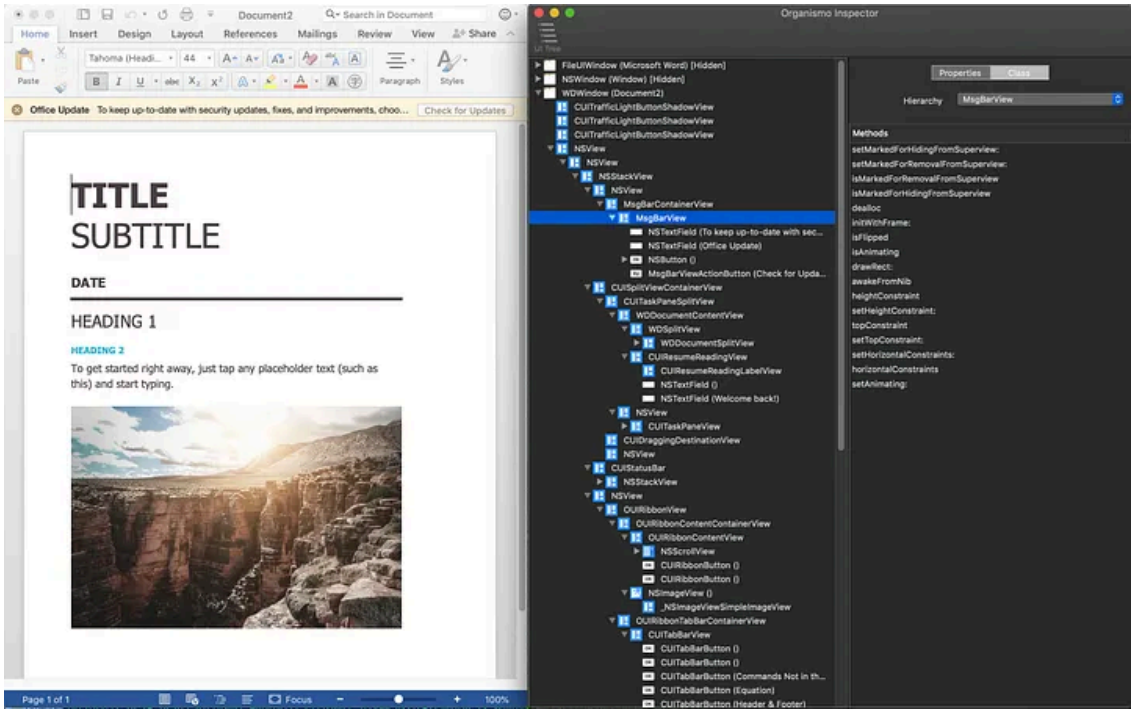
How to Inject Code into Mach-O Apps. Part II.

By Jon Gabilondo

Published: 2022-07-25 · Archived: 2026-04-05 14:15:30 UTC

Dynamic Code Injection Techniques

Press enter or click to view image in full size



Organismo-App-inspector within Microsoft Word 2018.

Update !

This article was written while finding a solution to inject Organismo framework, or any other library, into hardened applications. The article described a journey to overcome the limitations that prevent code injection, until we stumbled into the final hurdle of AMFI and the Gatekeeper.

Gladly, a little research on AMFI brought a surprising solution. You may jump to the end of the article to find out the solution to inject code freely on any Mac App. However the content until there is really interesting if you are into the details of the OS X (& iOS) system security.

In Part I we saw how easy it is to inject code into Mac Apps, from Calculator to Mail, even more surprisingly, into Microsoft Apps like Word 2018. Why would such important applications not have a simple protection (hardening)

against external code injection is not easy to understand. In fairness we must say that in Part I we worked with the condition of disabling System Integrity Protection (SIP) which is a major security layer on OS X.

In Part I we used a dynamic code injection techique using the DYLD_INSERT_LIBRARIES environment variable, an old property of the Dynamic Linker ‘dyld’ to load external libraries.

However, as one would expect, the simple dynamic code injection used in Part I would not succeed in (important) Apps such as iTunes, Xcode, Photos... These Apps are hardened to instruct the ‘dyld’ to disable the injection defined by environment variables as well as to reject code that does not match code signature of the Application.

In this story we will walk towards finding a solution to inject external code into hardened Apps like iTunes and Xcode.

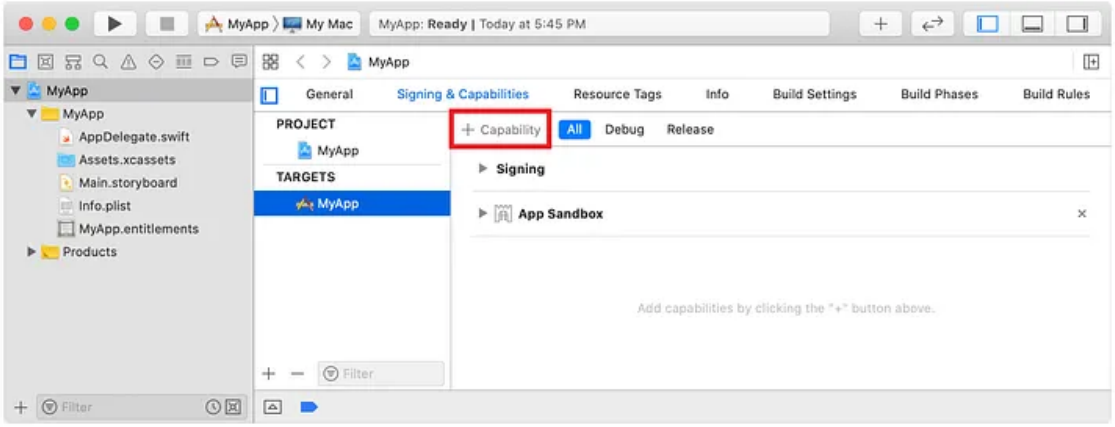
Hardened Runtime

The Hardened Runtime, along with System Integrity Protection (SIP), protects the runtime integrity of your software by preventing certain classes of exploits, like code injection, dynamically linked library (DLL) hijacking, and process memory space tampering.

There are two ways to harden your Application, the official one is by using entitlements and a less common one by creating a __RESTRICT segment in the Mach-O binary.

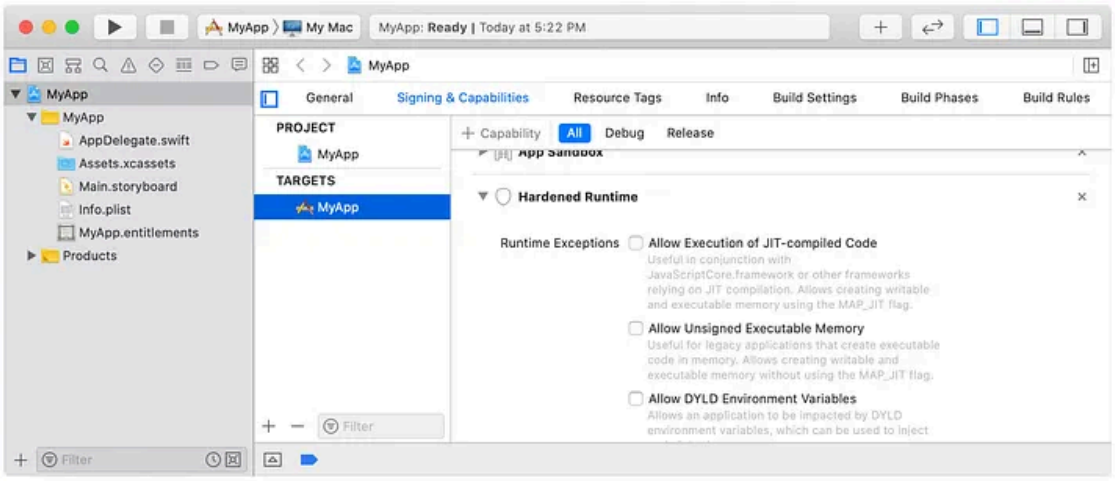
To harden your Application in Apple’s official way navigate in Xcode to your target’s Signing & Capabilities information and click the + button. In the window that appears, choose Hardened Runtime.

Press enter or click to view image in full size



Then add the capability ‘Hardened Runtime”. Check in ‘Build Settings’ that “Hardened Runtime” is enabled.

Press enter or click to view image in full size



Find Apple's documenttion here:

Hardening by Entitlements

We've seen that Xcode allows to harden one Application in a simple manner, but how are they applied to the binary and where are they kept in a binary.

Let's start by retrieving the the entitlements of any Application binary, this is done using *codesign*, lets' do it on Music.app:

```
$ codesign -d --entitlements :- /System/Applications/Music.appExecutable=/System/Applications/Music.
```

This is the output of the *codesign* command containing the xml list with the entitlements of the Music.app:

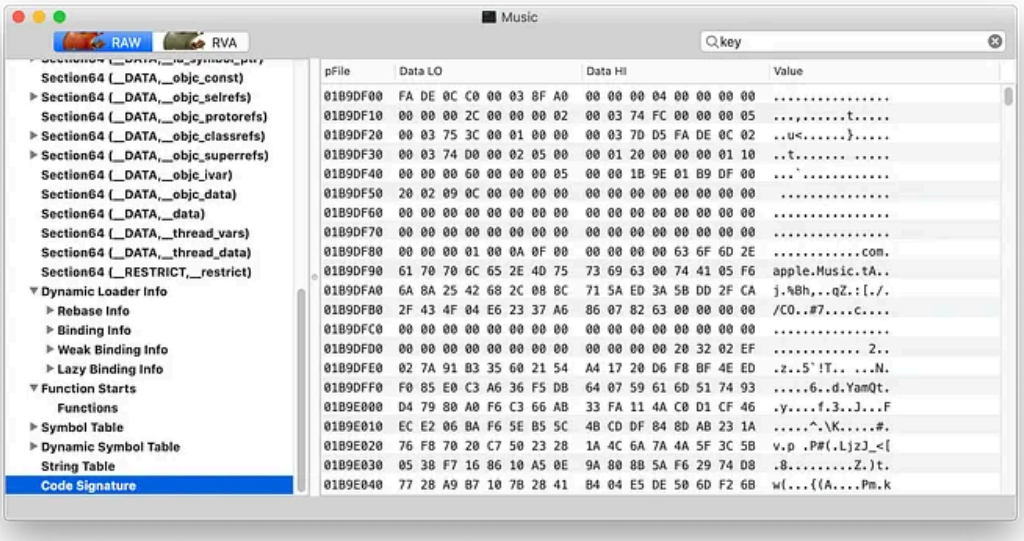
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd
<plist version="1.0">
<dict>
<key>com.apple.PairingManager.Read</key>
<true/>
<key>com.apple.PairingManager.RemovePeer</key>
<true/>
<key>com.apple.PairingManager.Write</key>
<true/>
<key>com.apple.amp.artwork.client</key>
<true/>
<key>com.apple.amp.devices.client</key>
<true/>
<key>com.apple.amp.library.client</key>
<true/>
<key>com.apple.application-identifier</key>
<string>com.apple.Music</string>
<key>com.apple.authkit.client.internal</key>
```

```
<true/>
<key>com.apple.avfoundation.allow-system-wide-context</key>
<true/>
<key>com.apple.avfoundation.allows-access-to-device-list</key>
<true/>
<key>com.apple.avfoundation.allows-set-output-device</key>
<true/>
<key>com.apple.cdp.recoverykey</key>
<true/>
<key>com.apple.mediaremote.allow</key>
<array>
<string>TVPairing</string>
</array>
<key>com.apple.private.accounts.allaccounts</key>
<true/>
<key>com.apple.private.applemediaservices</key>
<true/>
<key>com.apple.private.aps-connection-initiate</key>
<true/>
<key>com.apple.private.audio.notification-wake-audio</key>
<true/>
<key>com.apple.private.bmk.allow</key>
<true/>
<key>com.apple.private.commerce</key>
<array>
  <string>Accounts</string>
</array>
<key>com.apple.private.fpsd.client</key>
<true/>
<key>com.apple.private.notificationcenter-system</key>
<array>
 <dict>
 <key>identifier</key>
 <string>com.apple.appstoreagent</string>
 </dict>
</array>
<key>com.apple.private.rtcreportingd</key><true/><key>com.apple.private.security.storage.mo
bilesync.heritable</key>
<true/>
<key>com.apple.private.sqlite.sqlite-encryption</key>
<true/>
<key>com.apple.private.tcc.allow</key>
<array>
 <string>kTCCServiceAddressBook</string>
 <string>kTCCServicePhotos</string>
 <string>kTCCServiceAppleEvents</string>
 <string>kTCCServiceSystemPolicyAllFiles</string> <string>kTCCServiceCamera</string>
```

```
</array>
<key>keychain-access-groups</key>
<array>
 <string>com.apple.pairing</string>
 <string>com.apple.airplay</string>
 <string>apple</string>
</array>
</dict>
</plist>
```

At runtime the OSX/iOS needs to validate the operations and resources accessed by the executable, for that it needs to know what has the App been validated for, via the entitlements. The entitlements are kept in the binary Mach-O structure itself, in the ‘Code Signature’ setion.

Press enter or click to view image in full size



The Code Signature section in Music.app binary.

Entitlements is therefore a simple xml list of keys defining the entitlement and its properties. Although Apple documents and offers a handful of them to programmers, internally there might be hundreds.

The assignment of entitlements to an App is a matter of defining the entitlements xml file in the *codesign* command:

```
$ codesign --entitlements entitlements.xml  -f -s "iPhone Distribution: Company (XYZ)" Payload/Examp
```

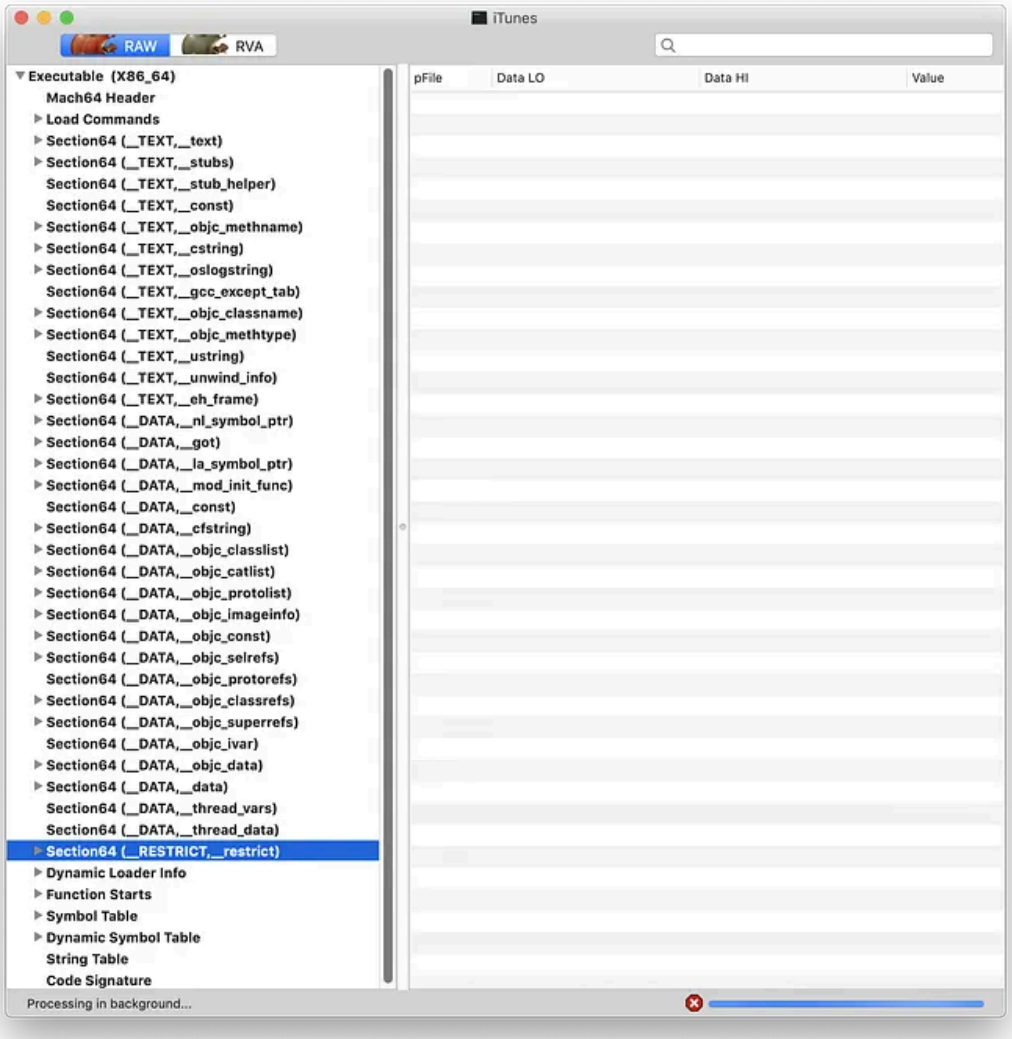
Hardened by __RESTRICT

A **__RESTRICT** segment is a section in the Mach-O binary file that can be created at link time. The section has no content. It acts like a flag to instruct the ‘dyld’ to perform code signature validation to all code loaded to the

process.

See bellow how does the __RESTRICT section look like in iTunes:

Press enter or click to view image in full size



The __RESTRICT section in iTunes.

Creating the __RESTRICT segment in Xcode is actually very easy. It is done by adding the following flags into your “Other Linker Flags”:

```
-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null
```

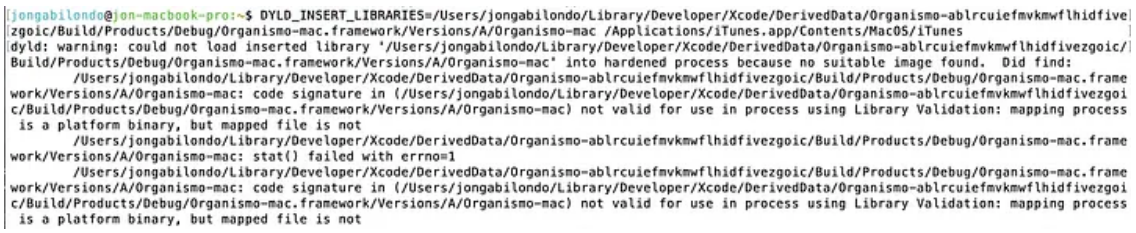
Why Is the Injection Rejected

The __RESTRICT segment flags the ‘*dyld*’ to activate the code signature validation for all library to be loaded.

We saw in Part I how the injection in iTunes failed with the following message:

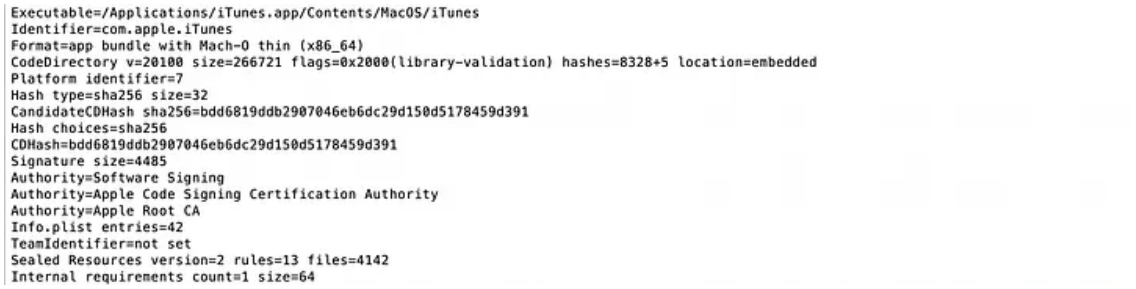
“dyld warning: could not load inserted library into hardened process because no suitable image found. Code signature in framework not valid for use in process using Library Validation”

Press enter or click to view image in full size



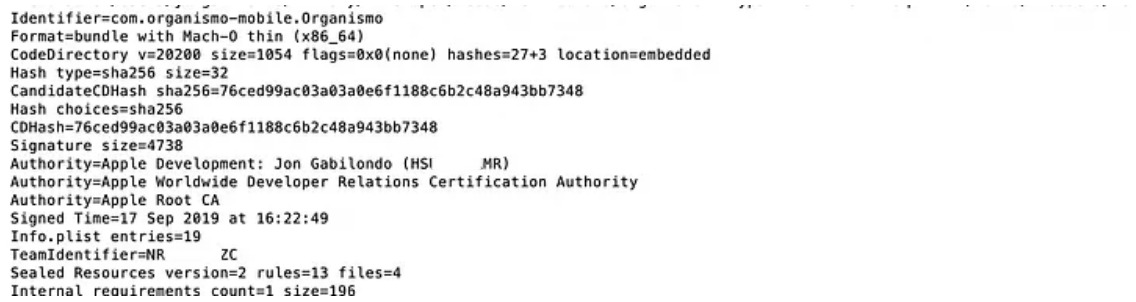
If we analyse the code signature of iTunes and Organismo we can see they obviously have a different code signing values (certificates):

Press enter or click to view image in full size



iTunes code signature values.

Press enter or click to view image in full size



Organismo code signature values.

Approaches to Overcome Hardening

In order to overcome the hardening barrier, these are the ideas that come to my mind. From foolish to realistic, they are:

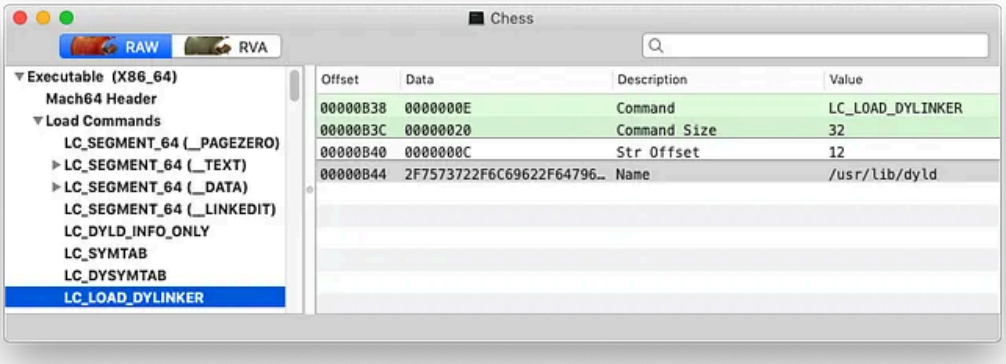
1. Build our own tweaked ‘dyld’ and instruct the Application to use it, instead of the default ‘/usr/lib/dyld’.
2. Codesign all software components with your own Certificate.
3. Remove the __RESTRICT segment.

Option 1. Create a Custom dyld

This is certainly a bold idea. To build a custom ‘dyld’ removing all the security measures that would not care about SIP or __RESTRICT, sounds exciting. This option comes motivated by two ‘reasonable’ facts. First, the ‘dyld’ code is open source:

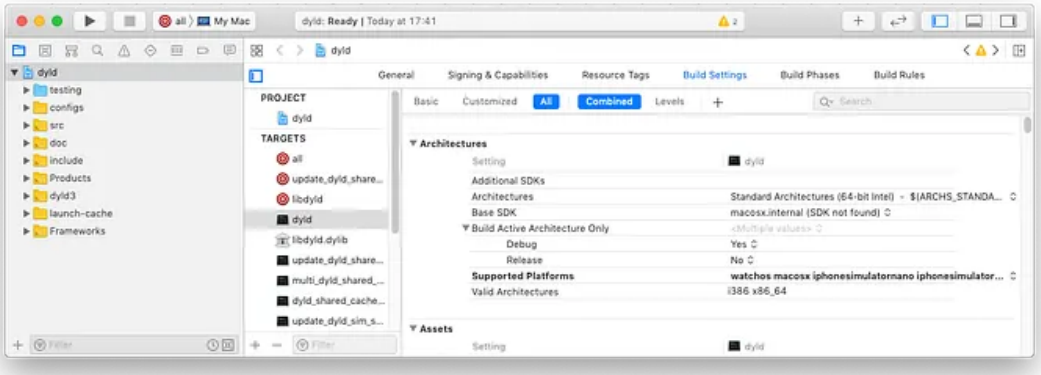
second, as we saw in part I, the Mach-O binary of every App specifies what dynamic linker must be used. In Mac OS X is always ‘/usr/lib/dyld’:

Press enter or click to view image in full size



As of today the macOS 10.14.5 open source site has more than one hundred projects. The Dynamic Linker of OS X and iOS is right there. Impressive.

Press enter or click to view image in full size



Well, first problem:

```
Base SDK   MacOSX.internal (SDK not found)
```

MacOSX.internal… Okay… I understand, Apple has its own SDK, makes sense. Can I get it ? Obviously not. So lets set the Base SDK to what we’ve got by default: ‘macOS’.

What happens from then on is a never ending compilation attempts with missing includes: *_simple.h, Block_private.h, coreSymbolicationDyldSupport.h, CrashReporterClient.h, cs_blobs, objc-shared-cache*, etc. etc. etc.

Get Jon Gabilondo’s stories in your inbox

One can painstakingly find the missing includes from different packages of the open source, until it makes no sense to continue. There must be a better way to do it, but it’s nowhere to be found.

This option ends here. One wonders how does Apple understand open source.

Option 2. Re-Codesigning with Your Certificate

We assume that the Organismo framework is rejected by the ‘dyld’ because its code signing certificate is different from the one of iTunes. Organismo uses your Developer Certificate, iTunes uses Apple’s Private Certificate.

Therefore we assume that code signing iTunes with our Dev Certificate, the ‘dyld’ will pass the code integrity validation when loading Organismo.

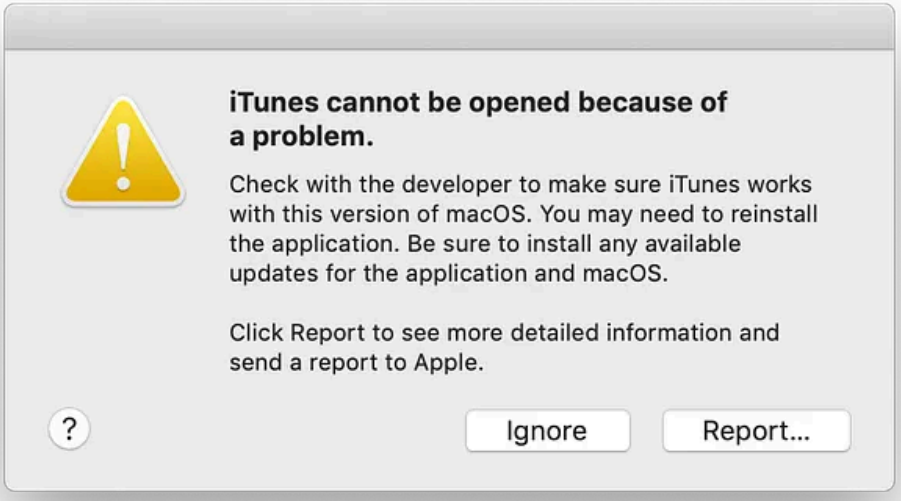
Re-codesigning any App is as simple as (remember to work on a copy of the App !):

```
$ codesign --deep --force --preserve-metadata=entitlements -s ‘Mac Developer: Jon Gabilondo (HSU·····
```

Checking the signature with ‘codesign -dvvv’ we can actually see that the signature is now ‘Jon Gabilondo’, which is what we wanted. Let’s run the modified iTunes to see if it runs properly, before we attempt the injection:

```
$ DYLD_INSERT_LIBRARIES=/path_to/Organismo-mac.framework/Versions/A/Organismo-mac /Users/jongabilond
```

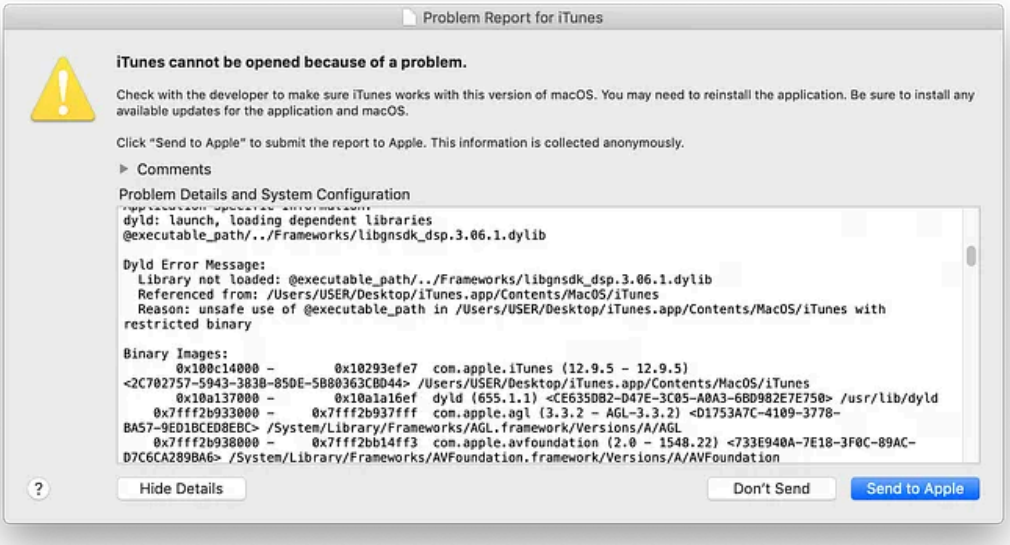
Press enter or click to view image in full size



The assumption of having the same code signature was right, partially. It did not consider the ramifications of dylib dependencies that will always end up in an Apple system dylib with Apple's Signature. Like we see in the error report of the modified iTunes, the dylibs of iTunes have Jon's signature, but its dependencies always end up in Apple's OSX /usr/lib dylibs.

Option 2 not good either.

Press enter or click to view image in full size



Error loading dylib because binary is restricted.

It is always very useful to read Apple's **Code Signing In Depth**:

Option 3. Removing the __RESTRICT segment

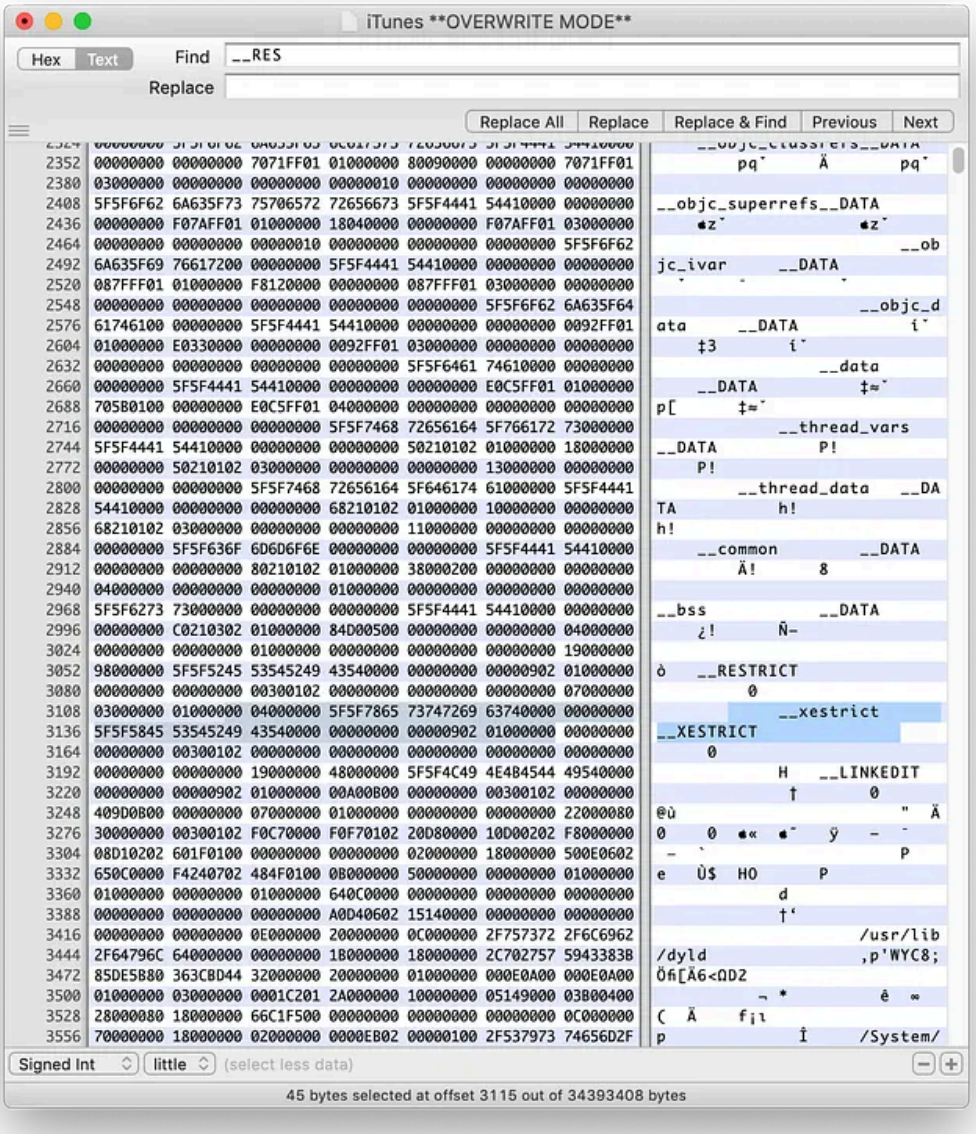
I’m left with the last bullet. If the __RESTRICT segment is the flag that is activating the code signature validation, we might just remove it from the Mach-O binary.

This approach requires static code modification, which obviuously causes the binary (checksums) invalidation. But we know how to recodesign an App with our Dev Certificate so there should be no problem.

Removing the __RESTRICT segment

Several tools could be used to modify a Mach-O binary file, you can choose your favourite. Here I will use iHex editor. Find the __RESTRICT segment and rename it , for instance to: __XESTRICT. Save the file and codesign it with your Dev Certficate.

Press enter or click to view image in full size



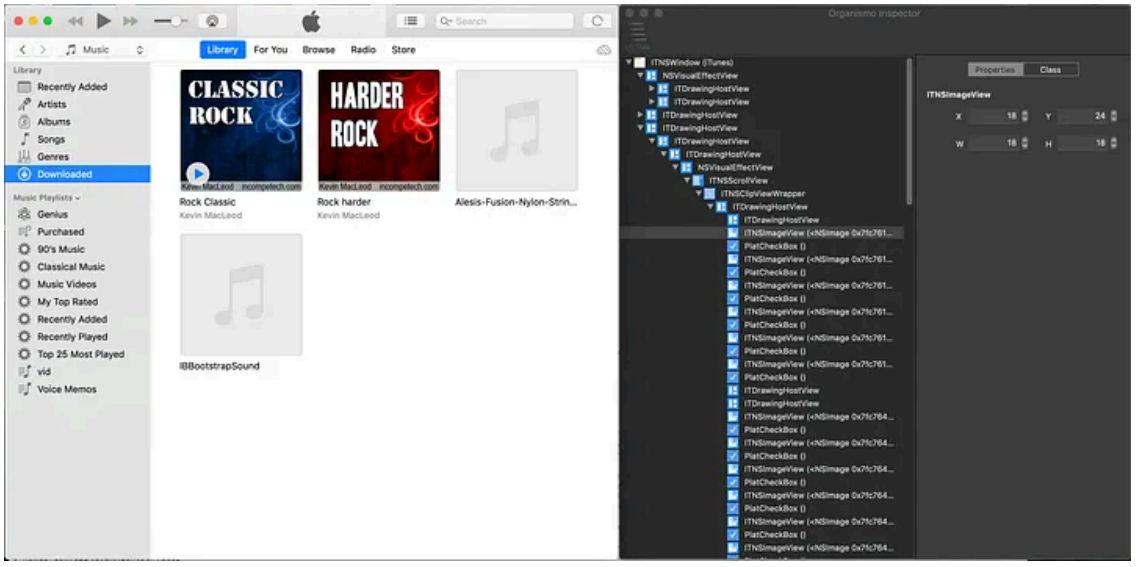
Renaming the __RESTRICT segment of iTunes Mach-O binary.

Now let’s inject Organismo into iTunes:

```
$ DYLD_INSERT_LIBRARIES=/path-to/Organismo-mac.framework/Versions/A/Organismo-mac /Users/jongabilond
```

This time Organismo framework was successfully loaded into iTunes.

Press enter or click to view image in full size



Update for Catalina!

Mac OSX Catalina has added new system protections. After changing the RESTRICT section the Apps and re-codesigning with the Dev Certificate it does result in a non valid App.

Music.app crashes on start. The logs hint to the the **taskgated-helper** and **amfid** processes**.** The error says: **Disallowing com.apple.Music because no eligible provisioning profiles found.**

Here the Console logs:

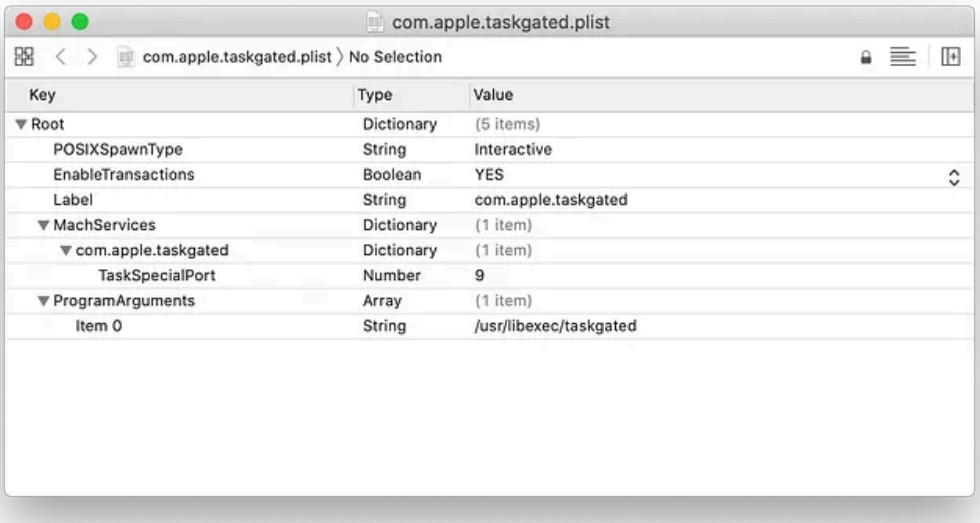
```
error 11:06:47.743379+0200 taskgated-helper Disallowing com.apple.Music because no eligible provisio
 success = 0;
}default 11:06:47.743834+0200 amfid Failure validating against provisioning profiles: No eligible pr
```

The **taskgated** process belong to a daemon that can be found in:

```
/System/Library/LaunchDaemons/com.apple.taskgated.plist
and the related:
/System/Library/LaunchDaemons/com.apple.taskgated-helper.plist
```

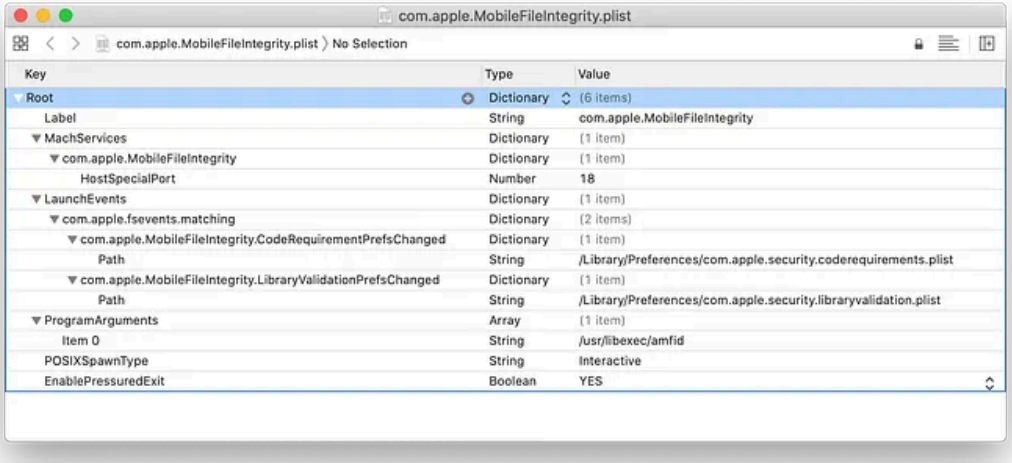
Which executables are : */usr/libexec/taskgated* and */usr/libexec/tasgated-helper*

Press enter or click to view image in full size



The **amfid** is a daemon of the Apple Mobile File Integrity found in */System/Library/LaunchDaemons/com.apple.MobileFileIntegrity.plist* which launches */usr/libexec/amfi*.

Press enter or click to view image in full size



This is a very interesting article on AMFI:

We can assume that these two Apple Security daemons are rejecting our codesigned Music.app because the entitlements it requires are not accepted for our Dev Certificate. To confirm this we can create an entitlements file based on Music.app's entitlements and removing the entries that start *com.apple.private.**. The resulting Music.app will pass the AMFI and Gatekeeper security validation and it will launch, but it will fail in operations that require those entitlements.

These is a great WWDC video on Gatekeeper and other security updates for Catalina:

The Solution

The ending of the first version of the article was this one:

For my purpose of making Organismo App Inspector still work in Catalina and coming versions, the direction to take is to get dyld, AMFI and the Gatekeeper 'out of the way', i.e. to disable their integrity validation.

The expression to '*get them out of the way'*, could have not been more premonitory. See the entry on AMFI on iphonewiki.

```
The amfi kext recognizes quite a few boot-args, including:amfi_unrestrict_task_for_pid – Allowing th
amfi_allow_any_signature – Allowing any digital signature on code, not just Apple's
amfi_get_out_of_my_way – disable amfi
cs_enforcement_disable – Disable code signing enforcement
cs_debug – Debug code signing
```

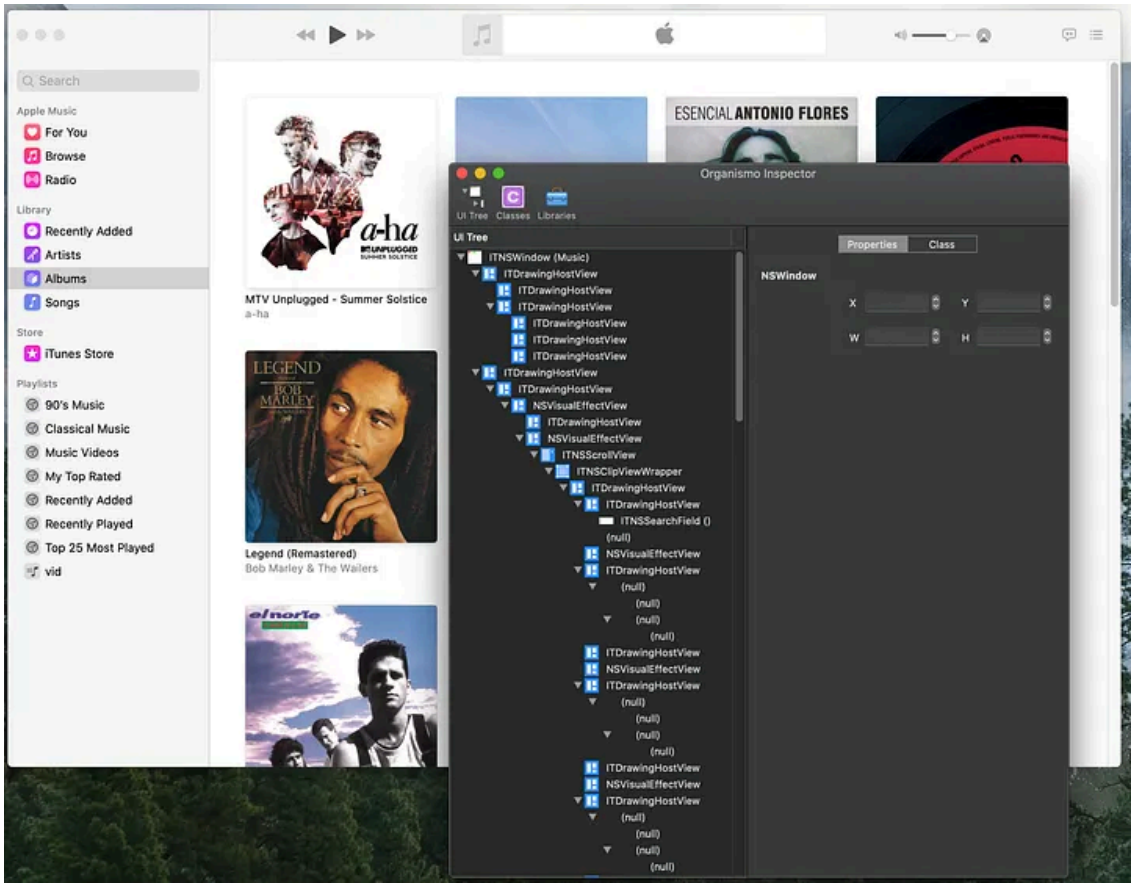
We can set a boot-arg to disable AMFI daemon process completely !

```
% sudo nvram boot-args="amfi_get_out_of_my_way=0x1"
(careful with the quotes if you copy-paste.)
```

Reboot. The AMFI will disappear from your system. Organismo is free to be injected into any App.

```
% DYLD_INSERT_LIBRARIES=/path/to/Organismo-mac.framework/Versions/A/Organismo-mac /System/Applicatio
```

Press enter or click to view image in full size



Organismo injected into Music.app

Thanks !

I hope you enjoyed it. You may explore it yourself using Organismo.

Source: https://jon-gabilondo-angulo-7635.medium.com/how-to-inject-code-into-mach-o-apps-part-ii-ddb13ebc8191