Analysis of an Info Stealer — Chapter 2: The iOS App

medium.com/@icebre4ker/analysis-of-an-info-stealer-chapter-2-the-ios-app-0529e7b45405

Fr4 January 10, 2024

```
uVar3 = _objc_allocWithZone(&_OBJC_CLASS_$_CNContactStore);
uVar3 = _objc_msgSend(uVar3,"init");
uVar5 = FUN_100004fe4(&DAT_100142f28);
lVar10 = _swift_initStackObject(uVar5,auStack_1e8);
*(lVar10 + 0x18) = 6;
*(lVar10 + 0x10) = 3;
contacts_data =
     static_String._unconditionallyBridgeFromObjectiveC(*PTR__CNContactFamilyNameKey_10012d8b8);
*(lVar10 + 0x20) = contacts_data;
contacts_data =
     static_String._unconditionallyBridgeFromObjectiveC(*PTR__CNContactGivenNameKey_10012d8c0);
*(lVar10 + 0x30) = contacts_data;
contacts_data =
     static_String._unconditionallyBridgeFromObjectiveC
               (*PTR__CNContactPhoneNumbersKey_10012d8c8);
*(lVar10 + 0x40) = contacts_data;
uVar5 = FUN_1000111a50(lVar10);
uVar7 = _objc_allocWithZone(&_OBJC_CLASS_$_CNContactFetchRequest);
uVar11 = FUN_100004fe4(&DAT_100142f30);
uVar11 = Array._bridgeToObjectiveC(uVar5,uVar11);
uVar7 = _objc_msgSend(uVar7,"initWithKeysToFetch:",uVar11);
_swift_bridgeObjectRelease(uVar5);
```

Fr4

--

Introduction

This is the second part of the article series: "*Analysis of an Info Stealer*". In this chapter, I will analyze the iOS info stealer app, which was delivered through the phishing website discussed in the preceding article. If you haven't had the chance to read the first article and you are curious about the distribution methods of this malicious app, you can catch up by reading it here:

Preface

Reversing an iOS app presents its own set of challenges, distinguishing itself from the process of reversing an Android app (that is definitely easier for a variety of factors). In light of this distinction, I decided to create this article with a twofold objective:

- To continue the analysis of this malicious campaign and examine the entire attack chain.

- But also to explain some basics and share some ‘tricks’ I use to speed up and improve the efficiency of iOS app analysis

Technical Analysis

In the previous chapter, I showed the iOS app download process, highlighting the specific use of an enterprise certificate. This certificate is installed before downloading the *.ipa* file and is used by threat actors to verify the legitimacy of the app.

Figure 1 — Malicious app verified using an enterprise certificate

Now that the malicious app has been installed on the iPhone, the initial step is to dump the *.ipa* file with frida dump in order to start the static analysis.

> “A **.ipa** file is an iOS and iPadOS application archive file which stores an iOS/iPadOS app. Files with the .ipa extension can be uncompressed by changing the extension to .zip and unzipping.”

Static Analysis

After downloading and unzipping the *.ipa* file, the observed structure in Figure 2 (on the left) reveals the main files for analysis. Specifically, I will focus on:

- file that stores settings and other data in a key-value format (Figure 2 — on the right)
- file, that is the Mach-O (Mach-Object) executable, which is a native format for executables on macOS and iOS. The following file contains the code of the app.

Figure 2 — .ipa file structure (on the left) and the content of the Info.plist (on the right)

The file provides valuable information, including:

- `CFBundleDisplayName: Telegram Viewer` is the user-visible name of the bundle, visible on the Home screen in iOS
- `CFBundleExecutable: viewer` is the name of the bundle’s executable file.
- `CFBundleIdentifier: com.cafe24.viewer` is an identifier string that specifies the app type of the bundle
- `CFBundleSupportedPlatforms: iPhoneOS` specifies the platforms for which the app is designed to run.
- `MinimumOSVersion: 14.0` specifies the minimum version of the operating system required to run the app.
- `NSContactsUsageDescription: ""` specifies a message that tells the user why the app is requesting access to the user’s contacts. When the app is launched, users can anticipate a prompt seeking permission to access contacts on their iPhone.

> In the context of iOS app development, the "**CF**" prefix stands for "**Core Foundation**." Core Foundation is a C-based framework in the macOS and iOS operating systems that provides fundamental data types and services for macOS and iOS applications. The "**NS**" prefix typically stands for "**NextStep**," which refers to the original name of the framework that evolved into Cocoa, the primary application framework on macOS and iOS. Keys with the "NS" prefix are often related to various configurations and permissions.

To analyze the *viewer* file, I utilized . Prior to delving into the code analysis, I used a Ghidra script developed by *Laurie Wired*, named "SwiftNameDemangler.py". This script helps improve code readability and simplifies Swift code, making it more accessible for in-depth examination. (If you have been dealing with reverse C++ code, you have most likely already used the demangling feature).

Figure 3 — Comparison between "normal" and demangled decompiled code

Since the amount of code inside *Ghidra* is quite huge, strings are a good starting point to begin the static analysis. From the analysis of the *Info.plist* file we have seen that the app would access the user's contacts, so the "" string is a good candidate.

Figure 4— String search inside Ghidra

Starting from this function (Figure 4) and using the "*Find References to*" feature of Ghidra, it is possible to identify what should be the "core" function of the malware: `FUN_1000111e4(void)` ; so let's break it down.

- The `CNContactStore` object represents the user's contacts store database, and you use it to fetch information from that database and save changes back to it.
- `authorizationStatusForEntityType` returns the current authorization status to access the contact data.

Figure 5 — Code of `FUN_1000111e4 function`

With these two lines of code, the malware checks whether it has access to the user's contacts, verifying if the required permission was granted during the app's launch.

Going down a little bit in the code (Figure 6), it is possible to observe that:

- `CNContactStore` see above
- `PTR__CNContactFamilyNameKey` , `PTR__CNContactGivenNameKey` , `PTR__CNContactPhoneNumberKey` appear to be pointers to an object that represents the key, the key and the key in the`CNContact` object.
- `CNContactFetchRequest` is an object that defines the options to use when fetching contacts.
- `initWithKeysToFetch` creates a fetch request for the specified keys.

Figure 6— Fetching contacts data code

In summary, this piece of code is fetching contacts data from the `CNContactStore` object, and storing the result in a variable.

Another interesting string to search is “” in order to get the Command and Control (C2) server’s URL but also other details such as the paths used, the parameters, etc.

Figure 7— String search inside Ghidra

In fact, analyzing the code, the following information can be extracted:

- The URL of the C2 server is: `https:]api.]telegraming.]pro`
- The paths used are: `getregistertoken` and `getuploadtoken`
- The library is used to manage the HTTP communication. This information can also be obtained by looking at the huge amount of references within the code or through Ghidra’s “” window.

Figure 8 — Alamofire library

Dynamic Analysis

With an overview of the malware’s functionality gained from static analysis, let’s proceed to the dynamic analysis using Frida, Objection and Burp, aiming to further analyze:

- how the malware works “in action”
- which directories and files are used
- the network traffic

When launching the “*TelegramViewer*” app, users are presented with a prompt seeking permission to access contacts, a behavior anticipated based on the earlier static analysis conducted.

Figure 9 — Launching of the malicious app

If the user allows permission, they are then prompted to enter a phone number and is shown a button labeled “Open Album.” It is worth noting that the use of the term “album” in this context could possibly be attributed to a copy-and-paste error, particularly when considering the distribution of other apps masquerading as secret album management apps.

Figure 10 — Malicious app “in action”

However once the button is pressed, the following POST requests are made:

- `/getregistertoken` : the network, iphone model, victim phone number and iOS version are sent to the C2 server

- `/getuploadtoken` : for each POST request the data of a contact saved on the device are sent. In particular the: first name, last name and phone number (as analyzed in the static analysis section)

Figure 11 — Data sent to the C2 server

Another way to intercept communications and/or extract useful information is through Frida. In particular, `frida-trace` is very useful for dynamic tracing of method calls. For instance, in this case, we are aware that the app utilizes Alamofire to handle communication. By using the following command, we can trace some methods of this library.

```
frida-trace -U "Telegram Viewer" -m "-[Alamofire.SessionDelegate URLSession*]."
```

The `-m` option is used to filter the methods that will be traced. It allows you to specify a method signature pattern to match against. In this case, the pattern is `"-[Alamofire.SessionDelegate URLSession*]."`, indicating that the trace should include methods of the `Alamofire.SessionDelegate` class that start with `URLSession`.

After running the command and the malicious app, Frida will auto-generate multiple JavaScript files within a directory called `__handlers__` . These scripts serve as a solid starting point for editing and generating the necessary information as output.

Figure 12 — Example of output of modified __handlers__ script

Lastly, another interesting tool to use during dynamic analysis is Objection, which can be utilized, for instance, to retrieve pertinent directories associated with the app by using the `env` command. This will print out the locations of the applications *Library*, *Caches* and *Documents* directories. Although in some cases it is possible to retrieve interesting files and data, in this case I did not find any useful information :(

Figure 13 — Objection “env” command output

Conclusions

In conclusion, the analysis of the iOS info stealer app, discovered through the phishing website outlined in the previous article, has unveiled how this malware is able to steal the contacts and other personal information from the infected iPhone of the victims.

Lastly, I hope you’ll find this short article useful as a starting point to analyze an iOS malware app. Stay tuned for the next article!

Indicators of Compromise (IOCs)

File Name: Telegram Viewer.ipa
Md5: 660ccad4b26abc543e64fe2319ae5771

iOS Mach-O Binary Name: viewer
Md5: 47f2c25499473a82348d1a1568c6b591

C2 Server: https:]//api.]telegraming.]pro