# How to Inject Code into Mach-O Apps. Part I.

jon-gabilondo-angulo-7635.medium.com/how-to-inject-code-into-mach-o-apps-part-i-17ed375f736e

Jon Gabilondo July 25, 2022

## Dynamic Code Injection Techniques

Jon Gabilondo


Calculator App with Organismo framework injected.

In this story we are going to research how to add your own code to already compiled Apps, i.e. any App you may have on your Mac.

This is an old subject and you may find many entries on the internet about it. Here we are going to avoid adding redundancy and focus in new in-depth information with hands on examples and custom made tools for immediate fun.

For those looking to add new functionalities to existing Apps, it would be better to firstly check if the App allows Plug-Ins and take that direction instead.

We are going to start using straight forward code injection methods and discover its limitations, going forward to more complicated methods to inject code in Apps that have been "hardened" to refuse code injection.

To inject "external" code to any App, it all boils down to let the dynamic linker "dyld" load your code to the same memory space as the App. Since the "dyld" is our friend, let's start by having a quick overview about it.

## The Dynamic Linker — dyld

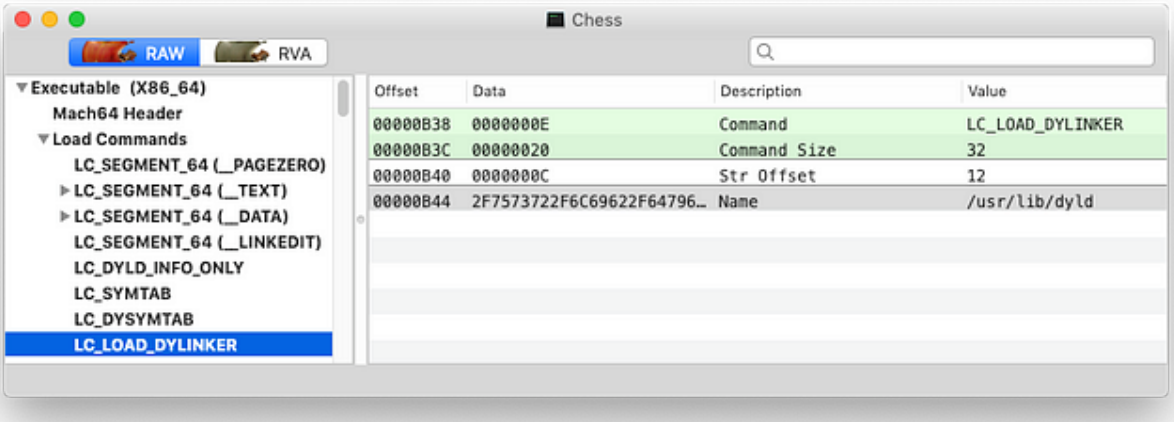
The dynamic loader or dyld is a fundamental part of any OS, without it nothing runs. In the case of OS X/iOS it is open source (!) which will give us an extraordinary opportunity to learn how it works. This is the latest code as of the date of this writing: https://opensource.apple.com/source/dyld/dyld-655.1.1/

The dyld is a vast subject with information you can find in any OS documentation and articles the web. When not, you have the code available. Here we are just going to give an overview for the sake of our purpose.

Let's see how does the dyld gets into play when we launch an application. Quoting Apple…

> When you launch an application, the system ultimately calls two functions on your behalf, `fork`and `execve`. The `fork` function creates a process; the `execve` function loads and executes the program.
>
> ….
>
> To run a different executable, your process must call the `execve` system call with a pathname specifying the location of the alternate executable (Mach-O file). The `execve` call replaces the program currently in memory with a different executable file.
>
> A Mach-O executable file contains a header consisting of a set of load commands. For programs that use shared libraries or frameworks, **one of these commands specifies the location of the linker** to be used to load the program. If you use Xcode, this is always `/usr/lib/dyld`, the standard OS X dynamic linker.

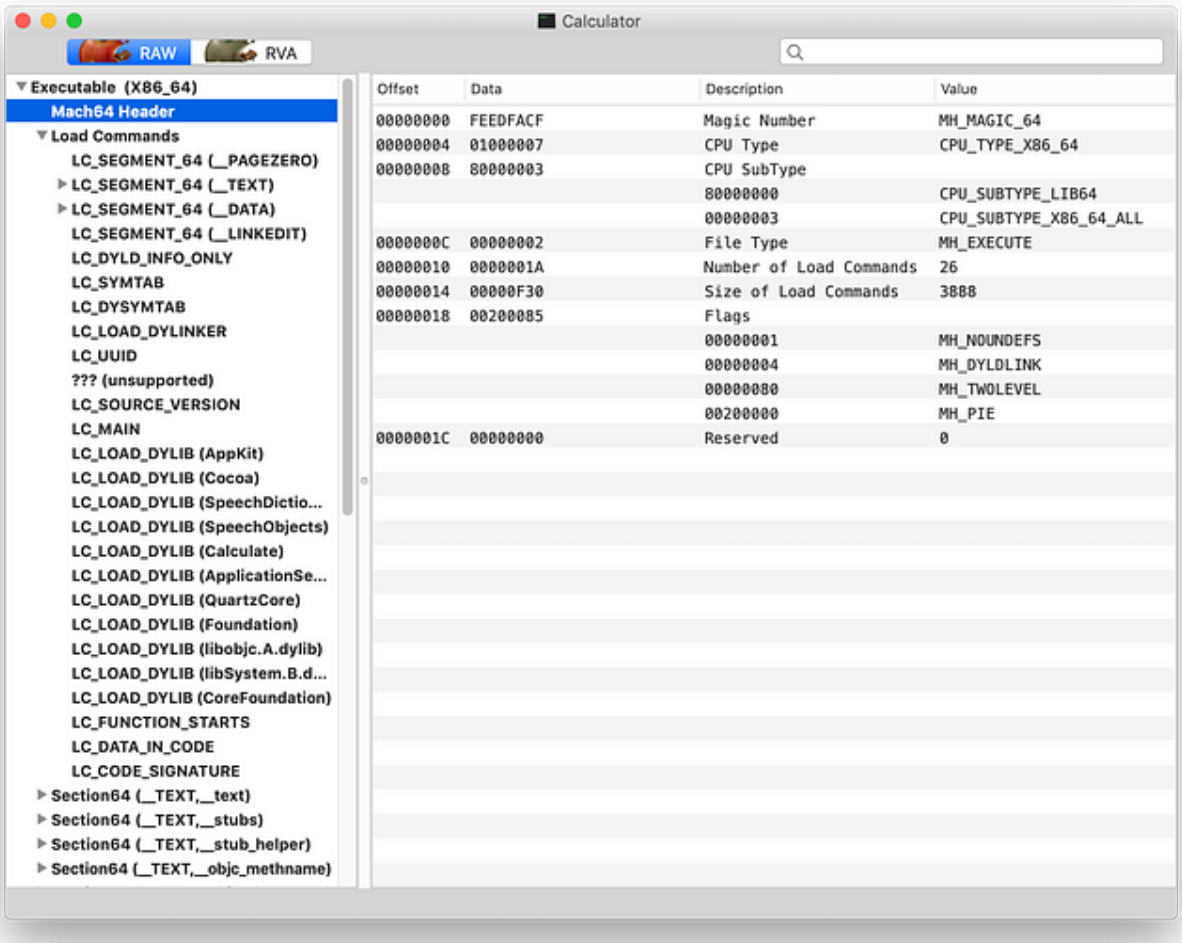
Let's check in a Mach-O file the load command that Apple refers to. Yes, it's there, in LC_LOAD_DYLINKER. It contains the path to the dyld to use */usr/lib/dyld*. This is already a very interesting insight. We could build an App that specifies another path to the dylib .. maybe to another version of the dyld, maybe one built by yourself.




The command that specifies the dynamic linker to use. (Chess App)

> When you call the `execve` routine, the kernel first loads the specified program file and examines the `mach_header` structure at the start of the file. The kernel verifies that the file appear to be a valid Mach-O file and interprets the load commands stored in the header. The kernel then loads the dynamic linker specified by the load commands into memory and executes the dynamic linker on the program file.
>
> **The dynamic linker loads all the shared libraries that the main program links against** (the *dependent libraries*) and binds enough of the symbols to start the program. It then calls the entry point function. At build time, the static linker adds the standard entry point function to the main executable file from the object file `/usr/lib/crt1.o`. This function sets up the runtime environment state for the kernel and calls static initializers for C++ objects, initializes the Objective-C runtime, and then calls the program's `main` function.

Here is a new screenshot of the mentioned Mach header and load commands. See the LC_LOAD_DYLIB commands bellow :




A Mach-O-View screenshot of the Calculator App. Unveils the mach header and the load commands.

From this overview we can extract an interesting fact in the context of our code injection goal. The executables (Mach-O files) have a list of the dylibs to load dynamically.

## System Integrity Protection (SIP) Disabled

Before going forward, just a reminder. If you are here it means you like off-road driving and probably you have SIP disabled already, if not you may want to do it to test the coming exercises.

Here some guides from Apple itself on SIP and how to disable it.

## About System Integrity Protection on your Mac

### System Integrity Protection is a security technology in OS X El Capitan and later that's designed to help prevent…

support.apple.com

## Configuring System Integrity Protection

### Describes a security feature that protects against unauthorized access to system locations and processes.

developer.apple.com

## The dyld and the Environment Variables

The dyld offers an easy interface for certain functionalities using environment variables. ‘man’ has some interesting information:

```
: If System Integrity Protection is enabled, these environment variables are ignored
when executing binaries protected by System Integrity Protection.(Notice the note
about SIP)
```

This is the list of the environment variables provided by ‘man’:

```
DYLD_FRAMEWORK_PATHDYLD_FALLBACK_FRAMEWORK_PATHDYLD_VERSIONED_FRAMEWORK_PATHDYLD_LIBRA
```

We can see in the the dyld open source code how the function *processDyldEnvironmentVariable()* handles the environment variables:

```
 processDyldEnvironmentVariable( * key,  * value,  * mainExecutableDir) { (
strcmp(key, “DYLD_FRAMEWORK_PATH”) == 0 ) {appendParsedColonList(value,
mainExecutableDir, &sEnv.DYLD_FRAMEWORK_PATH);}  ( strcmp(key,
“DYLD_FALLBACK_FRAMEWORK_PATH”) == 0 ) {appendParsedColonList(value,
mainExecutableDir, &sEnv.DYLD_FALLBACK_FRAMEWORK_PATH);}  ( strcmp(key,
“DYLD_LIBRARY_PATH”) == 0 ) {appendParsedColonList(value, mainExecutableDir,
&sEnv.DYLD_LIBRARY_PATH);}  ( strcmp(key, “DYLD_FALLBACK_LIBRARY_PATH”) == 0 )
{appendParsedColonList(value, mainExecutableDir, &sEnv.DYLD_FALLBACK_LIBRARY_PATH);}
( (strcmp(key, “DYLD_ROOT_PATH”) == 0) || (strcmp(key, “DYLD_PATHS_ROOT”) == 0) )
{...
```

## opensource-apple/dyld

### Contribute to opensource-apple/dyld development by creating an account on GitHub.

github.com

## How Does dyld Ignore Environment Variables

We just saw how the function *processDyldEnvironmentVariable()* processes the variables, but there is another function that “prunes” them, and the function is:

```
pruneEnvironmentVariables( * envp[],  *** applep)
```

We can see in the source code how is parsing and deleting the variables that match DYLD_* and LD_LIBRARY_PATH, and what is more interesting we can see the three reasons for the restiction:

```
//// For security, setuid programs ignore DYLD_* environment variables.// Additionally, the DYLD_* enviroment variables are removed// from the environment, so that any child processes don't see them.//  pruneEnvironmentVariables( * envp[],  *** applep){// delete all DYLD_* and LD_LIBRARY_PATH environment variables removedCount = 0; ** d = envp;( ** s = envp; *s != ; s++) {  ( (strncmp(*s, "DYLD_", 5) != 0) && (strncmp(*s, "LD_LIBRARY_PATH=", 16) != 0) ) {*d++ = *s;} {++removedCount;}}*d++ = ;// <rdar://11894054> Disable warnings about DYLD_ env vars being ignored.  The warnings are causing too much confusion.#if 0 ( removedCount != 0 ) {dyld::log("dyld: DYLD_ environment variables being ignored because "); (sRestrictedReason) { restrictedNot:; restrictedBySetGUid:dyld::log("main executable (%s) is setuid or setgid\n", sExecPath);; restrictedBySegment:dyld::log("main executable (%s) has __RESTRICT/__restrict section\n", sExecPath);; restrictedByEntitlements:dyld::log("main executable (%s) is code signed with entitlements\n", sExecPath);;}}#endif...
```

These are the three reasons for the restriction:

1. . These are flags can be assigned to an App to run with the privileges of the owning user or group, i.e. not in the current user’s context. Instead of creating an entry in the sudoers file, which must be done by root, any user can specify the setuid or setgid flag to be set for their own applications. These bits are indicated with an “s” instead of an “x” when viewing a file’s attributes via `ls -l`. The `chmod` program can set these bits with via bitmasking, `chmod 4777 [file]` or via shorthand naming, `chmod u+s [file]`. This is a vulnerability that can be exploited.
2. . These is a segment in the Mach-O file that can be created at link time. No specific content is needed. Acts like a flag to harden the process.
3. . In the code signing process of the App an entitlements flag can define the hardening of the App. The entitlements can be configured by Xcode enabling the runtime hardening:

https://help.apple.com/xcode/mac/current/#/dev88ff319e7

## Hardened Runtime Entitlements

### Enabling the Hardened Runtime capability allows your app to execute with additional security protections and resource…

developer.apple.com

## The DYLD_INSERT_LIBRARIES Variable

Returning to our purpose, there is one environment variable that sounds interesting for us **DYLD_INSERT_LIBRARIES.** It sounds like what we want to achieve. Let's read its documentation.

> DYLD_INSERT_LIBRARIES
>
> This is a colon separated list of dynamic libraries to load before the ones specified in the program. This lets you test new modules of existing dynamic shared libraries that are used in flat-namespace
>
> images by loading a temporary dynamic shared library with just the new modules. Note that this has no effect on images built a two-level namespace images using a dynamic shared library unless

This seems to be exactly what we are looking for. Let's inject something into a Mac App. What better to inject than Organismo.framework to have some fun, it is useful too.

## Our First Injection, Calculator.app

You may inject any library you may want, but if you want to have fun with an interesting tool to explore Apps then download the latest version of Organismo framework from here.

## JonGabilondoAngulo/Organismo-Lib

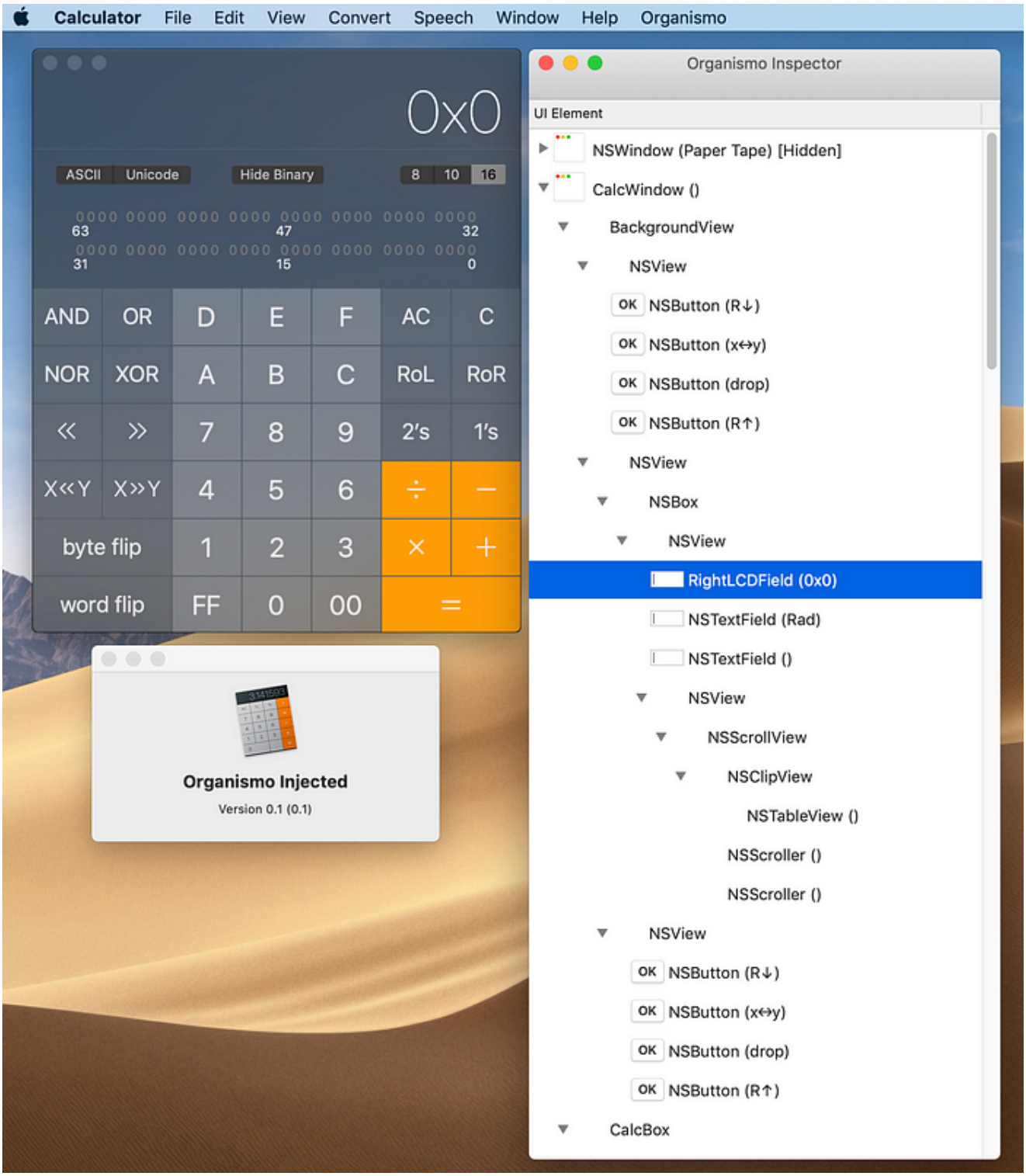
### Organismo framework for Mac be injected into Mac Apps to explore them at runtime. Organismo is a framework to bypass…

github.com

This is all it takes to insert a dylib into Calculator.App as of today with Mojave 10.14.5, SIP disabled. (Close Calculator App before).

```
$ DYLD_INSERT_LIBRARIES=/path_to/Organismo-mac.framework/Versions/A/Organismo-mac
/Applications/Calculator.app/Contents/MacOS/Calculator
```

Calculator should be running now. The Organismo dylib has been loaded by the dyld into the App. You can see a new Organismo menu, select Inspector and have fun exploring.




Calculator App with Organismo injected. Organismo shows the UI tree.

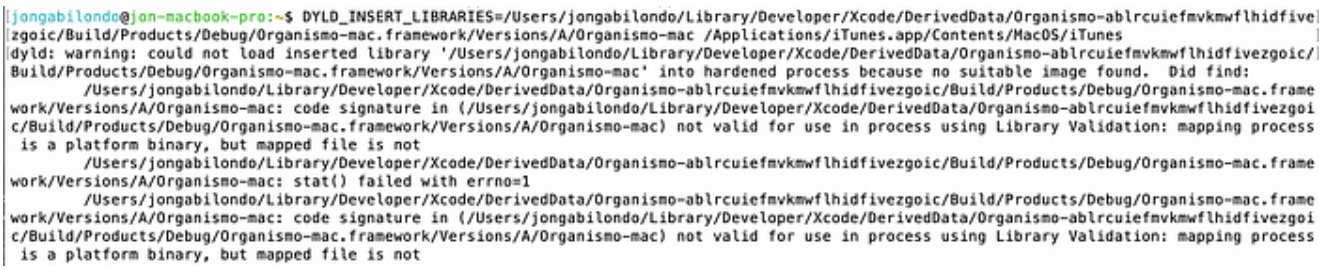
## Injecting Into Other Apps

You can have fun exploring other applications injecting Organismo. Some of the Apple Apps are not yet hardened and the simple injection explained so far, does work. This true at least up to Mojave 10.14.5 in Apps Calculator, Chess, Calendar, Mail, Number, Keynote, Dictionary, iMovie.

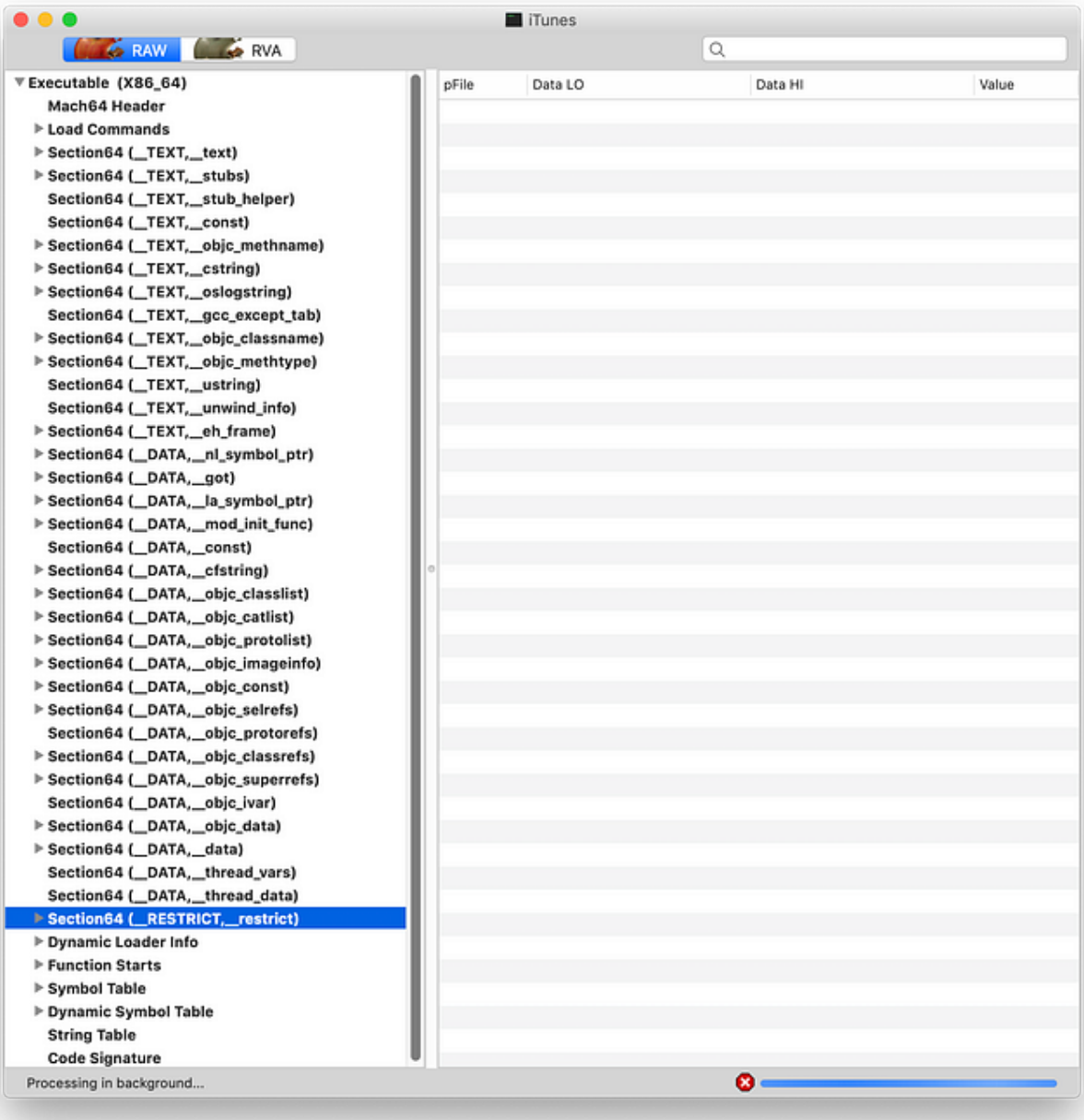
But this is matter of time until all Apps will be strongly hardened. We can an example of a hardened App with iTunes.

```
DYLD_INSERT_LIBRARIES=.../Organismo-mac.framework/Versions/A/Organismo-mac
/Applications/iTunes.app/Contents/MacOS/iTunes
```

The iTunes App is launched, but the Organismo library has been rejected. iTunes is hardened against external code. Although it tries to load the dylib because SIP is disabled, it finds another obstacle in the code signing control. The Organismo dylib codesign does not match the App's codesign.




Inspecting iTunes Mach-O we can confirm that it has been hardened by the method of the __RESTRICT segment. Have a look at the next screenshot:




iTunes App _RESTRICT segment.

## Hardened Apps. Non Injectable Apps ?

We can see in the open source code of dyld.cpp how the code signing restriction is handled by the dyld. The dyld asks the kernel if the code signature of the App makes it restricted. It checks the CS_ENFORCEMENT and CS_REQUIRE_LV flags from asking “csops” (code signing options).

This is the function where this is handled:

```
  processRestricted( macho_header* mainExecutableMH, * ignoreEnvVars, *
processRequiresLibraryValidation)
```

In Apps like iTunes, Xcode etc. we can’t inject code with DYLD_INSERT_LIBRARIES because they are restricted at codesign level.

But I have good news for those who still want to get more fun. We can take a more complex approach and still inject code even to iTunes, Xcode and any App with code signature restriction. It will be all revealed in part II of this story.

## Thanks !

Get to part II for more interesting stuff on code injection into hardened Mach-O Apps.

## How to Inject Code into Mach-O Apps. Part II.

### In Part I we saw how easy it is to inject code into Mac Apps, from Calculator to Mail, even more surprisingly, into…

medium.com

If you enjoyed it give it a clap and you may visit the github repo to download fun stuff and give some stars there too.

## JonGabilondoAngulo - Overview

### Sign up for your own profile on GitHub, the best place to host code, manage projects, and build software alongside 36…

github.com

## dyld Literature

## Executing Mach-O Files

### Explains the use of the OS X runtime architecture, including program types, loading and executing code, and using…

developer.apple.com

## Introduction

### Explains how to design, implement, and use dynamic libraries.

developer.apple.com

## mikeash.com: Friday Q&A 2012-11-09: dyld: Dynamic Linking On OS X

**In the course of a recent job interview, I had an opportunity to study some of the internals of dyld, the OS X dynamic…**

www.mikeash.com

## Mach-O Literature

## Overview of the Mach-O Executable Format

**Guidelines for reducing the size of an application binary.**

developer.apple.com

## Mach-O - Wikipedia

**Under NeXTSTEP, OPENSTEP, macOS, and iOS, multiple Mach-O files can be combined in a multi-architecture binary. This…**

en.wikipedia.org

## Mac Dev Center: Mac OS X ABI Mach-O File Format Reference

**This document describes the structure of the Mach-O (Mach object) file format, which is the standard used to store…**

web.archive.org

## aidansteele/osx-abi-macho-file-format-reference

**Mirror of OS X ABI Mach-O File Format Reference. Contribute to aidansteele/osx-abi-macho-file-format-reference…**

github.com

## Parsing Mach-O files - Low Level Bits

**The article covers basics of Mach-O parsing. If you're looking for a 'how to start', then you are at the right place.**

---

lowlevelbits.org