De-obfuscating ALCATRAZ

By Daniel Stepanic

Published: 2025-05-23 · Archived: 2026-04-05 16:30:52 UTC

Introduction

Elastic Security Labs analyzes diverse malware that comes through our threat hunting pipelines and telemetry queues. We recently ran into a new malware family called DOUBLELOADER, seen alongside the RHADAMANTHYS infostealer. One interesting attribute of DOUBLELOADER is that it is protected with an open-source obfuscator, ALCATRAZ first released in 2023. While this project had its roots in the game hacking community, it's also been observed in the e-crime space, and has been used in targeted intrusions.

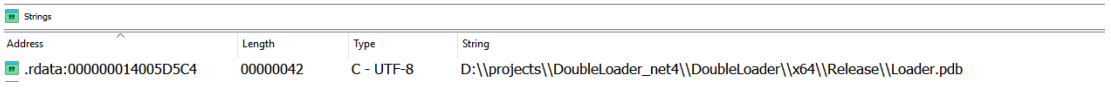
The objective of this post is to walk through various obfuscation techniques employed by ALCATRAZ, while highlighting methods to combat these techniques as malware analysts. These techniques include control flow flattening, instruction mutation, constant unfolding, LEA constant hiding, anti-disassembly tricks and entrypoint obfuscation.

Key takeaways

- The open-source obfuscator ALCATRAZ has been seen within new malware deployed alongside RHADAMANTHYS infections
- Obfuscation techniques such as control flow flattening continue to serve as road blocks for analysts
- By understanding obfuscation techniques and how to counter them, organizations can improve their ability to effectively triage and analyze protected binaries.
- Elastic Security Labs releases tooling to deobfuscate ALCATRAZ protected binaries are released with this post

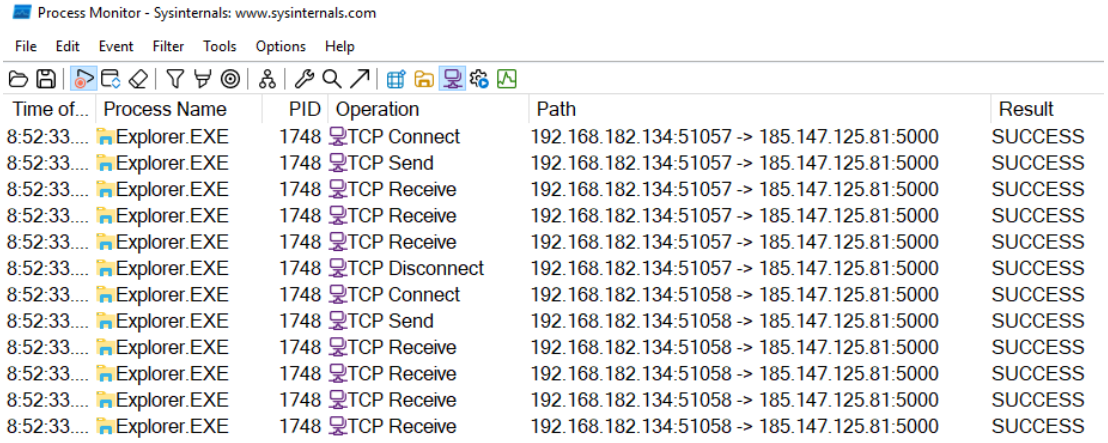
DOUBLELOADER

Starting last December, our team observed a generic backdoor malware coupled with RHADAMANTHYS stealer infections. Based on the PDB path, this malware is self-described as DOUBLELOADER.



PDB path in DOUBLELOADER

This malware leverages syscalls such as `NtOpenProcess`, `NtWriteVirtualMemory`, `NtCreateThreadEx` launching unbacked code within the Windows desktop/file manager (`explorer.exe`). The malware collects host information, requests an updated version of itself and starts beaconing to a hardcoded IP (`185.147.125.81`) stored within the binary.



Time of...	Process Name	PID	Operation	Path	Result
8:52:33....	Explorer.EXE	1748	TCP Connect	192.168.182.134:51057 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Send	192.168.182.134:51057 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Receive	192.168.182.134:51057 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Receive	192.168.182.134:51057 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Receive	192.168.182.134:51057 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Disconnect	192.168.182.134:51057 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Connect	192.168.182.134:51058 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Send	192.168.182.134:51058 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Receive	192.168.182.134:51058 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Receive	192.168.182.134:51058 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Receive	192.168.182.134:51058 -> 185.147.125.81:5000	SUCCESS
8:52:33....	Explorer.EXE	1748	TCP Receive	192.168.182.134:51058 -> 185.147.125.81:5000	SUCCESS

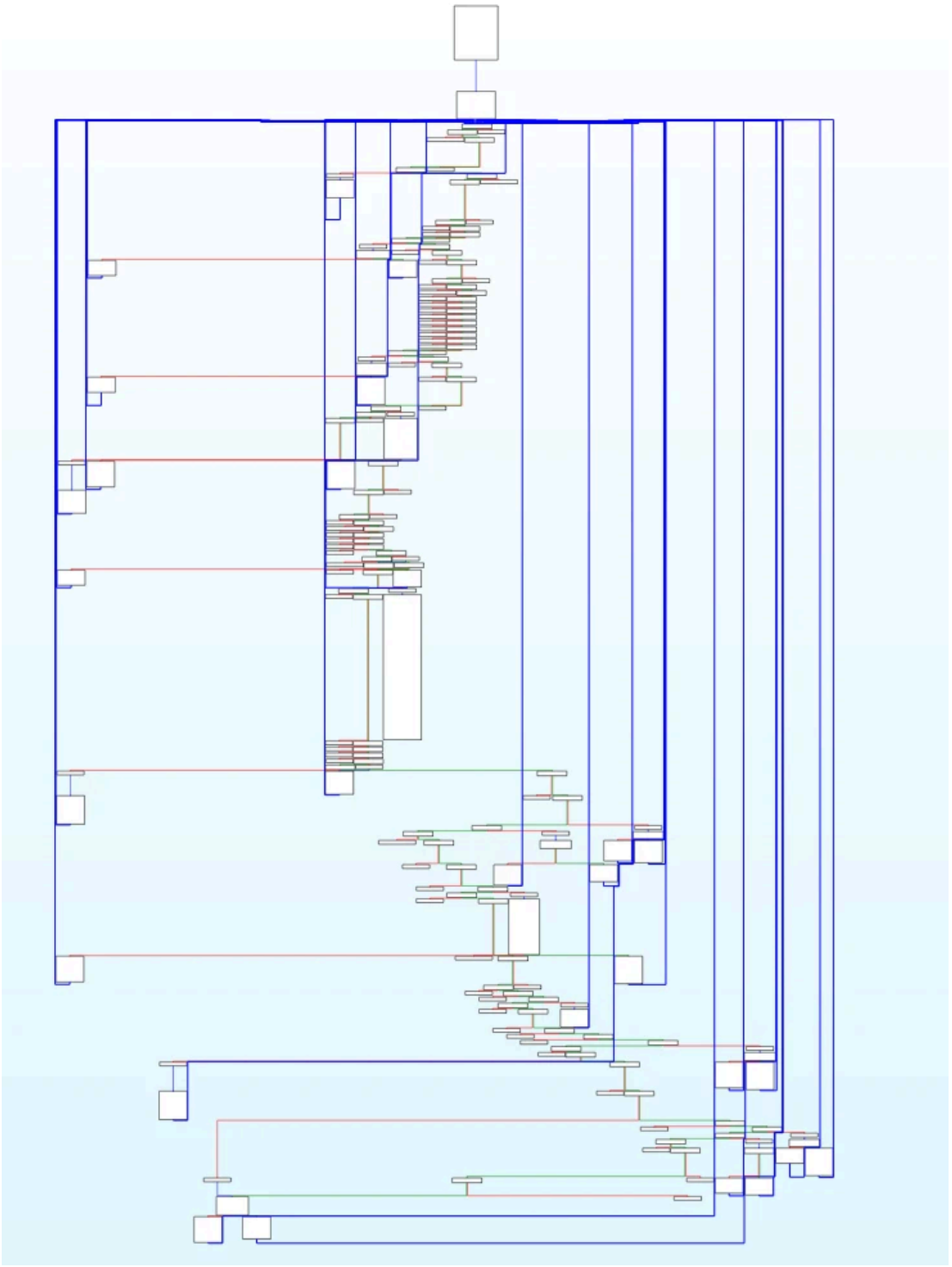
Outbound C2 traffic from DOUBLELOADER

DOUBLELOADER samples include a non-standard section (`.0Dev`) with executable permissions, this is a toolmark left based on the author's handle for the binary obfuscation tool, `ALCATRAZ` .

```
auto functions = pdb.parse_functions();
std::cout << "Successfully parsed " << functions.size() << " function(s)" << std::endl;

auto new_section = pe.create_section(".0Dev", 10000000, IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ | IMAGE_SCN_CNT_CODE);
```

Section creation using ALCATRAZ

Obfuscators such as ALCATRAZ end up increasing the complexity when triaging malware. Its main goal is to hinder binary analysis tools and increase the time of the reverse engineering process through different techniques; such as hiding the control flow or making decompilation hard to follow. Below is an example of obfuscated control flow of one function inside DOUBLELOADER.



Obfuscated control flow in DOUBLELOADER

The remainder of the post will focus on the various obfuscation techniques used by ALCATRAZ. We will use the first-stage of DOUBLELOADER along with basic code examples to highlight ALCATRAZ's features.

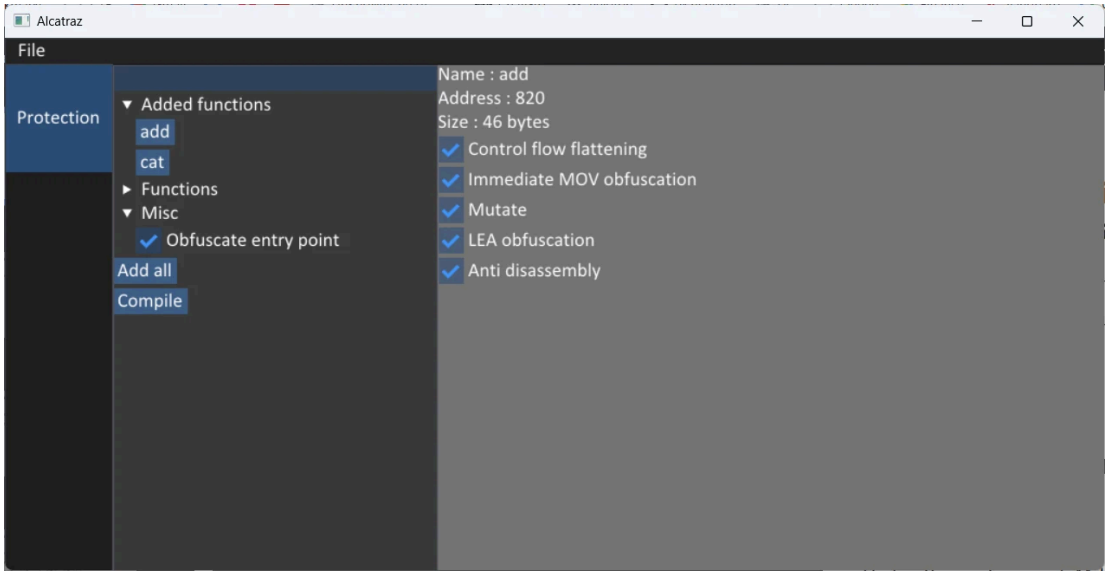
ALCATRAZ

ALCATRAZ Overview

Alcatraz is an open-source obfuscator initially released in January 2023. While the project is recognized within the game hacking community as a foundational tool for learning obfuscation techniques, it’s also been observed being abused by e-

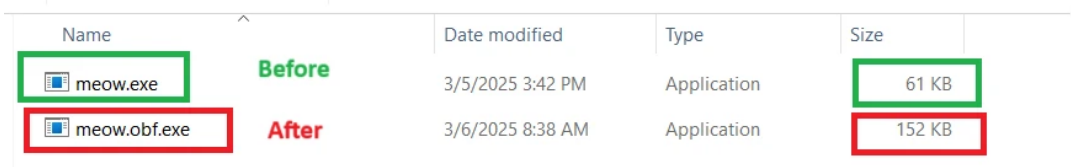
crime and APT groups.

Alcatraz's code base contains 5 main features centered around standard code obfuscation techniques along with enhancement to obfuscate the entrypoint. Its workflow follows a standard `bin2bin` format, this means the user provides a compiled binary then after the transformations, they will receive a new compiled binary. This approach is particularly appealing to game hackers/malware developers due to its ease of use, requiring minimal effort and no modifications at the source code level.



ALCATRAZ - menu

The developer can choose to obfuscate all or specific functions as well as choose which obfuscation techniques to apply to each function. After compilation, the file is generated with the string (`obf`) appended to the end of the filename.



Example of binary before and after obfuscation

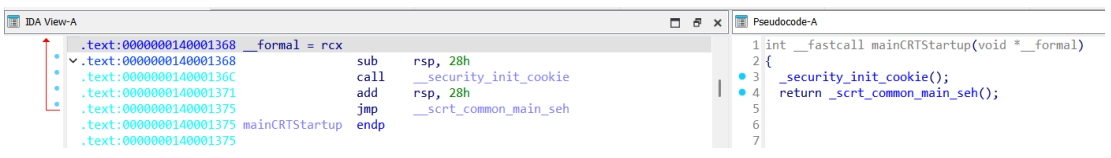
Obfuscation techniques in ALCATRAZ

The following sections will go through the various obfuscation techniques implemented by ALCATRAZ.

Entrypoint obfuscation

Dealing with an obfuscated entrypoint is like getting a flat tire at the start of a family roadtrip. The idea is centered on confusing analysts and binary tooling where it's not directly clear where the program starts, causing confusion at the very beginning of the analysis process.

The following is the view of a clean entrypoint (`0x140001368`) from a non-obfuscated program within IDA Pro.



Non-obfuscated entrypoint

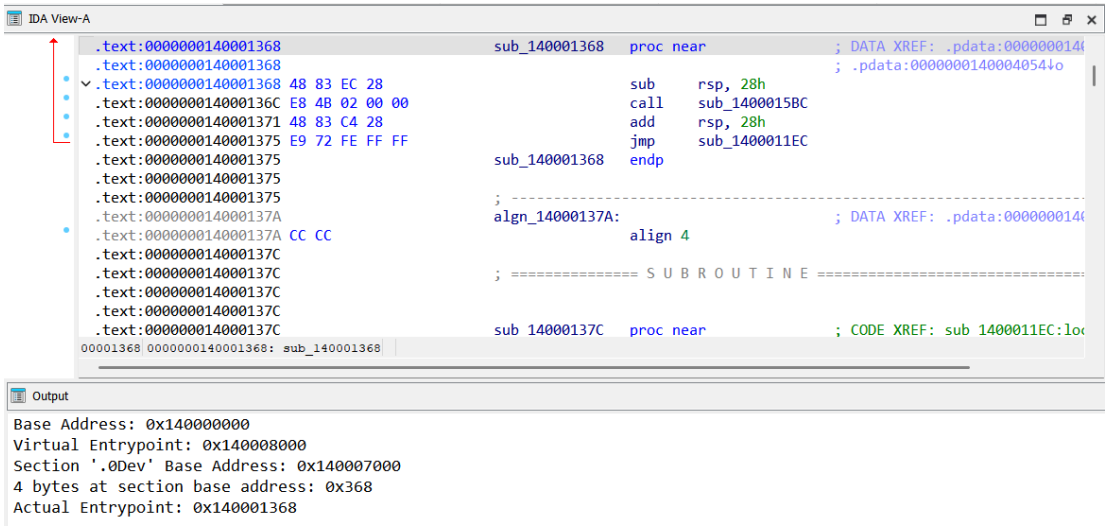
By enabling entrypoint obfuscation, ALCATRAZ moves the entrypoint then includes additional code with an algorithm to calculate the new entrypoint of the program. Below is a snippet of the decompiled view of the obfuscated entry-point.

```
__int64 __fastcall start(__int64 a1, unsigned int a2, signed __int64 diff_text_zerodev_sec)
{
  __int128 *raw_addr; // r11
  signed __int64 _TIB; // rsi
  char *ImageBaseAddress; // rdi
  IMAGE_NT_HEADERS64 *nt_header; // rbx
  __int64 num_sections; // r10
  __int64 section_able; // r9
  int *v10; // rcx
  __int128 v11; // xmm0
  __int64 v12; // xmm1_8
  char v13; // dl
  char v14; // al
  bool v15; // zf
  __int128 *v16; // rdx
  int v18[2]; // [rsp+20h] [rbp-38h] BYREF
  __int128 v19[2]; // [rsp+28h] [rbp-30h] BYREF
  __int64 v20; // [rsp+48h] [rbp-10h]

  raw_addr = 0LL;
  _TIB = diff_text_zerodev_sec;
  ImageBaseAddress = NtCurrentPeb()->ImageBaseAddress;
  nt_header = &ImageBaseAddress[*(ImageBaseAddress + 0xF)];
  num_sections = nt_header->FileHeader.NumberOfSections;
  section_able = &nt_header->OptionalHeader + nt_header->FileHeader.SizeOfOptionalHeader;
  if ( !nt_header->FileHeader.NumberOfSections )
    return (&ImageBaseAddress[__ROR4__(
                                LODWORD(nt_header->OptionalHeader.SizeOfStackCommit) ^ *&ImageBaseAddress[*(raw_addr + 3)],
                                nt_header->FileHeader.TimeDateStamp)])(
             a2,
             _TIB,
             diff_text_zerodev_sec,
             section_able);
```

Decompilation of obfuscated entrypoint

As ALCATRAZ is an open-source obfuscator, we can find the custom entrypoint code to see how the calculation is performed or reverse our own obfuscated example. In our decompilation, we can see the algorithm uses a few fields from the PE header such as the `Size of the Stack Commit`, `Time Date Stamp` along with the first four bytes from the `.0dev` section. These fields are parsed then used with bitwise operations such as rotate right (ROR) and exclusive-or (XOR) to calculate the entrypoint.

Below is an example output of IDA Python script (Appendix A) that parses the PE and finds the true entrypoint, confirming the original starting point (`0x140001368`) with the non-obfuscated sample.



Real entrypoint after obfuscation

Anti-disassembly

Malware developers and obfuscators use anti-disassembly tricks to confuse or break disassemblers in order to make static analysis harder. These techniques abuse weaknesses during linear sweeps and recursive disassembly, preventing clean code reconstruction where the analyst is then forced to manually or automatically fix the underlying instructions.

ALCATRAZ implements one form of this technique by modifying any instructions starting with the `0xFF` byte by adding a short jump instruction (`0xEB`) in front. The `0xFF` byte can represent the start of multiple valid instructions dealing with calls, indirect jumps, pushes on the stack. By adding the short jump `0xEB` in front, this effectively jumps to the next byte `0xFF` . While it's not complex, the damage is done breaking disassembly and requiring some kind of intervention.

```
.0Dev:000000018017BA6D
.0Dev:000000018017BA6D                          loc_18017BA6D:                          ; CODE XREF: .0Dev:000000018017B9C4↑j
.0Dev:000000018017BA6D 48 8B 01                 mov     rax, [rcx]
.0Dev:000000018017BA70 BA B6 33 9D 1E           mov     edx, 1E9D33B6h
.0Dev:000000018017BA75 66 9C                    pushf
.0Dev:000000018017BA77 F7 D2                    not     edx
.0Dev:000000018017BA79 81 C2 D8 52 44 91        add     edx, 914452D8h
.0Dev:000000018017BA7F 81 F2 49 C7 D5 CB        xor     edx, 0CBD5C749h
.0Dev:000000018017BA85 C1 C2 7A                 rol     edx, 7Ah
.0Dev:000000018017BA88 66 9D                    popf
.0Dev:000000018017BA8A 66 9C                    pushf
.0Dev:000000018017BA8C F7 D2                    not     edx
.0Dev:000000018017BA8E 81 C2 DC 7B 2E A3        add     edx, 0A32E7BDCh
.0Dev:000000018017BA94 81 F2 C8 10 97 87        xor     edx, 879710C8h
.0Dev:000000018017BA9A C1 C2 27                 rol     edx, 27h
.0Dev:000000018017BA9D 66 9D                    popf
.0Dev:000000018017BA9F 66 9C                    pushf
.0Dev:000000018017BAA1 F7 D2                    not     edx
.0Dev:000000018017BAA3 81 C2 BE D6 5C BA        add     edx, 0BA5CD6BEh
.0Dev:000000018017BAA9 81 F2 7A 7F 8C CA        xor     edx, 0CA8C7F7Ah
.0Dev:000000018017BAAF C1 C2 D7                 rol     edx, 0D7h
.0Dev:000000018017BAB2 66 9D                    popf
.0Dev:000000018017BAB4
.0Dev:000000018017BAB4                          loc_18017BAB4:                          ; CODE XREF: .0Dev:loc_18017BAB4↑j
.0Dev:000000018017BAB4 EB FF                    jmp     short near ptr loc_18017BAB4+1
```

Anti-disassembly technique in ALCATRAZ

In order to fix this specific technique, the file can be patched by replacing each occurrence of the `0xEB` byte with NOPs. After patching, the code is restored to a cleaner state, allowing the following `call` instruction to be correctly disassembled.

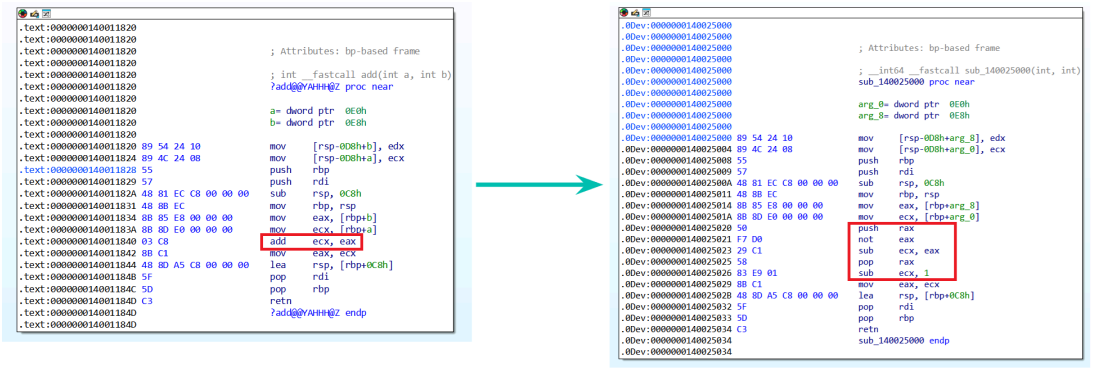
```
.0Dev:000000018017BA6D                          arg_28          = qword ptr  30h
.0Dev:000000018017BA6D
.0Dev:000000018017BA6D 48 8B 01                 mov     rax, [rcx]
.0Dev:000000018017BA70 BA B6 33 9D 1E           mov     edx, 1E9D33B6h
.0Dev:000000018017BA75 66 9C                    pushf
.0Dev:000000018017BA77 F7 D2                    not     edx
.0Dev:000000018017BA79 81 C2 D8 52 44 91        add     edx, 914452D8h
.0Dev:000000018017BA7F 81 F2 49 C7 D5 CB        xor     edx, 0CBD5C749h
.0Dev:000000018017BA85 C1 C2 7A                 rol     edx, 7Ah
.0Dev:000000018017BA88 66 9D                    popf
.0Dev:000000018017BA8A 66 9C                    pushf
.0Dev:000000018017BA8C F7 D2                    not     edx
.0Dev:000000018017BA8E 81 C2 DC 7B 2E A3        add     edx, 0A32E7BDCh
.0Dev:000000018017BA94 81 F2 C8 10 97 87        xor     edx, 879710C8h
.0Dev:000000018017BA9A C1 C2 27                 rol     edx, 27h
.0Dev:000000018017BA9D 66 9D                    popf
.0Dev:000000018017BA9F 66 9C                    pushf
.0Dev:000000018017BAA1 F7 D2                    not     edx
.0Dev:000000018017BAA3 81 C2 BE D6 5C BA        add     edx, 0BA5CD6BEh
.0Dev:000000018017BAA9 81 F2 7A 7F 8C CA        xor     edx, 0CA8C7F7Ah
.0Dev:000000018017BAAF C1 C2 D7                 rol     edx, 0D7h
.0Dev:000000018017BAB2 66 9D                    popf
.0Dev:000000018017BAB4 90                       nop
.0Dev:000000018017BAB5 FF 10                    call    qword ptr [rax]
```

Anti-disassembly recovery

Instruction Mutation

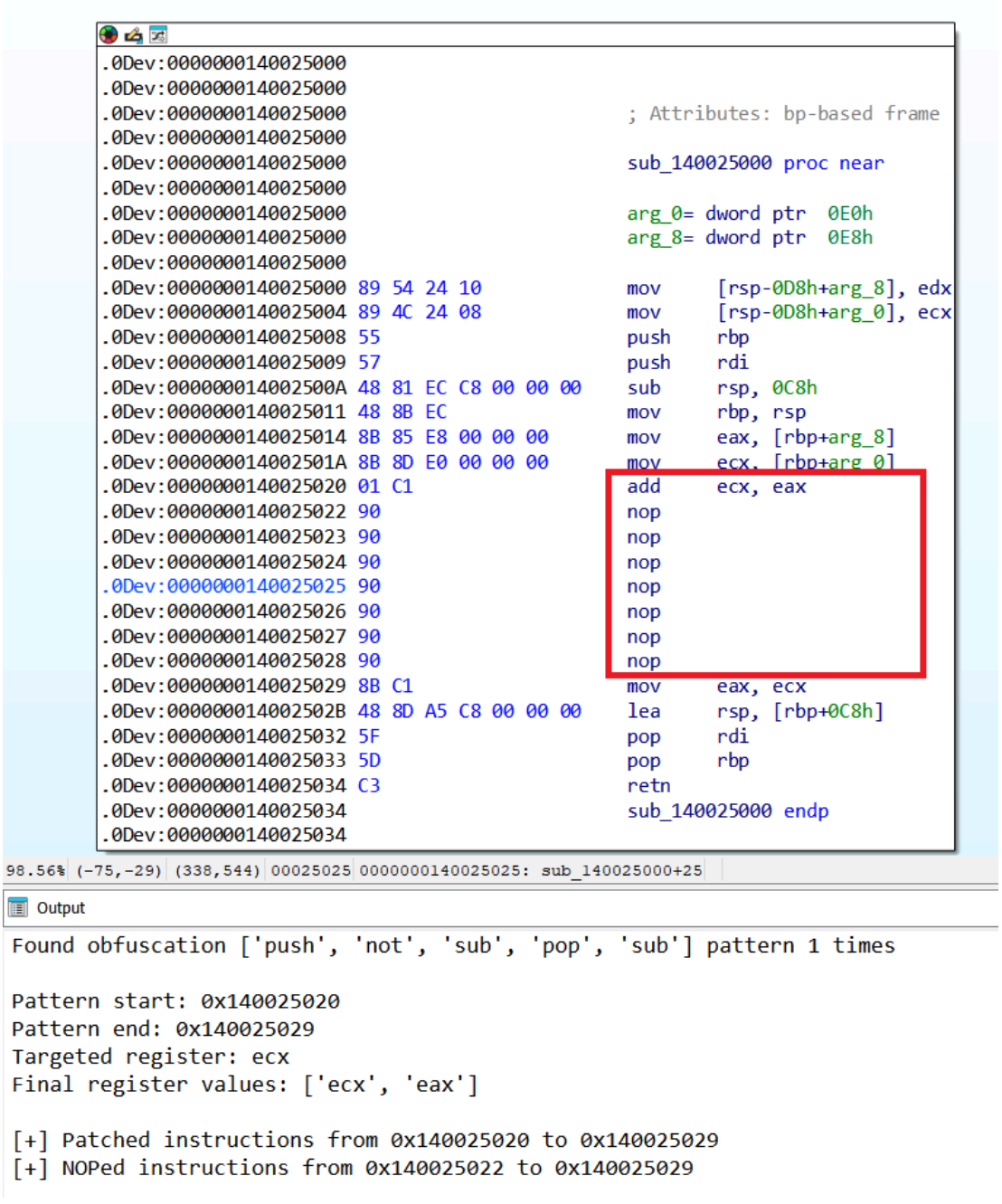
One common technique used by obfuscators is instruction mutation, where instructions are transformed in a way that preserves their original behavior, but makes the code harder to understand. Frameworks such as Tigress or Perses are great examples of obfuscation research around instruction mutation.

Below is an example of this technique implemented by ALCATRAZ, where any addition between two registers is altered, but its semantic equivalence is kept intact. The simple `add` instruction gets transformed to 5 different instructions (`push` , `not` , `sub` , `pop` , `sub`).



Example of instruction mutation via ALCATRAZ

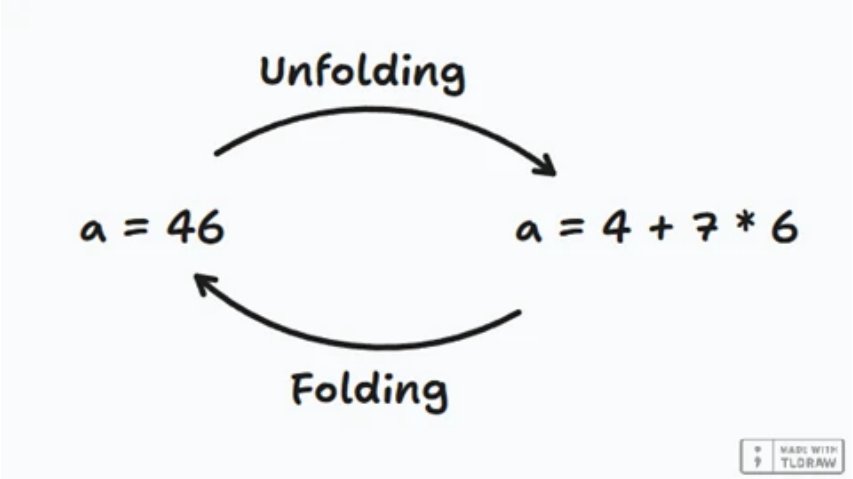
In order to correct this, we can use pattern matching to find these 5 instructions together, disassemble the bytes to find which registers are involved, then use an assembler such as Keystone to generate the correct corresponding bytes.



Recovering instructions from mutation technique

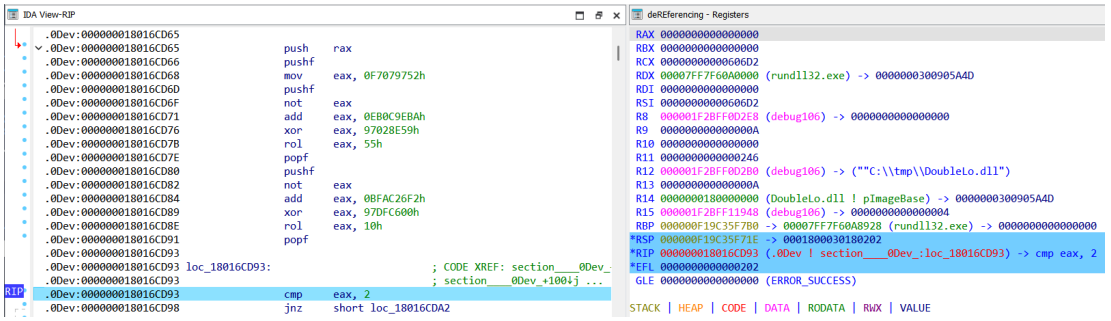
Constant Unfolding

This obfuscation technique is prevalent throughout the DOUBLELOADER sample and is a widely used method in various forms of malware. The concept here is focused on inversing the compilation process; where instead of optimizing calculations that are known at compile time, the obfuscator “unfolds” these constants making the disassembly and decompilation complex and confusing. Below is a simple example of this technique where the known constant (`46`) is broken up into two mathematical operations.



Unfolding process example

In DOUBLELOADER, we run into this technique being used anytime when immediate values are moved into a register. These immediate values are replaced with multiple bitwise operations masking these constant values, thus disrupting any context and the analyst's flow. For example, in the disassembly below on the left-hand side, there is a comparison instruction of EAX value at address (`0x18016CD93`). By reviewing the previous instructions, it's not obvious or clear what the EAX value should be due to multiple obscure bitwise calculations. If we debug the program, we can see the EAX value is set to `0` .



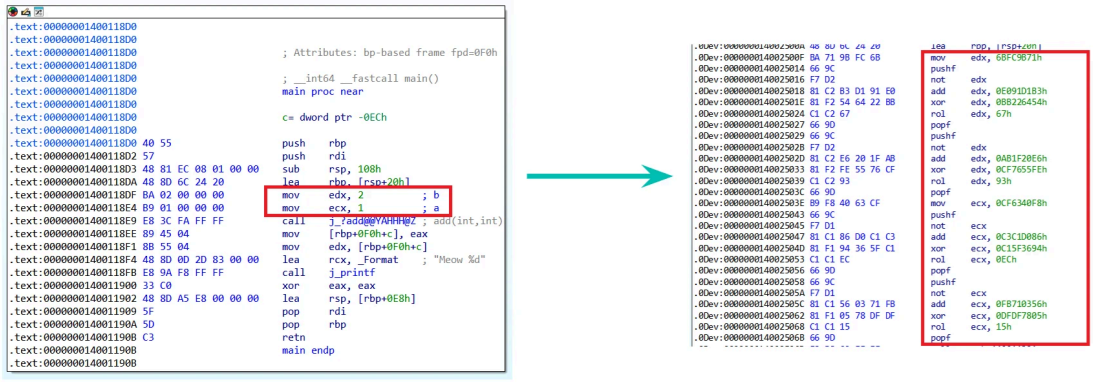
Viewing unfolding technique in debugger

In order to clean this obfuscation technique, we can confirm its behavior with our own example where we can use the following source code and see how the transformation is applied.

```
#include <iostream>

int add(int a, int b)

{

        return a + b;

}

int main()

{

        int c;
```

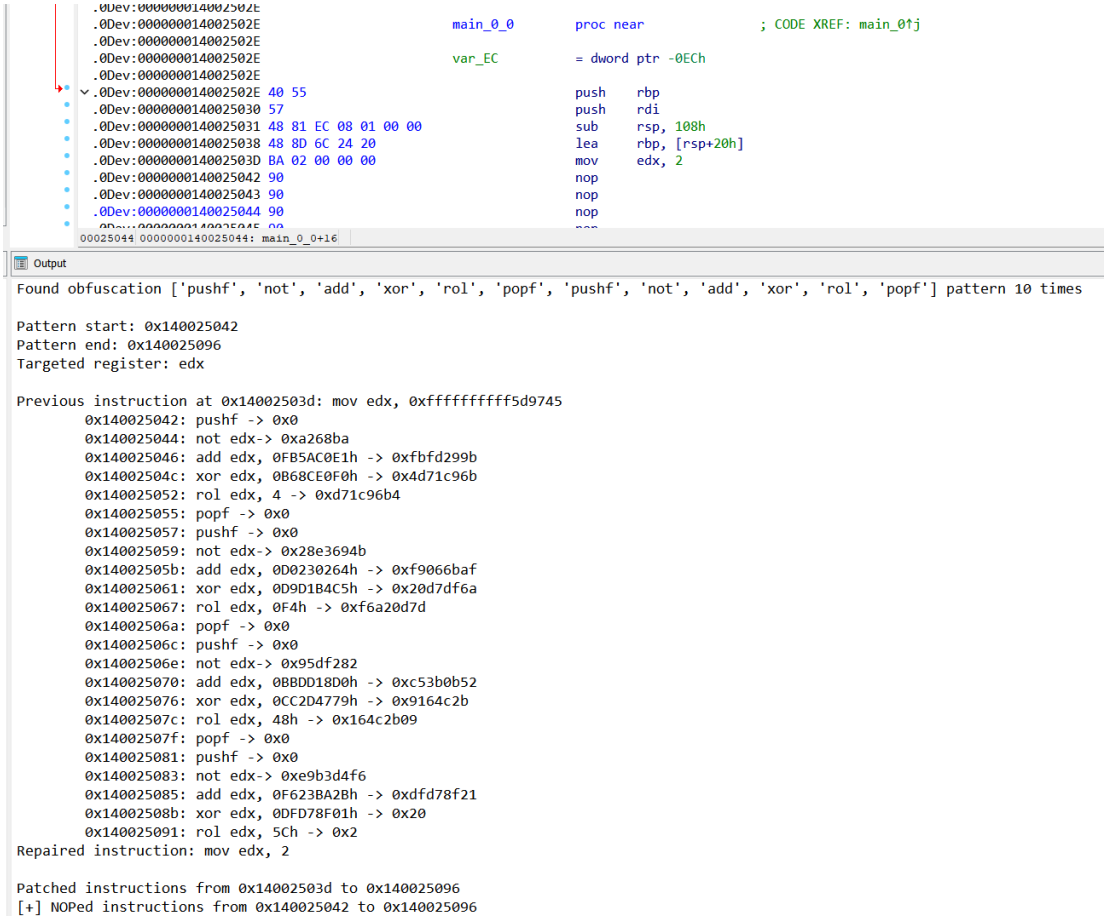
```
        c = add(1, 2);

        printf("Meow %d",c);

        return 0;

}
```

After compiling, we can view the disassembly of the `main` function in the clean version on the left and see these two constants (`2,1`) moved into the EDX and ECX register. On the right side, is the transformed version, the two constants are hidden among the newly added instructions.



Mutation transformation: before vs after

By using pattern matching techniques, we can look for these sequences of instructions, emulate the instructions to perform the various calculations to get the original values back, and then patch the remaining bytes with NOP's to make sure the program will still run.

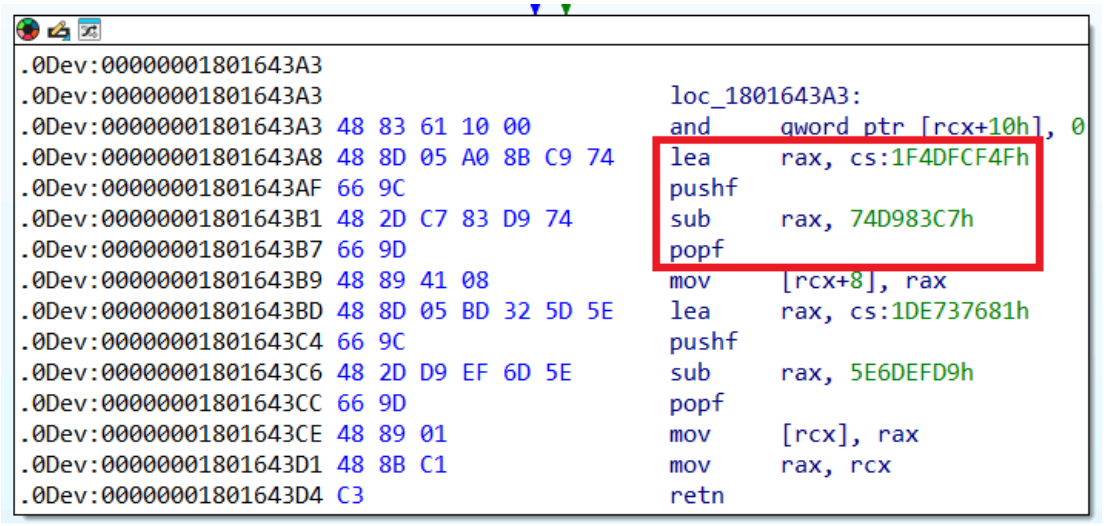


Using emulation to repair immediate moves

LEA Obfuscation

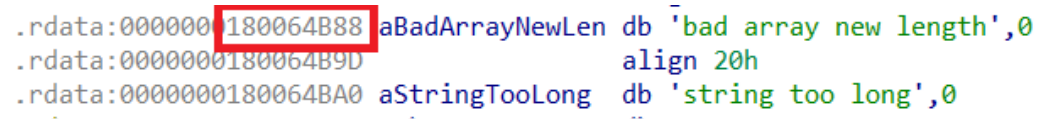
Similar to the previously discussed technique, LEA (Load Effective Address) obfuscation is focused on obscuring the immediate values associated with LEA instructions. An arithmetic calculation with subtraction will follow directly behind the LEA instruction to compute the original intended value. While this may seem like a minor change, it can have a significant impact breaking cross-references to strings and data — which are essential for effective binary analysis.

Below is an example of this technique within DOUBLELOADER where the RAX register value is disguised through a pattern of loading an initial value (`0x1F4DFCF4F`), then subtracting (`0x74D983C7`) to give us a new computed value (`0x180064B88`).



LEA obfuscation pattern in ALCATRAZ

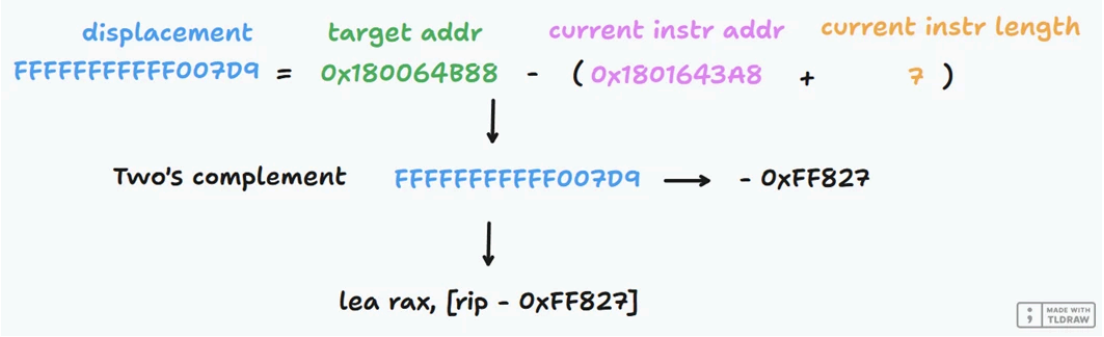
If we go to that address inside our sample, we are taken to the read-only data section, where we can find the referenced string `bad array new length`.



Referenced string after LEA obfuscation

In order to correct this technique, we can use pattern matching to find these specific instructions, perform the calculation, then re-construct a new LEA instruction. Within 64-bit mode, LEA uses RIP-relative addressing so the address is calculated based on the current instruction pointer (RIP). Ultimately, we end up with a new instruction that looks like this: `lea rax, [rip - 0xFF827]`.

Below are the steps to produce this final instruction:



Displacement calculation for LEA instruction

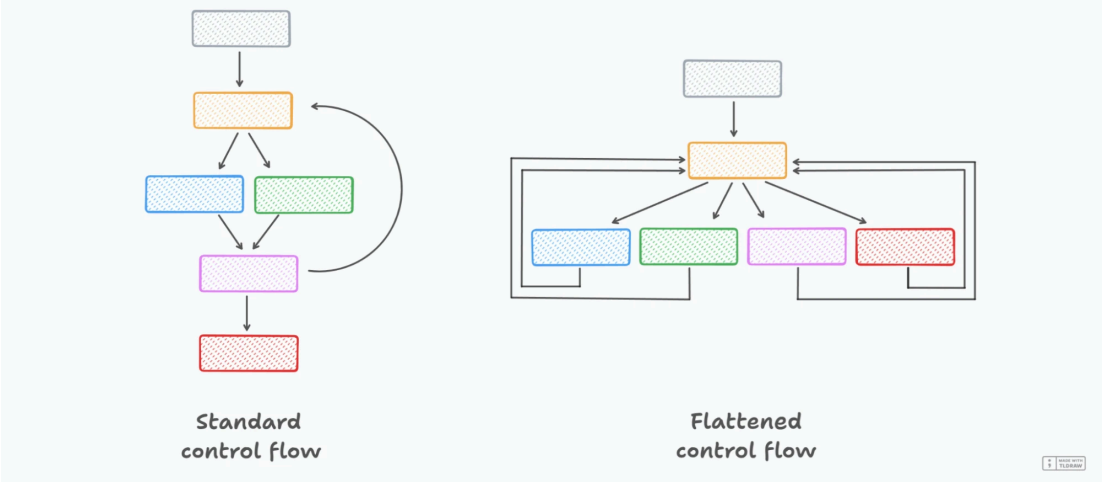
With this information, we can use IDA Python to patch all these patterns out, below is an example of a fixed LEA instruction.

```
.0Dev:00000001801643A3                          loc_1801643A3:                          ; CODE XREF: sub_180164357+42↑j
.0Dev:00000001801643A3                                                                  ; sub_180164357+47↑j
.0Dev:00000001801643A3 48 83 61 10 00                           and     qword ptr [rcx+10h], 0
.0Dev:00000001801643A8 48 8D 05 D9 07 F0 FF                     lea     rax, aBadArrayNewLen ; "bad array new length"
.0Dev:00000001801643AF 90                                       nop
.0Dev:00000001801643B0 90                                       nop
.0Dev:00000001801643B1 90                                       nop
.0Dev:00000001801643B2 90                                       nop
.0Dev:00000001801643B3 90                                       nop
.0Dev:00000001801643B4 90                                       nop
.0Dev:00000001801643B5 90                                       nop
.0Dev:00000001801643B6 90                                       nop
.0Dev:00000001801643B7 90                                       nop
.0Dev:00000001801643B8 90                                       nop
.0Dev:00000001801643B9 90                                       nop
.0Dev:00000001801643BA 89 41 08                                 mov     [rcx+8], eax
0016439D 00000001801643D: sub_180164357+46
Output
Found obfuscation ['lea', 'pushf', 'sub', 'popf'] pattern 144 times

Pattern start: 0x1801643a8
Pattern end: 0x1801643ba
Targeted register: rax
Targeted value: 0x1f4dfcf4f
Instruction: lea rax, [rip -1046567]

Patched instructions from 0x1801643a8 to 0x1801643ba
[+] NOPed instructions from 0x1801643af to 0x1801643ba
```

Patching LEA instructions in DOUBLELOADER

Control Flow Obfuscation

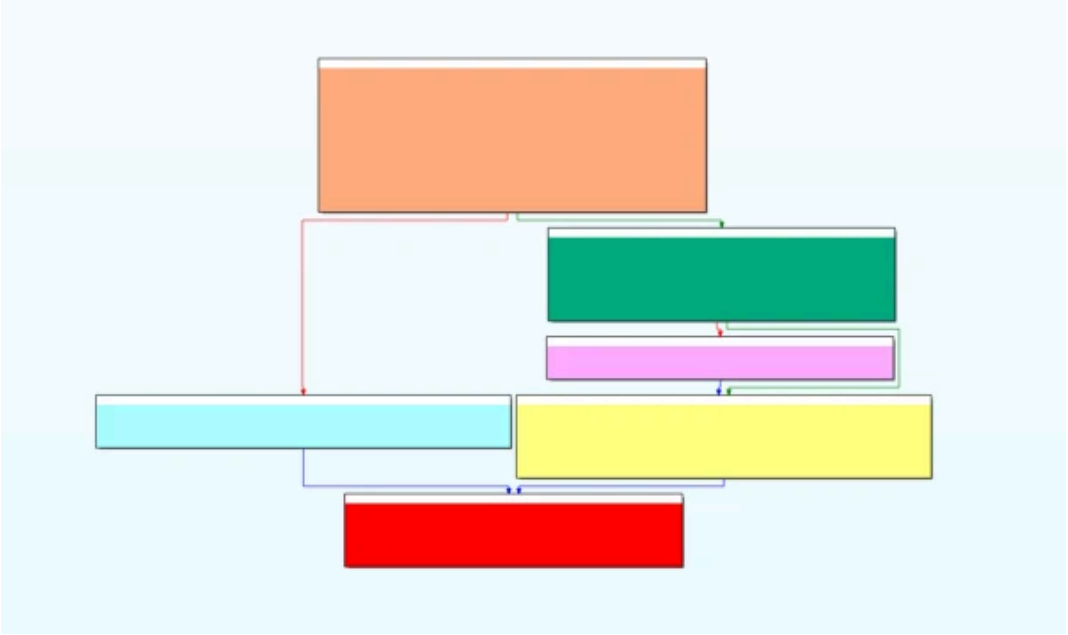
Control flow flattening is a powerful obfuscation technique that disrupts the traditional structure of a program's control flow by eliminating conventional constructs like conditional branches and loops. Instead, it restructures execution using a centralized dispatcher, which determines the next basic block to execute based on a state variable, making analysis and decompilation significantly more difficult. Below is a simple diagram that represents the differences between an unflattened and flattened control flow.



Standard control flow vs flattened control flow

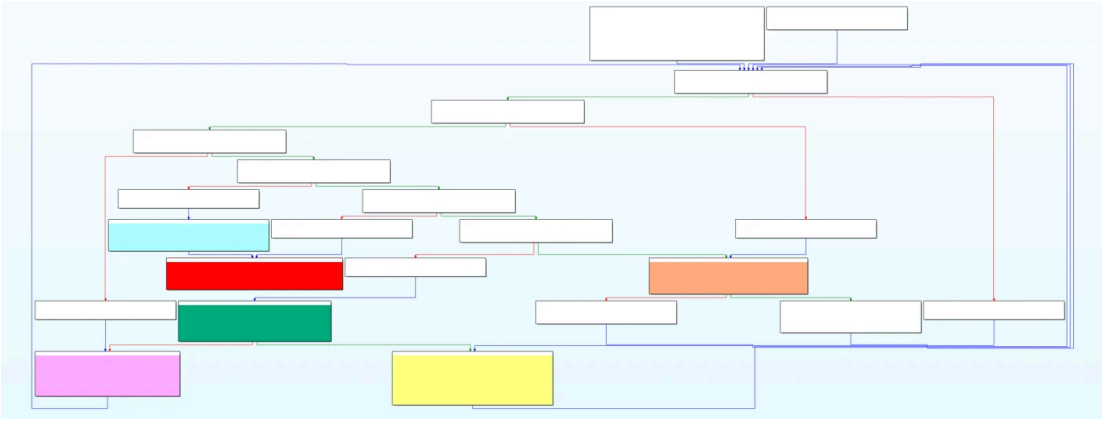
Our team has observed this technique in various malware such as DOORME and it should come as no surprise in this case, that flattened control flow is one of the main features within the ALCATRAZ obfuscator. In order to approach un-flattening, we focused on established tooling by using IDA plugin D810 written by security researcher Boris Batteux.

We will start with our previous example program using the common `_security_init_cookie` function used to detect buffer overflows. Below is the control flow diagram of the cookie initialization function in non-obfuscated form. Based on the graph, we can see there are six basic blocks, two conditional branches, and we can easily follow the execution flow.



Control flow of non-obfuscated security_init_cookie function

If we take the same function and apply ALCATRAZ's control flow flattening feature, the program's control flow looks vastly different with 22 basic blocks, 8 conditional branches, and a new dispatcher. In the figure below, the color-filled blocks represent the previous basic blocks from the non-obfuscated version, the remaining blocks in white represent added obfuscator code used for dispatching and controlling the execution.



Obfuscated control flow of security_init_cookie function

If we take a look at the decompilation, we can see the function is now broken into different parts within a `while` loop where a new `state` variable is used to guide the program along with remnants from the obfuscation including `popf/pushf` instructions.

Pseudocode-A

```
12  __asm { pushf }
13  state_value = 0;
14  while ( 1 )
15  {
16    while ( 1 )
17    {
18      if ( state_value == 4 )
19      {
20        __asm { popf }
21        goto LABEL_16;
22      }
23      if ( state_value )
24        break;
25      __asm { popf }
26      _CF = _security_cookie < 0x2B992DDFA232LL;
27      _OF = __OFSUB__(_security_cookie, 0x2B992DDFA232LL);
28      _ZF = _security_cookie == 0x2B992DDFA232LL;
29      _SF = (__int64)(_security_cookie - 0x2B992DDFA232LL) < 0;
30      if ( _security_cookie == 0x2B992DDFA232LL )
31      {
32        __asm { pushf }
33        state_value = 2;
34      }
35      else
36      {
37        __asm { pushf }
38        state_value = 1;
39      }
40    }
41    if ( state_value == 1 )
42    {
43      __asm { popf }
44      result = ~_security_cookie;
45      qword_14001C040 = ~_security_cookie;
46      return result;
47    }
```

Obfuscated decompilation of security_init_cookie function

For cleaning this function, D810 applies two different rules (`UnflattenerFakeJump` , `FixPredecessorOfConditionalJumpBlock`) that apply microcode transformations to improve decompilation.

```
 2025-04-03 15:44:50,182 - D810 - INFO - Starting decompilation of function at 0x140025098
2025-04-03 15:44:50,334 - D810 - INFO - glbopt finished for function at 0x140025098
2025-04-03 15:44:50,334 - D810 - INFO - BlkRule 'UnflattenerFakeJump' has been used 1 times for a total of 3 patches
2025-04-03 15:44:50,334 - D810 - INFO - BlkRule 'FixPredecessorOfConditionalJumpBlock' has been used 1 times for a total
```

When we refresh the decompiler, the control-flow flattening is removed, and the pseudocode is cleaned up.

```
uintptr_t security_init_cookie()
{
  uintptr_t result; // rax
  uintptr_t entropy; // [rsp-18h] [rbp-18h]

  if ( _security_cookie == 0x2B992DDFA232LL )
  {
    entropy = _get_entropy();
    if ( entropy == 0x2B992DDFA232LL )
      entropy = 0x2B992DDFA233LL;
    _security_cookie = entropy;
    qword_14001C040 = ~entropy;
    return ~entropy;
  }
  else
  {
    result = ~_security_cookie;
    qword_14001C040 = ~_security_cookie;
  }
  return result;
}
```

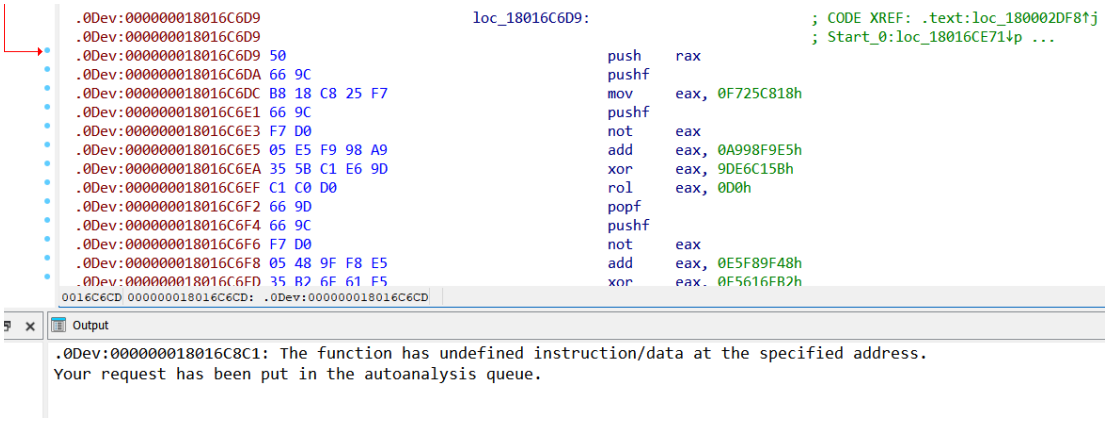
Control-flow obfuscation removed from decompilation by D810

While this is a good example, fixing control-flow obfuscation can often be a manual and timely process that is function-dependent. In the next section, we will gather up some of the techniques we learned and apply it to DOUBLELOADER.

Cleaning a DOUBLELOADER function

One of the challenges when dealing with obfuscation in malware is not so much the individual obfuscation techniques, but when the techniques are layered. Additionally, in the case of DOUBLELOADER, large portions of code are placed in function chunks with ambiguous boundaries, making it challenging to analyze. In this section, we will go through a practical example showing the cleaning process for a DOUBLELOADER function protected by ALCATRAZ.

Upon launch at the `Start` export, one of the first calls goes to `loc_18016C6D9`. This appears to be an entry to a larger function, however IDA is not properly able to create a function due to undefined instructions at `0x18016C8C1`.



Example of DoubleLoader causing error in IDA Pro

If we scroll to this address, we can see the first disruption is due to the short jump anti-disassembly technique which we saw earlier in the blog post (`EB FF`).

```
.0Dev:000000018016C8C0                          loc_18016C8C0:                          ; CODE XREF: .0Dev:loc_18016C8C0↑j
.0Dev:000000018016C8C0 EB FF                                    jmp     short near ptr loc_18016C8C0+1
.0Dev:000000018016C8C0                          ; ---------------------------------------------------------------------------
.0Dev:000000018016C8C2 D0 48 8B D8 48 C7                        dw 48D0h, 0D88Bh, 0C748h
.0Dev:000000018016C8C8 44 24 60 50 00           aDP             db 'D$`P',0
.0Dev:000000018016C8CD 00 00 33                                 db 2 dup(0), 33h
.0Dev:000000018016C8D0 F6 48 8D 05 03 48 B9 6C…                 dq 6CB94803058D48F6h, 6CCFDFFB2D489C66h, 4868244489489D66h
.0Dev:000000018016C8E8 89 74 24 70 48 89 5C 24…                 dq 245C894870247489h, 9C66E3A85E52BF78h, 0FCB4C25CC781D7F7h
.0Dev:000000018016C900 81 F7 AE E1 0A CA C1 C7…                 dq 0C7C1CA0AE1AEF781h, 81D7F79C669D666Dh, 0BAF781B4938B1AC7h
.0Dev:000000018016C918 F9 70 B5 C1 C7 93 66 9D…                 dq 9D6693C7C1B570F9h, 4A5FC781D7F79C66h, 0A68594EAF781A2A0h
.0Dev:000000018016C930 C1 C7 AE 66 9D 8B D7 33…                 dq 33D78B9D66AEC7C1h, 0FFEE59F815FFEBC9h, 0C933D78B80458948h
.0Dev:000000018016C948 EB FF 15 F1 59 EE FF 48…                 dq 48FFEE59F115FFEBh, 55056F0F66884589h, 90457F0F66FFEFA9h
.0Dev:000000018016C960 4C 8D 3D 9C 24 9E 46 66…                 dq 66469E249C3D8D4Ch, 46AE9CDBEF81499Ch, 0D78BA07D894C9D66h
.0Dev:000000018016C978 33 C9 EB FF 15 B7 59 EE…                 dq 0EE59B715FFEBC933h, 4C8D48A8458948FFh, 0EE598715FFEB6024h
.0Dev:000000018016C990 FF 8D 7E 30 66 85 C0 75…                 dq 75C08566307E8DFFh, 0D1F65CB89C665033h, 1D3205D0F79C6697h
.0Dev:000000018016C9A8 92 BE 35 16 51 91 81 C1…                 dq 0C18191511635BE92h, 0D0F79C669D660DC0h, 0CB0635E8C937F105h
.0Dev:000000018016C9C0 D0 B9 C1 C0 15 66 9D E9…                 dq 0E99D6615C0C1B9D0h, 0B89C6650FFFFFD3Bh, 0D0F79C661447F9A5h
.0Dev:000000018016C9D8 05 1B D4 B4 C8 35 9C D5…                 dq 0D59C35C8B4D41B05h, 669D6628C0C1BCFCh, 0C3B6677905D0F79Ch
.0Dev:000000018016C9F0 35 49 97 DC B9 C1 C0 10…                 dq 10C0C1B9DC974935h, 0A005D0F79C669D66h, 89283F25359261C9h
.0Dev:000000018016CA08 C1 C0 C4 66 9D E9 F5 FC                  dq 0FCF5E99D66C4C0C1h
.0Dev:000000018016CA10 FF FF                                    db 2 dup(0FFh)
```

Anti-disassembly technique in DoubleLoader

After fixing 6 nearby occurrences of this same technique, we can go back to the start address (`0x18016C6D9`) and use the MakeFunction feature. While the function will decompile, it is still heavily obfuscated which is not ideal for any analysis.

```
__int64 __fastcall sub_18016C6D9(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
  LRESULT v8; // rax
  __int64 v9; // rbx
  _QWORD *v10; // rbp
  __int64 v11; // rdi
  __int64 v12; // rsi
  __int64 v13; // r14
  const WCHAR *v14; // r15
  int v15; // eax
  __int64 v17; // rdx
  __int64 v18; // rax
  __int64 (__fastcall *v19)(_QWORD); // rax
  unsigned int v20; // edi
  signed __int16 v21; // ax
  __int64 v26; // rax
  __int64 (__fastcall *v27)(_QWORD, __int64, __int64, _QWORD); // rax
  __int64 v28; // rax
  void (__fastcall *v29)(_QWORD, __int64, __int64, _QWORD); // rax
  LRESULT v30; // rax
  _QWORD v36[3]; // [rsp-18h] [rbp-150h] BYREF
  WNDCLASSEXW v37; // [rsp+0h] [rbp-138h]
  _QWORD v38[4]; // [rsp+68h] [rbp-D0h] BYREF
  _QWORD v39[18]; // [rsp+88h] [rbp-B0h] BYREF
  LRESULT v40; // [rsp+118h] [rbp-20h]
  __int64 v41; // [rsp+128h] [rbp-10h]
  __int64 v42; // [rsp+130h] [rbp-8h]
  __int64 retaddr; // [rsp+138h] [rbp+0h]

  v40 = v8;
  __asm
  {
    pushf
    pushf
  }
  v15 = __ROL4__((-__ROL4__(798355607, 208) - 436691129) ^ 0xF5616FB2, 142);
  __asm { popf }
  while ( 1 )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        while ( v15 == 4 )
        {
          __asm { popf }
          TranslateMessage((const MSG *)(v10 - 10));
          v40 = DispatchMessageW((const MSG *)(v10 - 10));
          __asm
          {
            pushf
0016C6D9 sub_18016C6D9:16 (18016C6D9)
```

DoubleLoader function with ALCATRAZ obfuscation

Going back to the disassembly, we can see the LEA obfuscation technique used in this function below where the string constant `"Error"` is now recovered using the earlier solution.

```
.0Dev:000000018016CB8F                          loc_18016CB8F:                          ; CODE XREF: sub_18016C6D9+56↑j
.0Dev:000000018016CB8F E8 CC A4 E9 FF                           call    sub_180007060
.0Dev:000000018016CB94 4C 8B 00                                 mov     r8, [rax]
.0Dev:000000018016CB97 41 8B 50 10                              mov     edx, [r8+10h]
.0Dev:000000018016CB9B 45 8B 80 18 03 00 00                     mov     r8d, [r8+318h]
.0Dev:000000018016CBA2 48 8B C8                                 mov     rcx, rax
.0Dev:000000018016CBA5 E8 16 A5 E9 FF                           call    sub_1800070C0
.0Dev:000000018016CBAA 44 8B CF                                 mov     r9d, edi
.0Dev:000000018016CBAD 4C 8D 05 02 D9 9A 61                     lea     r8, cs:1E1B1A4B6h
.0Dev:000000018016CBB4 66 9C                                    pushf
.0Dev:000000018016CBB6 49 81 E8 D6 56 AB 61                     sub     r8, 61AB56D6h
.0Dev:000000018016CBBD 66 9D                                    popf
.0Dev:000000018016CBBF 48 8D 15 8B AC CD 40                     lea     rdx, cs:1C0E47851h
.0Dev:000000018016CBC6 66 9C                                    pushf
.0Dev:000000018016CBC8 48 81 EA 09 2A DE 40                     sub     rdx, 40DE2A09h
.0Dev:000000018016CBCF 66 9D                                    popf
.0Dev:000000018016CBD1 33 C9                                    xor     ecx, ecx
.0Dev:000000018016CBD3 90                                       nop
.0Dev:000000018016CBD4 FF D0                                    call    rax
.0Dev:000000018016CBD6 E9 A2 00 00 00                           jmp     loc_18016CC7D
.0Dev:000000018016CBDB                          ; ---------------------------------------------------------------------------
.0Dev:000000018016CBDB 50                                       push    rax
.0Dev:000000018016CBDC 66 9C                                    pushf
```

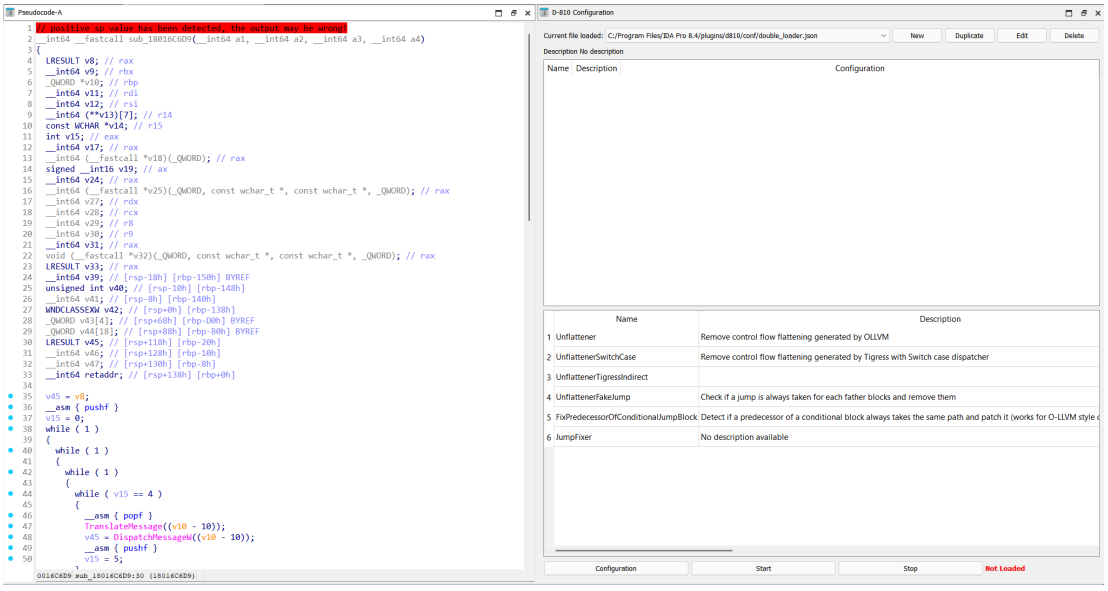
Restoring string constant from LEA obfuscation

Another example below shows the transformation of an obfuscated parameter for a `LoadIcon` call where the `lpIconName` parameter gets cleaned to `0x7f00` (`IDI_APPLICATION`).

```
.0Dev:000000018016C8E1 90                               nop
.0Dev:000000018016C8E2 90                               nop
.0Dev:000000018016C8E3 89 44 24 68                      mov     dword ptr [rsp+198h+var_138.lpfnWndProc], eax
.0Dev:000000018016C8E7 48 89 74 24 70                   mov     qword ptr [rsp+198h+var_138.cbClsExtra], rsi
.0Dev:000000018016C8EC 48 89 5C 24 78                   mov     [rsp+198h+var_138.hInstance], rbx
.0Dev:000000018016C8F1 BF 52 5E A8 E3                   mov     edi, 0E3A85E52h
.0Dev:000000018016C8F6 66 9C                            pushf
.0Dev:000000018016C8F8 F7 D7                            not     edi
.0Dev:000000018016C8FA 81 C7 5C C2 B4 FC                add     edi, 0FCB4C25Ch
.0Dev:000000018016C900 81 F7 AE E1 0A CA                xor     edi, 0CA0AE1AEh
.0Dev:000000018016C906 C1 C7 6D                         rol     edi, 6Dh
.0Dev:000000018016C909 66 9D                            popf
.0Dev:000000018016C90B 66 9C                            pushf
.0Dev:000000018016C90D F7 D7                            not     edi
.0Dev:000000018016C90F 81 C7 1A 8B 93 B4                add     edi, 0B4938B1Ah
.0Dev:000000018016C915 81 F7 BA F9 70 B5                xor     edi, 0B570F9BAh
.0Dev:000000018016C91B C1 C7 93                         rol     edi, 93h
.0Dev:000000018016C91E 66 9D                            popf
.0Dev:000000018016C920 66 9C                            pushf
.0Dev:000000018016C922 F7 D7                            not     edi
.0Dev:000000018016C924 81 C7 5F 4A A0 A2                add     edi, 0A2A04A5Fh
.0Dev:000000018016C92A 81 F7 EA 94 85 A6                xor     edi, 0A68594EAh
.0Dev:000000018016C930 C1 C7 AE                         rol     edi, 0AEh
.0Dev:000000018016C933 66 9D                            popf
.0Dev:000000018016C935 8B D7                            mov     edx, edi        ; lpIconName
.0Dev:000000018016C937 33 C9                            xor     ecx, ecx        ; hInstance
.0Dev:000000018016C939 90                               nop
.0Dev:000000018016C93A FF 15 F8 59 EE FF                call    cs:LoadIconW
```

Restoring LoadIcon parameter from immediate mov obfuscation

Now that the decompilation has improved, we can finalize the cleanup by removing control flow obfuscation with the D810 plugin. Below is a demonstration showing the before and after effects.



Decompilation cleanup of DoubleLoader function using D810

This section has covered a real-world scenario of working towards cleaning a malicious obfuscated function protected by ALCATRAZ. While malware analysis reports often show the final outcomes, a good portion of time is often spent up-front working towards removing obfuscation and fixing up the binary so it can then be properly analyzed.

IDA Python Scripts

Our team is releasing a series of proof-of-concept IDA Python scripts used to handle the default obfuscation techniques imposed by the ALCATRAZ obfuscator. These are meant to serve as basic examples when dealing with these techniques, and should be used for research purposes. Unfortunately, there is no silver bullet when dealing with obfuscation, but having some examples and general strategies can be valuable for tackling similar challenges in the future.

YARA

Elastic Security has created YARA rules to identify this activity.

- Windows.Trojan.DoubleLoader

Observations

The following observables were discussed in this research.

Observable	Type	Name	Reference
`3050c464360ba7004d60f3ea7ebdf85d9a778d931fbf1041fa5867b930e1f7fd`	SHA256	`DoubleLo.dll`	DOUBLELOADER

References

The following were referenced throughout the above research:

- https://github.com/weak1337/Alcatraz
- https://gitlab.com/eshard/d810
- https://eshard.com/posts/d810-deobfuscation-ida-pro
- http://keowu.re/posts/Analyzing-Mutation-Coded-VM-Protect-and-Alcatraz-English/
