Sem Voigtländer

Jailed INC  The Netherlands

Apple private APIs

Security by obscurity

Apple Private APIs: Security by Obscurity

# Documenting the undocumented

## Index

### 1. Firmware and Private APIs

### 1. Location of Private APIs

### 2. Use of private APIs

### 3. Benefits of private APIs

### 2. Reverse engineering Private APIs

### 1. Live runtime extraction: An abusive approach

### 2. Extraction of the dynamic library shared cache

### 3. Dumping classes and generating header files

### 4. Live runtime reverse engineering

### 5. Reconstructing the private APIs

### 3. Finding, abusing and exploiting design and security flaws in Apple Private APIs

### 4. The future of Private APIs

# Finding and extracting Private APIs

## iOS Firmware and Private APIs

### Firmware

iOS Firmware is downloadable from Apple’s servers.

It comes in format of an IPSW, which is the extension Apple identifies iOS firmware with.

An IPSW is just a zip file, renaming the .ipsw extension to .zip will allow you to be able to extract It.

After the extraction a folder is created containing a few DMG files and folders with binary images.

The folders contain the kernel and bootloader for all modes of the device.

The DMG files are disk images for each mode iOS can be in.

The DMG file with the biggest file is the root file system.

Since iOS 10 Apple does not encrypt the root file system anymore as Apple wants to make the way free for security researchers.

The disk image is verified at boot time, modifications are therefore not possible as the kernel will not load the disk image.

A dmg file can be extracted with any software able to extract Apple HFS Journaled File systems.

On Linux, there is a project called darling-dmg that can do this which is Open Source under the GPL 3.0 License (<https://github.com/darlinghq/darling-dmg>).

The reason we want to extract the root file system is because it contains the resources we need to get the private frameworks from.

This method can be skipped if you own an iOS device, but if you have no iOS device then this is the only way to get the frameworks.

Downloading frameworks shared on GitHub repositories by other researchers is not recommended as those can be truncated and incomplete.

After extraction of the rootfs there will be a file called dyld\_shared\_cache in

/System/Library/Caches/com.apple.dyld/dyld\_shared\_cache\_armX

Where X is your desired processor architecture.

This file is compressed with Apple’s own LZSS Compression Algorithm.

An implementation of the compression can be found open source by Apple at (<https://opensource.apple.com/source/boot/boot-132/i386/boot2/lzss.c>).

There is also a mac binary available created by me that can automatically extract all frameworks from the dyld\_shared\_cache.

### Private APIs

On any iOS device the Private Frameworks are in a directory that you have read permissions for even inside an appcontainer of a sandboxed process.

The path to the location of the Private Frameworks is /System/Library/PrivateFrameworks.

Each Framework (ex: Preferences.Framework) is a folder containing the binary library, an (xml) property list containing information about it (Info.plist), sometimes localization (.lproj, .strings).

The binary is a flat MACH-O binary of type dynamic library, something comparable to what in Linux is a .so file (ex: /usr/lib/libgcc.so).

### Use of Private APIs

Private APIs in iOS are used to perform tasks in the higher level of the operating system.

They are mostly designed to provide communication with low-level processes with as goal to process information provided by its sender.

They are used to easily do things that involve system operations such as looking up which applications are installed, managing VPN connections or Extracting files to locations outside the sandbox.

The reason Private APIs are Private is because they reveal for a large part how the higher level of the operating system works.

They perform operations that can be sometimes insecure, therefore they are kept private.

Private means that these APIs are not documented by the vendor, however the symbols are in these frameworks are not obfuscated yet and obfuscating them would be pointless as they can be dumped in runtime. This makes it easy to reverse engineer the frameworks and get an idea about how they are used and what parts of the system they interact with.

### Benefits of Private APIs

The reason why my research went out to private APIs is because they can be useful in case of performing operations that Apple doesn’t make possible with their public APIs, because the private Frameworks once reverse engineered save you time programming functionality that they already have, because if you use them you can minify the size of your application because use symbols that already exist on the file system, but mostly I started researching them because I had worries about the security of those. I couldn’t come up with a reason why they are kept undocumented and close source other than that Apple is trying to prevent a security breach by obfuscating possible vulnerabilities. After research I was proven to be right.

## Reverse Engineering Apple Private APIs

### Reverse engineering without an iOS device

Without an iOS device you can take the extracted Private Frameworks from the dynamic library shared cache in combination with open source software called class-dump.

Class-dump is able to reconstruct header files for all objective-C based binaries and in this case private APIs. The header files give a better insight in what the private API does and how you can use it. They are not very good for use in projects.

Linking against Private Frameworks is not possible many times, because these frameworks are loaded into the memory of the application and the binaries are very big.

Imagine you want to load a few binaries of 50MB into the memory while your memory page size is 16K only. That just won’t work. So you will have to load the frameworks programmatically and reconstruct the fake symbols to make use of the API.

### Reverse engineering with an iOS device

With an iOS device private APIs can be loaded into the memory of an iOS Application with dlopen or NSBundle.

The easiest way is to load them with NSBundle.

Because they are MACH-O format binaries, they all start with an offset: FE ED FA CE.

You can look up the main executable segment of the binary (\_mh\_execute\_header) in the MACH-O header with dlopen.

Using functionality of NSRuntime, Obj-c Runtime and dlfcn you can reverse engineer the classes.

A project is released Open Source by me at <https://github.com/MTJailed/PrivateAPIManager>

The project is able to dump classes from a framework or library, dump the class’s public properties and dump the methods of the class including guesses of the parameters types.

Using

Class SomeClassName = NSClassFromString(@”Example”);

You can reconstruct classes.

Make sure you do this inside an interface.

You will get a selector missing error at build time.

Reconstructing the selectors inside the interface will silence the warnings.

### Finding, Abusing and Exploiting design and security flaws in Apple Private APIs.

Remember that I started this research because I though the main reason for keeping these APIs private could be security by obscurity? I was right.

Private APIs seem to have been exploited since iOS 6, but after iOS 8 they have been forgotten about by both Apple and Security researchers until shortly.

About three months ago Italian security researcher Luca Todesco from KJC Research Ltd. Found a vulnerability in the StreamingZip Private API. Apple had the vulnerability patched in iOS 11.1.1 and Luca Todesco did not disclose a PoC to the public.

I have found the exact vulnerability based on the information Apple’s patch log gave:

“A path handling issue was addressed by improved path handling”

“An attacker might be able to modify the file system through a crafted archive from within a sandboxed process”

The issue is in the unzipping method of the private API. You can make the XPC Service use its own sandbox token by reading the private property from itself.

After that you can perform path traversal in the output path parameter of the extractor.

Causing you to be able to write files to /var/tmp and /var/mobile.

In iOS 6 vulnerabilities in Private APIs were way more dangerous, but this new report shows us that they still exist.

The vulnerability I am talking about in iOS 6 was that a private API was able to reset and set a new passcode of the iPhone, causing an attacker to be able to change the iPhone’s passcode with a maliciously crafted app.

Not the kind of things that you want to be possible on your iPhone.

What nowadays is the major security, more a design flaw, is that the APIs are able to reveal information about files, data and directories outside the policies and paths of the sandbox.

For example you can get the installed applications, the phone number of the owner of the phone and multiple privacy disclosing information.

This is not a vulnerability but Apple’s sandbox policies normally prohibit the disclosure of this information and Apple does care about the privacy of their customers.

The information can be used to identify you and in some contexts the information can be used to detect whether a user has a vulnerable application installed or not.

### The future of Apple Private APIs

Apple enforces code-signing. Meaning all code is verified by a daemon (amfid). And if the code does not have a valid certificate signed by Apple’s CA then the code will not run.

Entitlements are a list of capabilities that a developer can add to his application.

Apple will sign and verify this entitlements file and it is not possible to change the contents at runtime due to the sandbox policies.

If amfid detects that an app contains invalid entitlements then it will not run the app.

My idea and hypothesis is that Apple will enforce entitlements specific for the use of Private APIs and their capabilities in the future. This makes it impossible for developers to use Private APIs and ends the ability to use them for good.

Does this improve security? No, the vulnerabilities if they exist are still exploitable in another context they remain of course.

It took Apple 12 years before they open sourced the iOS kernel and iOS’s bootloader.

I think Apple will not feel empathy for open sourcing the Private APIs.

Therefore I think Apple will focus on improving the iOS Sandbox to take down the vulnerabilities in private APIs.

Does this solve the security issues completely? No.

Imagine the scenario of an attacker finding an exploitable vulnerability in Webkit.

The attacker gains arbitrary code execution and knows a vulnerability exist in the MobileBackup private API that gives him the ability to read the private keys from iOS’s keychain.

The attacker then exploits the vulnerability in the Private Framework through Webkit and is still able to make use of the vulnerability and get the private keys as webkit is not a sandboxed application.

So to conclude this is what I think the future of Private APIs, Apple will remain to leave them close source, but will improve sandbox policies and enforce entitlements to prevent Private Frameworks from being exploited or abused by App developers.

Attackers however are still able to exploit the vulnerabilities from within a privileged context and therefore I can keep pointing out to Apple, that there is need to open source the Private Frameworks for the sake of security.