_address = idaapi.get_screen_ea()
   if selected_function := ida_funcs.get_func(start_address):
      function_name = idc.get_func_name(start_address)
      print(f"[+] Analyzing Function Name: {function_name}, Address:
{start_address:x}")
      deobfuscated_mov_ops = analyze_mov_obfuscation(func=selected_function)
      print(f"[+] Deobfuscated {deobfuscated_mov_ops} MOV obfuscation")
   else:
        print(f"[-] This address is not a function")
        exit(-1)
```

The `analyze_mov_obfuscation` function simply walks through the target function, finds the following pattern and calculates the required value.

```
pattern:
{
    mov     eax, VALUE
    pushf
    not     eax
    add     eax, VALUE
    xor     eax, VALUE
    rol     eax, VALUE
    popf
}
```

Before writing the analyze_mov_obfuscation function, it is necessary to add the functions that perform the operations wrapped with ROL, ROR and pushf&popf to the script:

```
rol = lambda val, r_bits, max_bits: ((val << (r_bits % max_bits)) &
(2**max_bits - 1)) | ((val & (2**max_bits - 1)) >> (max_bits - (r_bits %
max_bits)))
ror = lambda val, r_bits, max_bits: ((val & (2**max_bits - 1)) >> (r_bits %
max_bits)) | ((val << (max_bits - (r_bits % max_bits))) & (2**max_bits - 1))
MASK = 0xFFFFFFFF

def calculate_mov_value(init_value, add_value, xor_value, rol_value):
   # mov     eax, VALUE
   calculated_value = init_value
   # not     eax         -> bitwise NOT
   calculated_value = (~calculated_value) & MASK
   # add     eax, VALUE
   calculated_value = (calculated_value + add_value) & MASK
   # xor     eax, VALUE
   calculated_value = calculated_value ^ xor_value
   # rol     eax, VALUE
   calculated_value = rol(calculated_value, rol_value, 32)
   return calculated_value
```

For this pattern we only need to check the initial mov and pushf values. So we check the instruction with current_address and the next instruction:

```
# Get function start address
current_address = func.start_ea
while current_address < func.end_ea:
  instruction = idautils.DecodeInstruction(current_address)
  if not instruction:
     break

  if instruction.itype == idaapi.NN_mov:
     # Check if the next opcode is pushf
     next_instruction = idautils.DecodeInstruction(current_address +
instruction.size)
     if next_instruction.itype == idaapi.NN_pushf:
        print("[+] MOV Obfuscation pattern found!")
        ...
     else:
        current_address += instruction.size
  else:
     current_address += instruction.size
```

After the initial address is assigned to a register, we cannot calculate this pattern once because the number of times the special calculation is done is randomly selected. Therefore, it is necessary to continue in a loop until we get a different opcode other than the specific opcodes. There is an example of this process done 3 times in the same function:

```
mov     ecx, 5600B3EBh
pushf
not     ecx
add     ecx, 84DD6D12h
xor     ecx, 84F2EEE7h
rol     ecx, 2Bh
popf
pushf
not     ecx
add     ecx, 0B4156550h
xor     ecx, 0B460F07Dh
rol     ecx, 5Bh
popf
pushf
not     ecx
add     ecx, 0E02D13C8h
xor     ecx, 0C483568Bh
rol     ecx, 66h
popf
```

This is how we do the calculation process in the script:

```
calculated_value = 0
add_value = xor_value = rol_value = 0
# Get value from {mov eax, VALUE} and MASK for 32bit
initial_value = instruction.ops[1].value & MASK
# Add current_address and mov and pushf
current_address += instruction.size + next_instruction.size
while True:
   current_instruction = idautils.DecodeInstruction(current_address)
   if current_instruction.itype == idaapi.NN_not:
      current_address += current_instruction.size
   elif current_instruction.itype == idaapi.NN_add:
      add_value = current_instruction.ops[1].value & MASK
      current_address += current_instruction.size
   elif current_instruction.itype == idaapi.NN_xor:
      xor_value = current_instruction.ops[1].value & MASK
      current_address += current_instruction.size
   elif current_instruction.itype == idaapi.NN_rol:
      rol_value = current_instruction.ops[1].value & MASK
      current_address += current_instruction.size
   elif current_instruction.itype == idaapi.NN_popf:
      calculated_value = calculate_mov_value(
         init_value=initial_value,
         add_value=add_value,
         xor_value=xor_value,
         rol_value=rol_value
      )
      current_address += current_instruction.size
   elif current_instruction.itype == idaapi.NN_pushf:
      # The pattern we are looking for can be repetitive,
      # so we need to keep the process going.
      initial_value = calculated_value
      current_address += current_instruction.size
   else:
      break
print(f"[+] Calculated MOV value: {hex(calculated_value)}")
```

Github Repo

YARA Rule

```
import "pe"

rule SUSP_EXE_Alcatraz_Obfuscator_April_23 {
    meta:
        description = "This rule detects samples obfuscated with Alcatraz."
        author      = "Utku Corbaci (rhotav) / 0xReverse"
        date        = "2025-04-23"
        sharing     = "TLP:CLEAR"
        tags        = "windows,exe,suspicious,obfuscator"
        os          = "Windows"

    strings:
        // B8 41 CD A8 27   mov     eax, 27A8CD41h
        // 66 9C            pushf
        // F7 D0            not     eax
        // 05 AB FD E1 DD   add     eax, 0DDE1FDABh
        // 35 CA 3C 0F BF   xor     eax, 0BF0F3CCAh
        // C1 C0 62         rol     eax, 62h
        // 66 9D            popf
        $obfuscation_mov = {B8 ?? ?? ?? ?? 66 9C F7 D0 05 ?? ?? ?? ?? 35 ?? ??
?? ?? C1 C0 ?? 66 9D}

        // 48 8D 05 74 81 47 77        lea     rax, cs:1B748621Eh
        // 66 9C                       pushf
        // 48 2D A6 2B 48 77           sub     rax, 77482BA6h
        // 66 9D                       popf
        $obfuscation_lea = {48 8D ?? ?? ?? ?? ?? 66 9C 48 2D ?? ?? ?? ?? 66 9D}
    condition:
        pe.is_pe
        and for any i in (0..pe.number_of_sections - 1): (
            (pe.sections[i].name == ".0Dev")
        )
        and (all of ($obfuscation_*))
}
```

unpac.me Hunting Results

References

1. https://keowu.re/posts/Analyzing-Mutation-Coded-VM-Protect-and-Alcatraz-English
2. https://gist.github.com/icecr4ck/7a7af3277787c794c66965517199fc9c
3. https://python.docs.hex-rays.com/
4. https://grazfather.github.io/posts/2016-09-18-anti-disassembly/
5. https://gist.github.com/trietptm/5cd60ed6add5adad6a34098ce255949a

6. https://phrack.org/issues/71/15