# How to modify Mach-O binary for Obfuscation and Hooking

Anh Khoa Nguyen
*khoana@verichains.io*
*Verichains*

Thien Nhan Nguyen
*Verichains*

## Abstract

To be written

## 1 Introduction

To be written
The remainder of this paper is structured as follows. Section 2 offers an in-depth exploration of the background, encompassing obfuscation techniques, binary analysis, an overview of Apple's loader, and, more specifically, the Mach-O binary format. In Section 4, we provide a detailed account of the implementation process to obfuscate Mach-O binaries. These steps include binary modifications and restoration of essential information during runtime. We also address the nuances of obfuscating Objective-C compiled binaries and introduce additional information that can be leveraged to enhance obfuscation, along with any associated drawbacks. Finally, in Section **??**, we offer our concluding remarks.

## 2 Background

### 2.1 Binary Obfuscation

Binary executable files encapsulate assembly instructions, program data, and essential execution information for the operating system to execute. Binary obfuscation aims to eliminate critical information that reverse engineers rely on for their analysis. Nevertheless, it is essential to note that binary obfuscation is inherently platform-specific. The predominant focus has been on Windows binaries (commercialized) [8, 12, 16] and Linux binaries [6, 9, 10, 14, 17], while Apple binaries have received comparatively limited attention. Some previous work [4, 13, 17] has proposed modest adjustments to debugging information within binaries, which helps to impede the analysis capabilities of reverse engineering platforms.
Another widely recognized form of binary obfuscation is known as "packing" [11]. This method involves compressing or encrypting the binary code and then unpacking it at runtime. However, it is important to note that because the code is unpacked during runtime, it remains susceptible to memory extraction, which could potentially allow an attacker to recover the original code.

### 2.2 Apple's loader `dyld`

The Apple loader [1], known as `dyld`, is responsible for the execution of programs, including the loading of the binary and its associated libraries, the resolution of dynamic symbols, the rebasing of offsets, and the final execution of the binary. Due to the shared cache mechanism [7] introduced in iOS 13.5 or macOS 11.0, important libraries, including the system (often referred to as `libSystem` [2]), C++, Objective-C Runtime [3], Foundation, and Swift Runtime libraries, are loaded into memory and are only available there. In older versions of Apple's operating systems, direct file system access to the `dyld` loader was possible. However, in recent versions, such access is no longer feasible as `dyld` and these libraries are now exclusively in memory since the system boots.

### 2.3 Mach-O binary format

The Mach-O binary format is inherently complex. To gain a comprehensive understanding of our proposed obfuscation technique, it is imperative that we closely examine this binary format. It is crucial to emphasize that our obfuscation methodology does not pertain to the obfuscation of the binary code itself. Instead, our focus lies on the obfuscation of vital information stored within the binary file. Therefore, a thorough understanding of how the Mach-O binary format stores this information is important in understanding our approach.

#### 2.3.1 Basic Mach-O structure

The Mach-O binary format can be comprehensively examined from multiple perspectives. One fundamental approach is to dissect it on the basis of its encoding of binary data. In this sense, a Mach-O binary comprises a header, a sequence of

load commands, and subsequent raw binary data. The header provides essential information about the binary, encompassing its type (whether executable or library), endianness, architecture, and the number of load commands. Load commands, crucial for the loader's runtime operations, facilitate the mapping of the binary into memory and the execution of preliminary tasks. Some load commands reference the raw binary data.
Alternatively, another perspective to comprehend the Mach-O binary is through its segmentation. Typically, the binary consists of three key segments: `_TEXT` contains assembly instructions; `_DATA` and `_DATA_CONST` store static binary data; `_LINK_EDIT` segment is dedicated to loader instructions.

### 2.3.2 Dynamic library load chain

In most cases, programs cannot function as standalone entities but rely on dynamic libraries. These libraries are registered in the header of Mach-O binaries using commands such as `LC_DYLIB` (or similar equivalents). These commands establish a load chain, organized in a specific order, and recursively link each item. In addition to these explicit commands, the loader also dynamically loads libraries that are essential for the binary's runtime operation. These may include standard libraries, Foundation libraries, as well as Objective-C and Swift libraries, among others.
The loader is responsible for locating and loading libraries into memory. These libraries fall into different categories: system installed libraries and user-provided libraries are identified by their names within the `LC_DYLIB` load command. These names can represent full or relative paths. Full paths are self-explanatory, whereas relative paths can be more intricate, involving file system-relative paths or the use of *rpath* variables. There are three rpath variables: `@executable_path`, `@loader_path`, and `@rpath`. They serve as references to libraries, with `@executable_path` pointing to the location of the executable, `@loader_path` indicating the loader's location, and `@rpath` being defined through a series of `LC_RPATH` commands. Libraries using the `@rpath` reference will be iteratively replaced through each item in the `LC_RPATH` chain to search for the corresponding file on the disk.

### 2.3.3 Dynamic Symbols

Functions from external libraries are often used as a means of code reuse. When a binary does not statically link with a library, it must specify the required library and functions statically in its binary format and will be resolved at runtime. This approach to code reuse is known as dynamic loading. In Mach-O binaries, all the information necessary for dynamic loading, usually referred to as import table, is spread across various segments, including `_LINK_EDIT`, `_DATA`, and `_DATA_CONST`. The import table in Mach-O has undergone several updates over time. The original version of the import table used a custom bytecode chain, while the updated version introduced in iOS 14 employs fix-ups chains.
During both load time and runtime, the loader of a Mach-O binary reads the import table, searches for the addresses of symbols, and rewrites them in memory for reference by the executable or library code. To facilitate this functionality, the binary allocates space for a list of stubs. These stubs serve as templates and serve as branching targets. When these stubs are resolved by the loader, the target functions become known, allowing calls to dynamic library functions as shown in Table 1.

### 2.3.4 Rebase

In binary files, references (pointers) to other data are often stored as file offsets. During execution, when the binary is loaded into memory at a specific address range, these references need to be adjusted from relative (offset in the file) to absolute addresses. This process is called rebasing, where pointers are rebased from 0 to the loaded address. Readers might be familiar with Position-Independent Code, and the rebase is the design for this mechanism in Mach-O binaries. While there is no specific term for the list of pointers to be rebased at runtime, for the sake of brevity, we can refer to these as the "rebase table".

### 2.3.5 Bytecode chain

In the original design of Mach-O binaries, the import table and the rebase table were implemented using bytecode chains. These chains embody the basic form of a state machine instruction. This bytecode has a special opcode `BIND_OPCODE_DO_BIND` to determine where a state defines a symbol, or opcodes with prefix `REBASE_OPCODE_DO_REBASE` to define a rebase pointer. This approach optimizes storage by specifying only changes between multiple items.
In this design, there are four different chains, each serving distinct purposes: Rebase, Non-Lazy, Lazy, and Weak. The Rebase chain is the rebase table. Non-Lazy, Lazy, and Weak chains are used for dynamic symbol resolution, but operate at different stages of the binary execution. Non-Lazy symbols must be resolved during the load time, while lazy symbols can be resolved when first called. Weak symbols are used to avoid collision in the symbol name.
Lazy symbols are resolved through an indirect call to the loader, which subsequently reads the bytecode chain to extract a single symbol and writes back the function address. This process is executed via a procedure in `dyld` known as `dyld_stub_binder`. An overview of this type of resolution is given in Table 2.

### 2.3.6 Chained fixups

In later versions of the Mach-O binary format, performance optimization led to the deprecation of bytecode chains in favor of fixups chains. Unlike bytecode chains, fixups chains

do not separate between rebasing and dynamic symbol resolution; instead, they are processed together. This approach significantly enhances overall performance by reducing the number of runs through the binary.
In this design, there exist sequences of contiguous 8-byte values. Each 8-byte unit incorporates a single bit to signify whether it is intended to serve as a rebase pointer or to represent a dynamic symbol. For rebase pointers, the unused bits are repurposed to specify the readdressing mechanism, while dynamic symbols utilize the remaining bits to encode both the index within the library list and the index within the string table corresponding to the symbol name. The 8-byte values are modified in place when rebased or resolved during load time.

#### 2.3.7 Export trie

In Mach-O binaries, dynamic symbols that are meant to be discovered during dynamic symbol resolution are stored in an export trie. This data structure resembles a prefix trie and derives its name from this resemblance. The essential characteristic of an export trie is that all items share a common root, which requires that all symbols be prefixed with an underscore.

#### 2.3.8 Fat binary

A fat binary is a common type of executable binary used in Apple devices. It functions as a wrapper for a multi-architecture executable containing different architectures of Mach-O binaries of the same program. When submitting applications to Apple, a fat binary is typically required. However, when a user downloads the application to a specific device, only the Mach-O binary with the corresponding architecture for that device is actually downloaded and used. This approach ensures compatibility with various Apple devices while optimizing the download size for each specific target.

## 3 Related Works

In this section, we survey the existing open-source solutions for Mach-O binary obfuscation, focusing specifically on methods that take a binary as input and produce an obfuscated output. We exclude approaches that involve decompilation followed by obfuscation on the decompiled code from our consideration; packers are also excluded due to them being a different kind of obfuscation. This enumeration aims to provide an overview of the current landscape of Mach-O binary obfuscation techniques that adhere to the specified criteria.
We first go through a list of common obfuscation techniques on Mach-O binaries; these techniques are well known and often suggested, as well as supported, by many. These techniques often involve the removal of exported symbols. The exported symbols are not mandatory in the main executable because the binary entry point is accessible through the `LC_MAIN` load command and other symbols are not referenced by other libraries. Removing the list of exported symbols (and sometimes, public symbols) can be easily performed through compiler / linker argument invocation or directly remove the associated load commands, `LC_SYMTAB` for instance.
Unused sections and data within the binary are also targeted for removal in certain obfuscation methods. This process is relatively straightforward since these portions of the program are deemed unnecessary for execution.
Numerous obfuscation methods focus on renaming Objective-C symbols, as evident in tools like MachObfuscator [4] and ios-class-guard [13]. The underlying rationale is straightforward: Renaming Objective-C class names and methods to strings of equal length, often employing random or generated names. This intentional obfuscation adds complexity for reverse engineers, requiring more effort to decipher the meaning of classes without the aid of descriptive names.
The Poor Man's Obfuscator [17] employs a more intricate obfuscation scheme. In this approach, various load commands are altered to feed incorrect information to binary analysis platforms. Obfuscation options, called transformations, include actions such as randomizing the names of exported symbols, redirecting the addresses of exported symbols to different locations, adjusting the offsets and sizes of sections in load commands, and modifying entries in the `LC_FUNCTION_START` table. Upon scrutiny, this obfuscation scheme introduces moderate disruptions that challenge many binary analysis platforms.

## 4 Implementation

In this section, we present an in-depth exploration of our obfuscation technique, tailored specifically for Mach-O binaries.

## 4.1 Design Overview

The primary objective is to manipulate the load-time data embedded within the binary. By modifying these critical pieces of information, we render the binary incapable of successful loading into memory. These information elements frequently serve as crucial input for static analysis tools, such as IDA or Ghidra. The removal of this information creates an environment of partial knowledge, making it harder for analysts to reverse the binary. A similar obfuscation concept was introduced in a previous work for Windows PE binaries [8].
We introduce a control-flow intervention between the binary loading process and its execution. This is necessary because the loader cannot perform a full load of the binary due to missing information that results in a crash during execution. To keep the binary working as normal, the intervened code performs a part of the loader's workflow using the extracted information. This process is later termed *restoration logic*.

This intervening control flow is inserted through an external library or is injected, as described in [**?**].
In this paper, we use an external library to perform restoration logic. The obfuscated binary is stripped of unnecessary information, and the information needed for the loading process is extracted, then included in the external library. The binary is also added with a load command to load our external library. As an optional measure, the extracted data can undergo static encryption, with decryption occurring at load time when our restoration logic is executed. It is worth noting that our restoration logic may rely on functions from the loader, potentially exposing the runtime restoration process. To enhance the resilience of this restoration method, we offer a mechanism to conceal the invocation of these functions by jumping to the direct address.

## 4.2 Extracting information

Because our obfuscation alters the loading process, the information used by the loader is considered for extraction. This information is typically stored in `_LINK_EDIT` segment. Commands that use these segments can be removed if they are not necessary. Our obfuscation chose to extract the information in `LC_DYLD_INFO_ONLY`, `LC_CHAINED_FIXUPS`. In addition, we also extract the list of constructor function pointers; these are often called before the binary's main procedure.
**`LC_DYLD_INFO_ONLY`** load command provides information in the form of bytecode chains. If the binary has this load command, we eliminate the lazy and weak bytecode chains by rewrite the data so that the loader would skip through. This is accomplished by configuring the bytecode chain size to a value of 0 within the load command and subsequently overwriting the bytecode chain section within the `_LINK_EDIT` segment with random values.
**`LC_CHAINED_FIXUPS`** load command offers a sequence of fixups chains. By traversing these chains, we can extract all the imported symbols. These symbols are typically stored as indices pointing to an indexed store of strings, where each index corresponds to the symbol's name and the hosting library. To exclude these symbols from the loading process, we undertake a two-fold process. First, we rewrite each chain dynamic symbol values to a rebase values. Subsequently, we completely eliminate the string table that holds the symbol names.
In both situations, the loader should proceed without experiencing a crash. If the table is empty in the case of bytecode chains, the loader can skip reading the dynamic symbols. Similarly, in the case of chained fixups, they can be resolved as a rebase value, ensuring a seamless continuation of the loading process.
**Constructor functions** refer to pointers that are invoked by the loader once all images have been loaded into memory. These functions are called sequentially and recursively (into each loaded library) as part of the initialization process. To remove these functions, several methods are available. One straightforward approach is to modify the `LC_SECTION` flag to exclude the section from being listed as constructor pointers. Additionally, pointers are typically checked to ensure that they reside within the binary's memory region. When adjusting these pointers to point outside the memory region, the loader will disregard them, effectively achieving removal of these functions.

## 4.3 Removing redundant information

Some data are automatically generated during the compilation process by the compiler and the linker. This information serves no inherent purpose during runtime and, as such, can be removed. Examples of such debugging-related data are defined in commands like `LC_SYMTAB`, `LC_DYSYMTAB`, `LC_FUNCTION_STARTS`, and `LC_DATA_IN_CODE`, among others. Their exclusion from the binary file does not compromise its functionality during execution, but rather streamlines the binary by eliminating superfluous debugging-related content. The complete removal of this information effectively prevents a basic analysis that relies on these debug symbols to make sense of the binary program.
Depending on the nature of the load command and its functionality, it may be considered for removal. Load commands that fall under the category of informative or debugging data are typically candidates for removal, provided that their absence does not disrupt the overall load process or while running.
Our technique also allows for the removal of load commands related to system libraries. These libraries are always present in memory and can be accessed by any process. The inclusion of these load commands in the binary is only necessary for referencing dynamic symbols. However, since our obfuscation method extracts all dynamic symbols from the binary, the references to system libraries can also be eliminated.

## 4.4 Restoration

During runtime, the retrieval of extracted information, which is crucial for the complete loading of the binary, occurs. We employ a constructor function that is scheduled to run before the main executable. By utilizing the entire set of parameters provided by the loader, we are able to determine the base address of the main executable that has been loaded into memory, as depicted in Listing 2.
Having obtained the base address of the main executable, we can proceed with the restoration process by simulating the loader's actions. For each piece of extracted information, we execute the corresponding restoration procedure in accordance with its specific logic.
The load commands **`LC_DYLD_INFO_ONLY`** and **`LC_CHAINED_FIXUPS`** have different representations, but both contain a collection of dynamic symbols. Each symbol

in the collection includes the symbol name, the exporting library, and the address where the function pointer is stored. By extracting data from these load commands, we generate a dynamic symbol list. During runtime, we iterate through this list to locate the symbol and update the function pointer. The symbol can be easily found using `dlsym`. Writing to the function pointer requires that the address be writable, as described in Listing 1, because the loader locks the `__DATA_CONST` segment as read-only after it finishes and our restoration logic performs after the loading process.

```
#include <mach/mach.h>
vm_protect(mach_task_self(), offset, size, 0,
           VM_PROT_READ | VM_PROT_WRITE);
```

Listing 1: Modify the virtual memory range from `offset` to `offset+size` to Read-Write.

**Constructor functions** can be invoked directly. We can calculate the function addresses and invoke them with parameters passed to our constructor because the loader consistently passes the same arguments to all constructors during initialization, enabling us to call these functions manually without reliance on the loader.
The aforementioned mechanism serves as the fundamental concept for obfuscating a Mach-O binary. In the process, we have incorporated the utilization of an external library. In order to achieve a high level of obfuscation, we employ the same obfuscation techniques to obscure the external library and conceal the restoration logic. However, this approach presents a unique challenge as the library now needs to restore itself without passing through the loader. Further details regarding this issue can be found in Appendix A.

## 4.5 Objective-C compiled binary

The previously described restoration logic is highly effective when applied to binaries compiled from C or C++. However, within the Apple ecosystem, Objective-C is a predominant language for application development. Objective-C is a unique component of Apple's technology stack and is seamlessly integrated into the loading process of executables through custom passes. Consequently, addressing the challenges associated with Objective-C compiled binaries requires a distinct approach. Before delving into these nuances, it is essential to clarify the synergy between Objective-C and the `dyld` loader.

### 4.5.1 Relationship with `dyld`

The Objective-C runtime is consistently loaded into memory and automatically mapped to the same virtual memory space as the executable. Within this runtime, a collection of hooks is made available and these hooks are strategically used by the `dyld` at various stages of the binary loading and unloading processes. During the initialization of the Objective-C runtime via `libSystem`, an array of callbacks is supplied to `dyld`.
By the callback structure `_dyld_objc_callbacks_v1`, the Objective-C runtime registers three functions at different stages of the binary loading and unloading processes: when the binary is mapped into memory, when the binary is invoked to call constructors, and when the binary is subsequently unmapped from memory.

### 4.5.2 Objective-C data in binary

The binaries compiled from Objective-C include sections identified by the prefix `_objc`. These sections are integral to the functioning of the Objective-C runtime, facilitating the initialization of Objective-C classes and selectors. In summary, Objective-C runtime performs the initialization of class objects and selectors when the binary is mapped to memory, and Objective-C `+load` methods are called during constructor invocation.
Objective-C binaries contain class definitions represented as data. These classes are defined using two special pointers: `isa` and `superclass`. The `isa` represents the metaclass, while the `superclass` represents the parent class. Each class has its own metaclass, and the `superclass` pointer points to the class data of the parent class. It is important to note that the `superclass` pointer can never be null because all classes in Objective-C must inherit from `NSObject`.

### 4.5.3 Restoration logic with Objective-C

Dynamic symbols in Objective-C, which include classes referencing other classes in different libraries, are also considered as dynamic symbols. Our obfuscation technique successfully eliminates these symbols. However, the Objective-C runtime workflow necessitates the loading of these classes. Furthermore, the Foundation library must be initialized before it can be referenced. The current restoration logic is implemented prior to any Objective-C runtime and Foundation initialization, which would lead to crashes. To address this issue, we incorporate a shellcode snippet to redirect the execution of the main function until all Objective-C classes have been resolved.
In order to achieve this, we prevent the Objective-C runtime from executing its class loading mechanism by modifying the names of two sections in the binary: `__objc_classlist` and `__objc_nlclslist`. We insert a shellcode before the start of the `_TEXT` segment and edit the `LC_MAIN` load command to points to the shellcode location.
The shellcode is created to be compact, with the primary objective of executing a function that resolves all Objective-C classes, referred to as `restore_objc`, and jumps to the main function after it finishes.
Objective-C logic for processing the class data is done through private methods like `readClass`,

realizeClassWithoutSwift, remapClass, to name a few. These symbols cannot be found in the export trie. However, they are available in the LC_SYMTAB directives. We can search for these symbols addresses and rebuild the logic as described in _read_images protocol of the Objective-C runtime.

In iOS environment, these symbols are not declared in LC_SYMTAB. However, we can try to locate these symbols indirectly through public symbols that invoke them.

The process of locating the restore_objc procedure can be complex. To keep the shellcode as compact as possible, we have opted to store the procedure's address in a location that can be easily calculated. Specifically, we have chosen the end of the _DATA segment for this purpose. At this location, the first pointer value immediately following the end of the segment represents the address of the restore_objc procedure, while the second pointer value indicates the address of the binary's main function. These pointer values are written after the restoration logic.
In practical situations, the space available for the shellcode before the _TEXT segment and the number of pointer values after the _DATA segment may be limited. Therefore, it is important to keep the shellcode as concise as possible and reduce the number of required pointer values. Generally, there should be enough space available since these segments are page-aligned, unless the code and data sizes are exact multiples of the page size, which would result in no extra space. If the available space is insufficient, it is recommended to use [?] or similar methods to add additional code in the binary for the purpose of restoration logic.

## 5 Hooking

Hooking is a prominent topic in the field of security and has evolved with tools such as Frida [15] and Fishhook [5]. Frida allows developers to inject hooks at arbitrary addresses, Fishhook allows developers to replace the body of a function at runtime after some setup procedures. However, the hooking method we have described here is distinct from Frida's approach, rather it is quite similar to Fishhook but on a binary level. Although we do not have the flexibility to hook arbitrary addresses, our method enables us to modify function invocations, directing them to our custom functions through dynamic symbol resolution. This approach is particularly valuable when we need to intercept and modify system API calls, such as file opening (e.g., fopen), or when we want to disable certain functions from being called altogether.

### 5.1 C API hooking

As explained in Section 4, our obfuscation technique resolves dynamic symbols at runtime. This provides an opportunity to intercept the C API. The concept is straightforward: The symbol that needs to be intercepted is manually resolved, and the function pointer is then overwritten by our intercepted function.
The concept is comparable to Fishhook and dyld interposing, as it involves altering the API function pointer to a different function. However, unlike Fishhook, we do not need to have access to the program's source code. Additionally, unlike dyld interposing, we manually install the hook without relying on the dyld API.

### 5.2 Objective-C class method hooking

As stated earlier in Subsection 4.5.2, the metadata for each class is stored statically in the binary. This metadata contains references to the metaclass, parent class, and a structured called class_ro. When loading the class, the Objective-C runtime reads this structured to initialize the class prototype. The prototype includes references to the class name, method list, property list, ivars, and other information.
The list of methods, called the method list, includes metadata for each method. Each metadata entry includes a selector, type, and implementation (which is a function pointer to the method). The Objective-C runtime goes through the list of methods and associates each selector with its corresponding implementation function pointer. This association is used when invoking a class method using objc_msgSend, which is the underlying mechanism for invoking Objective-C methods such as [instance method].
A possible approach to enable hooking Objective-C class methods is to modify the function pointer in a straightforward manner. This allows the Objective-C runtime to associate a different function with it. However, there is a challenge due to the fact that these method pointers are typically stored in the _TEXT segment, namely the __objc_methlist section, which is usually designated for read and execute operations only. To make this section writable, the max-protection property for the section must have the write bit set to 1. During runtime, the memory is first changed to read-write to allow modifications, and once all the necessary changes have been made, the memory is switched back to read-execute to enable execution. It is important to note that this approach may work on unverified applications, such as those distributed outside the App Store. However, it is uncertain whether the App Store policies prevent the _TEXT section from being writable. Therefore, instead of directly modifying the function pointer, an alternative approach is to modify the class_ro data to point to a different method list.
The above explanation is applicable only to classes that are defined within the binary. This is why the class_ro data is included. However, for classes that are imported, the class_ro data is not available and modifying the method list is not possible. Nonetheless, imported classes are defined as dynamic symbols, and due to the way the Objective-C runtime handles

class prototypes, hooking the methods of imported classes can be achieved through inheritance.
Objective-C is built upon the C API, and its syntax is transformed into calls to the C API. When creating an instance of a class, the underlying C API used is `objc_alloc(Class cls)`. This API requires the class metadata as an argument. In the case of imported classes, the class metadata is represented by a dynamic symbol. To implement hooking, a different class metadata is provided, which has a `superclass` pointer pointing to the original class metadata. This hooking class metadata also includes a method list that can potentially override the original class methods.
In Objective-C, class methods have a strict definition for their arguments. The first argument is always the class instance pointer, followed by the selector pointer. Any additional arguments, as pointers, are passed in when the method is invoked.

## References

[1] Apple. dyld.

[2] Apple. LibSystem.

[3] Apple. Objective-C Runtime.

[4] Kamil Borzym. MachObfuscator.

[5] Facebook. fishhook.

[6] Vector 35 Inc. Binary Ninja.

[7] iPhoneDev. dyld shared cache.

[8] Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Takeshi Yada. Stealth loader: Trace-free program loading for api obfuscation. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*, pages 217–237. Springer, 2017.

[9] Byoungyoung Lee, Yuna Kim, and Jong Kim. binob+ a framework for potent and stealthy binary obfuscation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 271–281, 2010.

[10] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[11] Markus FXJ Oberhumer. Upx the ultimate packer for executables. *http://upx. sourceforge. net/*, 2004.

[12] Oreans. Themida.

[13] Polidea. ios-class-guard.

[14] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, pages 275–290, 2007.

[15] Ole André V. Ravnås. Frida.

[16] VMProtect Software. VMProtect.

[17] Romain Thomas. The Poor Man's Obfuscator, 2022.

## A Obfuscate the restoration logic

### A.1 Manual `dlsym`

The loader's ability to resolve symbols into function addresses using either `LC_DYLD_INFO_ONLY` or `LC_CHAINED_FIXUPS` is well understood. Essentially, the loader maintains a list of loaded libraries and utilizes the export trie of each library to find public symbols based on their names. However, it is important to note that in some cases, a function may be re-exported from another library. In such situations, a recursive search through libraries is necessary to locate the address of the function. This recursive search ensures that the loader can accurately resolve symbols even when they are re-exported from different libraries. Occasionally, the re-exported symbol may also be renamed, requiring subsequent searches to use the new name.
When searching for symbols, it is crucial to consider that symbols may refer to their hosting library using relative paths. These relative paths can be expressed as either directory-relative paths or through path variables such as `@rpath`, `@executable_path`, or `@loader_path`. To ensure precise resolution of these relative paths, it is recommended to convert them into their respective full paths.
In order to obtain the list of loaded libraries in memory, a series of three symbols can be used: `_dyld_image_count`, `_dyld_get_image_header`, and `_dyld_get_image_name`. By invoking these symbols sequentially, a comprehensive list of loaded libraries can be compiled. This functionality is demonstrated in Listing 3. More specifically, `_dyld_get_image_header` provides the base address of the library at a specific index, while `_dyld_get_image_name` returns the full path of the library.

It should be noted that the file path obtained from `_dyld_get_image_name` and the library name specified in the `ID_DYLIB` load command may not match. `dyld` examines the `LC_DYLIB` load commands to determine which library to load based on the `ID_DYLIB` value.

### A.2 Obfuscate the external library

Our obfuscation method can also be applied to the external library, which is a Mach-O binary. During the obfuscation

process, all symbols are removed and need to be reinstated during runtime. As the library is known to be executed first, we can utilize this opportunity to resolve our library. However, a drawback of this approach is the restricted usage of dynamic symbols.
In order to address the issue of restricted usage of dynamic symbols and ensure the restoration of the external library, a possible solution is to employ a solitary dynamic symbol as a point of reference. This dynamic symbol can then be utilized to locate the header of the dynamic library. Following this rationale, we can locate the header of the `dyld` library by selecting any dynamic symbol of our preference, such as `dyld_get_sdk_version`.
During the extraction phase, all symbols except for `dyld_get_sdk_version` are removed. This specific symbol is used to locate the `dyld` library in memory. The extracted information is then written into a new section of the library called `_EXTRACTED` for easy access. By obtaining a reference to `dyld`, we can restore the function pointers of dynamic symbols using the information stored in `_EXTRACTED`. It is necessary to use many symbols in `dyld`, but these symbols can be found by traversing the export trie of `dyld` in memory.
By executing the aforementioned procedures, both the obfuscation binary and the restoration library become completely obfuscated. The obfuscated versions of these components reveal very few symbols, and in the case of an Objective-C compiled binary, references to classes are also eliminated.

## B Code snippets

```
foo@address:
    0x00000000

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(a)

```
foo@address:
    0xAABBCCDD

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(b)

Table 1: Assembly stubs: (a) Stub before dynamic symbol resolution and, (b) Stub after dynamic symbol resolution. `foo@address` in (a) is uninitialized, while in (b) it is given a concrete address.

```
dyld_stub_binder:
    0x11223344

foo@stub_helper:
    mov x12, foo_bytecode_offset
    mov x8, [dyld_stub_binder]
    blx x8

foo@address:
    foo@stub_helper

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(a)

```
dyld_stub_binder:
    0x11223344

foo@stub_helper:
    mov x12, foo_bytecode_offset
    mov x8, [dyld_stub_binder]
    blx x8

foo@address:
    0xAABBCCDD

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(b)

Table 2: Lazy dynamic symbols resolution: (a) Stub before lazy dynamic symbol resolution and, (b) Stub after lazy dynamic symbol resolution. `foo@address` in (a) is initialized with a `stub_helper` while in (b) it is given a concrete address. `foo_bytecode_offset` is template for the offset of `foo` symbol in the lazy bytecode chain.

```
adr x8, 0
movz x9, #0x9999
add x8, x8, x9
stp x30, x8, [sp], #-0x10
stp x3, x2, [sp], #-0x10
stp x1, x0, [sp], #-0x10
ldr x9, [x8]
blr x9
ldp x1, x0, [sp, #0x10]!
ldp x3, x2, [sp, #0x10]!
ldp x30, x8, [sp, #0x10]!
ldr x9, [x8, #8]
br x9
```

```
lea     r8,[rip+0x0]
mov     r9,0x4030201
add     r8,r9
push    rdi
push    rsi
push    rdx
push    rcx
push    r8
mov     r9,QWORD PTR [r8]
call    r9
pop     r8
pop     rcx
pop     rdx
pop     rsi
pop     rdi
mov     r9,QWORD PTR [r8+0x8]
jmp     r9
```

Table 3: Shellcode inserted: ARM64 on the left; Intel 64 on the right. The second instruction of both versions is the offset from the shellcode to the end of __DATA section. Making the pop on 2 words value of `r8` (in Intel 64) or `x8` (in ARM64) be the address of the `restore_objc` and the main function.

```
struct ProgramVars {
  void *mh; // mach_header or mach_header64
  int *NXArgcPtr;
  const char ***NXArgvPtr;
  const char ***environPtr;
  const char **__prognamePtr;
};

__attribute__((constructor)) static void
restoration(int argc, const char *const argv[], const char *const envp[],
     const char *const apple[], const struct ProgramVars *vars) {
    const void* main_binary_base = vars->mh;
    // ...
}
```

Listing 2: Using **ProgramVars** struct

```
#import <mach-o/dyld.h>
uint32_t count = _dyld_image_count();
for(uint32_t i = 0; i < count; i++) {
    const char *name = _dyld_get_image_name(i);
    const void *header = _dyld_get_image_header(i);
}
```

Listing 3: Get a list of loaded libraries

```c
const uint32_t magic64 = 0xfeedfacf;
const uint32_t magic32 = 0xfeedface;

void *find_header(void *_func) {
  const uint64_t page_size = 0x1000;
  uint64_t func = (uint64_t)_func;
  uint64_t start_searching = func + (0x1000 - (func % page_size));
  uint32_t *x = (uint32_t *)(start_searching);
  while (*x != magic64 && *x != magic32) {
    x -= 0x1000 / 4;
  }
  return (void *)x;
}
```

Listing 4: Searching for Mach-O base address