De- obfuscating ALCATRAZ

Subscribe

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.

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.

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`.

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.

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.

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.

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.

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.

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.

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.

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.

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`).

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.

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.

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`.

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.

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.

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`).

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`.

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:

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

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.

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.

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.

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.

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 of 2 patches
```

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

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`.

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`).

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.

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.

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

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.

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: