BINARY BROTHERHOOD

PLAN WISELY | TEST QUICKLY | REMEDIATE RISK

White Paper
iOS Application Penetration Testing
Bypassing common Jailbreak detection mechanisms
Part I

Version 1.1, January 2021

Binary Brotherhood, Inc.
2093, Philadelphia Pike, Claymont
DE, United States, 19703
Phone: 1 864 474 5086 | 1 323 498 3636
International: 64 210 232 6722
E-mail: contact@binarybrotherhood.io

Table of Contents

1. Introduction

1.1 Word ahead

General jailbreaking terms and history of jailbreaking (there are plenty of articles, blogs, talks, and research papers out there) will not be covered in this article as the focus will be on how to overcome what is usually the first obstacle when performing the penetration testing of iOS applications, and that is, bypassing the jailbreak detection, assuming that the application is implementing one.

On the other hand, penetration testers usually use jailbroken iPhone devices as it allows them to run arbitrary unsigned code, meaning using custom tools and tweaks, making the penetration testing process more straightforward. When performing an iOS application penetration testing, a penetration tester is usually faced with two challenges, the first being jailbreak detection. The second one is SSL Certificate Pinning to inspect the application's network traffic and perform server-side testing.

The more sophisticated the jailbreak detection is, the more difficult it is to bypass it.
In most cases, bypassing jailbreak detection is simple. It does not represent a great challenge for an experienced mobile application penetration tester as most jailbreak detection methods fall into the following categories:

- File existence checks
- URI scheme registration checks
- Sandbox behaviour checks
- Dynamic linker inspection

We will examine each of those techniques in our series of articles, starting with the simple jailbreak detection implementations where a single function returns true or false depending on whether the device is jailbroken and later as we progress to more advanced jailbreak detection mechanisms implementing anti-debugging techniques.

Because there are plenty of already available Frida scripts for jailbreak detection bypass, the goal of this article will be to describe a complete process of inspecting the application and writing scripts from the ground up to evading a jailbreak detection.

For the task ahead, we are going to use a jailbroken iPhone running iOS 14.3 (the latest version as of writing), Hopper Disassembler for reverse-engineering the application and Frida (dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers), and some other macOS tools that can help us in our journey.

1.2 Background

Apple strongly discourages iOS users from jailbreaking and states that "Unauthorized modification of iOS can cause security vulnerabilities, instability, shortened battery life, and other issues," which is true. However, end-users often tend to jailbreak their devices to install various tweaks, add additional features, and install third-party apps from unofficial app stores, possibly being malicious.

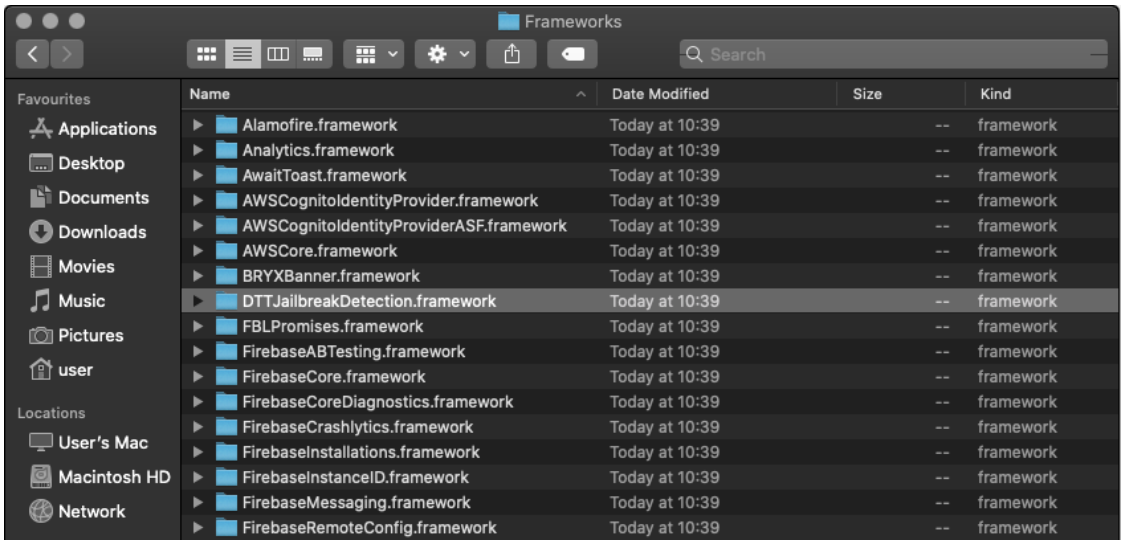
This poses an enterprise security risk, meaning that companies implement their detecting methods if their application is running on a jailbroken device. If so, the application usually alerts the user and does not allow the user to continue using the application.
While it is essential to implement jailbreak detection mechanisms for prevention, they cannot be wholly relied upon and sometimes can provide a false sense of security. Nevertheless, a jailbreak detection mechanism in place can provide valuable information for Telemetry.
Many companies, especially financial institutions, want insight into how many of their users are running the application on an insecure device, as some of those users are potentially malicious ones.

2. Reverse engineering, a popular Fintech application

The first thing we will need is to find a popular Fintech application that implements a simple jailbreak detection; however, due to this article's nature being for educational purposes only, the name of the application will remain private.

Let us get to work!

We will assume that the reader already knows how to dump decrypted iOS applications*. We will skip this step and immediately check the *Frameworks* folder to find out is there a jailbreak detection library present, as shown in the next figure:



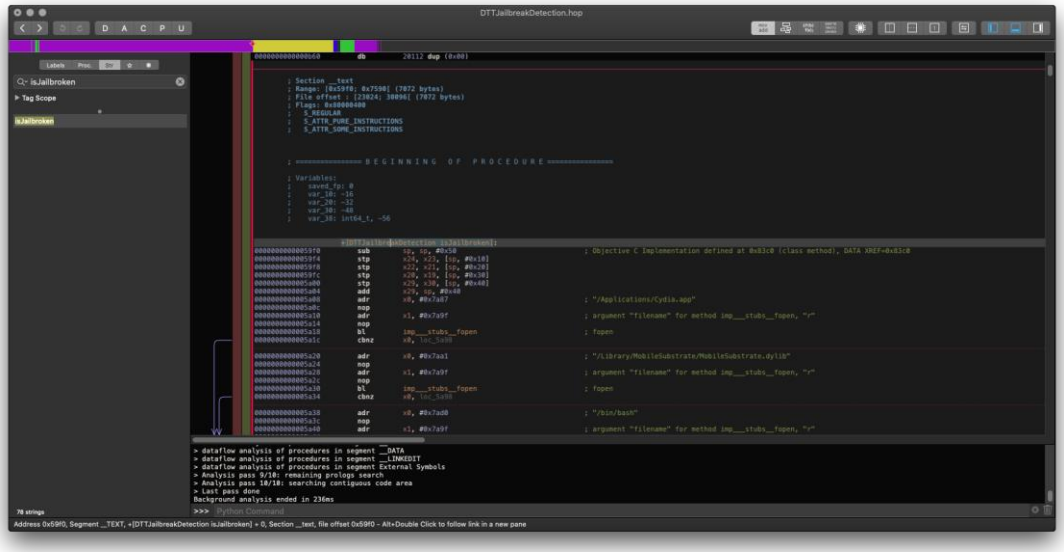
As we can notice the usege of a *DTTJailbreakDetection** framework, and a quick Google search revealed that this is a library to detect if an iOS device is jailbroken or not.

We can quickly review this library's source code; however, not many jailbreak detection libraries will have their source code available, so we will proceed as if we do not have such information.

We would first want to look for symbols in the framework to find more information, and for that task, we can use the *nm* tool as illustrated in the figure below:

```
Downloads — -zsh — 116×58
[user@Users-Mac Downloads % nm DTTJailbreakDetection
0000000000059f0 t +[DTTJailbreakDetection isJailbroken]
                U _CFStringGetCStringPtr
0000000000007a18 S _DTTJailbreakDetectionVersionNumber
0000000000079e0 S _DTTJailbreakDetectionVersionString
00000000000086d0 S _OBJC_CLASS_$_DTTJailbreakDetection
                U _OBJC_CLASS_$_NSFileManager
                U _OBJC_CLASS_$_NSObject
                U _OBJC_CLASS_$_NSURL
0000000000008680 S _OBJC_CLASS_$_PodsDummy_DTTJailbreakDetection
                U _OBJC_CLASS_$_UIApplication
00000000000086a8 S _OBJC_METACLASS_$_DTTJailbreakDetection
                U _OBJC_METACLASS_$_NSObject
0000000000008658 S _OBJC_METACLASS_$_PodsDummy_DTTJailbreakDetection
00000000000086f8 s _OBJC_METACLASS_$___ARCLite__
                U __DefaultRuneLocale
00000000000083b8 s __OBJC_$_CLASS_METHODS_DTTJailbreakDetection
0000000000008518 s __OBJC_$_PROTOCOL_INSTANCE_METHODS___ARCLiteKeyedSubscripting__
0000000000008550 s __OBJC_$_PROTOCOL_METHOD_TYPES___ARCLiteKeyedSubscripting__
0000000000008420 s __OBJC_CLASS_RO_$_DTTJailbreakDetection
0000000000008370 s __OBJC_CLASS_RO_$_PodsDummy_DTTJailbreakDetection
0000000000008318 s __OBJC_LABEL_PROTOCOL_$___ARCLiteKeyedSubscripting__
00000000000083d8 s __OBJC_METACLASS_RO_$_DTTJailbreakDetection
0000000000008328 s __OBJC_METACLASS_RO_$_PodsDummy_DTTJailbreakDetection
0000000000008750 d __OBJC_PROTOCOL_$___ARCLiteKeyedSubscripting__
0000000000008638 s __OBJC_PROTOCOL_REFERENCE_$___ARCLiteKeyedSubscripting__
```

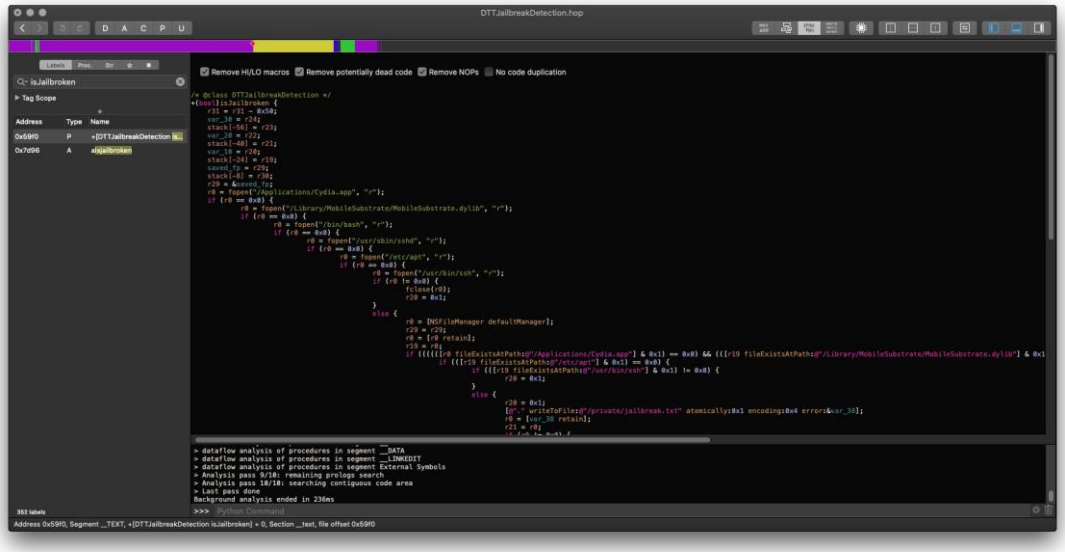
The first thing noticed is that there is a method *+[DTTJailbreakDetection isJailbroken]*.
We say it is worth inspecting, so we will investigate using Hopper Disassembler and search for *isJailbroken* string as shown below:



We now have noticed several familiar strings that are helping us to go deeper:

- “/Applications/Cydia.app”
- “/Library/MobileSubstrate/MobileSubstrate.dylib”
- "/bin/bash"

With this information in hand, the best thing to do is to decompile the method and try to understand the process of jailbreak detection, and the next figure shows it:



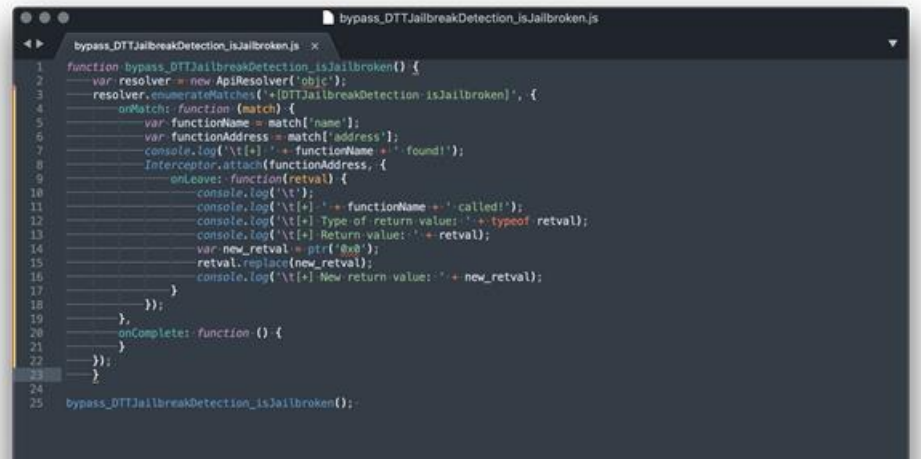
By looking at the decompiled code, we can see that the (bool) method checks for the existence of several files using *NSFileManager* instance method *fileExistsAtPath:** and if the file is found, the method returns *0x1* (True). The list whose existence is checked as follows:

- /Applications/Cydia.app
- /Library/MobileSubstrate/MobileSubstrate.dylib
- /bin/bash
- /etc/apt
- /usr/bin/ssh
- /usr/sbin/sshd

This means that we need to hook this method and return *0x0* (false) to bypass jailbreak detection. Knowing this, we can proceed with writing our Frida script.

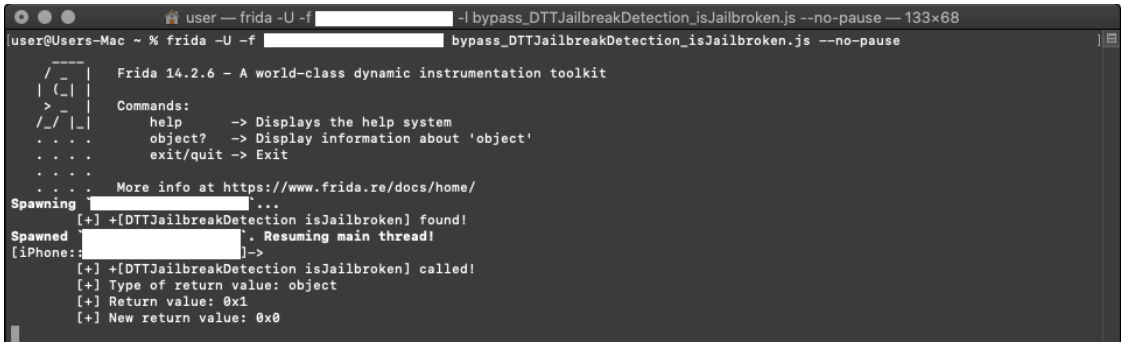
2.1 Writing a jailbreak detection bypass script using Frida

Assembling all the pieces we gathered because of application reverse engineering, and our final Frida script will look like this example:



Our script is fundamental. It enumerates the method we previously identified. Once found, the process is hooked, the return value is determined (0x1) and replaced with the new return value (0x0).

At this point, we will assume you already know how to execute a Frida script, so let us try it out and observe the outcome as demonstrated in the following figure:



3. Conclusion

In this Part I, we detailed a basic example of the jailbreak detection bypass method.
The method depends on an approach that returns true or false, well known and the most used by the iOS penetration testers.

If a client is looking to follow threat model analysis, more sophisticated methods must be implemented. However, you must keep in mind that even those can be circumvented with varying levels of difficulty. The overall goal here is to make a determined attacker's job as difficult as possible.

In Part II, we will describe a more advanced technique.

4. References

- https://github.com/thii/DTTJailbreakDetection
- https://developer.apple.com/documentation/foundation/nsfilemanager/1415645-fileexistsatpath

5. About Us

With a strong focus and experience in the United States market, Binary Brotherhood Inc. is a United States tight-knit, client-focused, IT security consultancy company, with only one scope to provide a unique alternative to the traditional penetration testing approach, Security Skills as a Service - the best of worlds between the standard penetration testing, security consultancy, and the bug bounties programs, fuelled by the most skilled and vetted security professionals globally distributed.

Find us in AWS Marketplace. | Check Clutch.co reviews.

We are specialized in providing the following adversarial simulations services:

- Penetration Testing (Web Apps, API, Infrastructure)
- Mobile Security Testing
- IoT / API Penetration Testing
- Security Source Code Review
- Thick Clients and Desktop Applications Security Review
- Vulnerability Assessment
- Threat Modelling
- Cloud Red Teaming
- Cloud Config Security Review
- Containers Security Review
- Cloud Architecture Security Review
- Cloud Adoption and Migration Consultancy

6. Copyright

7. Disclaimer