Dolosoft

iOS application security software

Ander Moran

SUMMARY

The analysis and modification of iOS applications remains an important topic in information security. This report considers different methods of analysis, how these methods work, and how they are implemented into Dolosoft, which is software capable of fully automating iOS app analysis and attacks based on dynamic code injection. Depending on the language the app was written in and the complexity of the attack, the time needed to do so can be reduced from hours to minutes through a suite of features built around dynamic code injection.

Table of Contents

1 Introduction

With millions of iPhone, iPad, and iPod Touch consumers using 3rd party iOS applications daily, it is essential to evaluate the security of these apps. People across the world use their smartphones every day to do things like browse the web or send pictures to loved ones. As a result of this constant usage, there is a plethora of sensitive personal data on mobile devices including text messages, pictures, notes, contacts, and geographical location. How these applications handle this data is of big concern given the data's sensitive nature.

When an iOS app is submitted to Apple's App Store, it is examined by Apple to confirm there is no malicious activity coming from the app. The inspection is effective most of the time, but there have been multiple instances of iOS apps slipping past Apple's checks, resulting in an abuse of user's privacy [1]. Because of this mistrust, it is important to analyze apps. Static analysis is the primary way apps are probed by experts in the industry. There is a lot of Objective-C metadata stored inside the binary, a Mach-O file, of an iOS app such as class names and methods. This makes static analysis very effective when using professional grade disassemblers like IDA or Hopper. These disassemblers can reveal assembly instructions, program flow, and function names but can only reveal data that is present within the binary file.

Many parts of the application's behavior cannot be understood with static analysis alone. For example, a social media app assigns a unique account number for every user's account. When the user logs into the app, the account number is encrypted then sent from the social media company's servers to the mobile app where it is decrypted and used by the app. If one tries to find the account number by statically analyzing the binary, they will fail due to the account number not being stored within the binary itself but rather sent through network traffic. If one tries to intercept the network packet, the account number will be encrypted so the data is of no use. This is where dynamic analysis, the testing and evaluation of a program during execution in real-time, is needed. Using dynamic analysis, it is possible to log the variable containing the account number value in real time.

The difference between static and dynamic analysis is the difference between a mechanic looking at a car engine while the car is off versus looking at a car engine while the car is on. Sure, the mechanic can view the engine, look at the parts, and amass a large amount of useful information; however, the mechanic cannot say how long it takes for the engine to start up or how loud the engine is, which are characteristics the mechanic would know if the mechanic was able to observe the engine in action. Another weakness of static analysis is the amount of time it takes for security researchers to analyze the app's behavior. It takes several days, even for those with plenty of experience, to read through lines of assembly when a modern app's binary can be hundreds of megabytes in size. This time consuming static analysis is worth the time in certain scenarios but in many cases the information collected is not worth the time spent. There are many advantages dynamic analysis has over static analysis, including a clear distinction between code and

data, information about variables, and the ability to modify the code executed by the app without changing the app's code. Dynamic analysis of iOS apps remains a secondary option due to the lack of software available, which makes the process tedious.

In this report, I discuss common mobile app analysis techniques as well as the power behind Dolosoft, software built to efficiently dynamically analyze and verify the security of iOS applications. I also describe the challenges I faced in developing software for a field where no one had ventured yet. By conjoining different elements of dynamic iOS app analysis, I have enabled an efficient, practical manner for testing application security. In short, I have contributed the following:

1. I explain the challenges facing dynamic analysis for iOS applications.
2. I show different methods on how to prevent iOS apps from being dynamically analyzed.
3. I open source my software built for automating the process of dynamically reverse engineering and attacking iOS applications.

2 Background & Related Works

Dynamic analysis is possible on iOS applications due to a property of the Objective-C language called method swizzling. To simply put it, method swizzling is dynamically changing what a method does. A more formal definition of method swizzling is "the process of changing the implementation of an existing selector" [2]. Imagine an elevator going to the 6th floor when the button for the 2nd floor is pressed.

```
1  @implementation Math
2  + (double)pi {
3      return 3.14;
4  }
5  @end
6
7  int main(int argc, char *argv[]) {
8      SEL s = NSSelectorFromString(@"pi");
9      [Math performSelector:s];
10 }
```

Listing. 1. A snippet of Objective-C code using a selector.

A selector is essentially the name of the method but is defined by Apple, the author of Objective-C, as "the name used to select a method to execute for an object, or the unique identifier that replaces the name when the source code is compiled" [3]. A selector object is associated with a designated class method. I will clarify this with the sample code in Listing 1. The selector is stored into the `SEL s` variable on line 9. The `NSSelectorFromString` function searches through all the selectors declared in the program to match the selector name passed through the parameter. When the selector that matches the name passed is found, it returns the selector object. Knowing this, line 9

instructs the `Math` class to call the method associated with the selector stored in the variable `s`. This results in line 9 returning the value of 3.14. This works because each method has a unique selector associated with it. Method swizzling is achieved by using the Objective-C runtime library.

The Objective-C runtime library is a library created by Apple that enables a program to change its functionality dynamically at runtime, allowing for code to be exceptionally adaptable. The library is useful for changing a framework's functionality to better suit the developer's needs, low level debugging, and bridging Objective-C code with other languages. Runtime is dynamically linked in every Mach-O binary that runs

```
typedef struct objc_class *Class;

struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;
#if !__OBJC2__
    Class super_class
    const char *name
    long version                                    OBJC2_UNAVAILABLE;
    long info                                       OBJC2_UNAVAILABLE;
    long instance_size                              OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars                    OBJC2_UNAVAILABLE;
    struct objc_method_list **methodLists           OBJC2_UNAVAILABLE;
    struct objc_cache *cache                        OBJC2_UNAVAILABLE;
    struct objc_protocol_list *protocols            OBJC2_UNAVAILABLE;
#endif
} OBJC2_UNAVAILABLE;
```

Listing. 2. The structure of an Objective-C class from Apple's source code [5].

Objective-C code [4]. Looking at Listing 2, it becomes apparent that every `Class` object in Objective-C has a method list it can call. The method list contains a group of `Method` objects that were defined in the implementation of the class. Inside the definition of an

```
struct objc_method {
    SEL method_name                                 OBJC2_UNAVAILABLE;
    char *method_types                              OBJC2_UNAVAILABLE;
    IMP method_imp                                  OBJC2_UNAVAILABLE;
}
```

Listing 3. The structure of an Objective-C method from Apple's source code [5].

Objective-C method shown in Listing 3, there are two properties that are interesting with regards to methods swizzling, `SEL method_name` and `IMP method_imp`. `SEL method_name` points to the name of the selector and `IMP method_imp` is a pointer that points to the starting address of the method's implementation. An implementation in Objective-C is simply a pointer to a function. Within the Objective-C runtime library

there is a function called `method_exchangeImplementations` that swaps implementations of two methods [6]. The concept of exchanging the implementations of two methods is the foundation of method swizzling in Objective-C.

```
IMP imp1 = method_getImplementation(m1);
IMP imp2 = method_getImplementation(m2);
method_setImplementation(m1, imp2);
method_setImplementation(m2, imp1);
```

Listing 4. The atomic version of the `method_exchangeImplementations` function [6].

The code displayed in Listing 4 shows how the `method_exchangeImplementations` function would be written. First, the `IMP`s of both `Method` objects (`m1` & `m2`) are retrieved using the `method_getImplementation` function and stored. Next, the implementations of each method are set to the other method's `IMP`. The result is that whenever m1 is called it performs the task of m2 and whenever m2 is called it performs the task of m1.

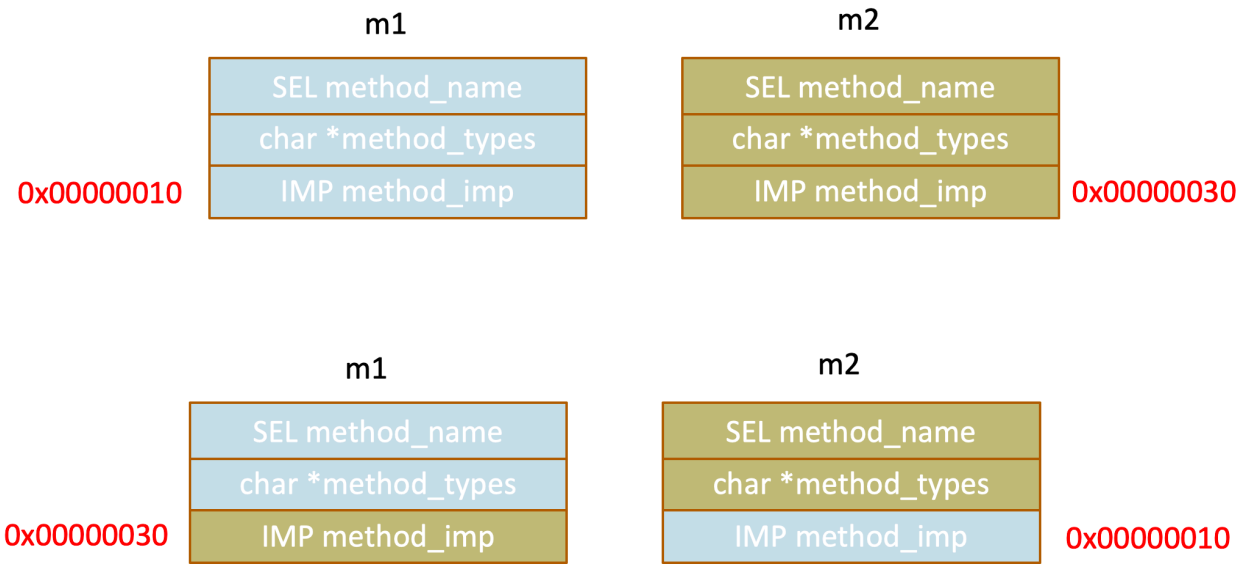

Fig. 1. A simplified view of the result produced by the `method_exchangeImplementations` function.

As illustrated in Figure 1, the address pointing to the `IMP`s of each Objective-C `Method` object are exchanged so now the address that each method's `IMP` points to differs from the original address. Method swizzling enables developers to change the behavior of a program without modifying the original binary and without having access to the source code of the program.

Swift, a programming language developed in 2014 by Apple, is another popular language for iOS app development. Apps written in Swift can still be analyzed because the Objective-C runtime library supports Swift [14], so the class metadata is still present

in compiled Swift code. Unfortunately for dynamic analysis and attacking, Swift does not support method swizzling. Variables can still be logged and attacks can still be executed, but only on Swift classes that are bridged with Objective-C such as `UIViewController`. A class written in pure Swift cannot be dynamically analyzed with Dolosoft as Apple did not build Swift to support swizzling.

Static techniques expose information about a program without the need for execution. As stated in the introduction, static analysis is the primary method of analyzing iOS applications. The two most popular programs for iOS reverse engineering are IDA and Hopper. Using the REobjc module, IDA can statically analyze parts of any decrypted 32-bit or 64-bit iOS application binary that is provided [7]. If the iOS app was written in Swift, there are a few open source GitHub projects that can parse Swift metadata and analyze the binary [8]. There is also a debugger, similar to `gdb`, that works for iOS apps within IDA, but the user must have the Xcode project of the target application due to OS restrictions within iOS [9]. IDA equipped with both the REobjc module and any Swift module is very powerful; however, it is limited given the nature of dynamic analysis. Hopper works similarly to IDA. It is a disassembler that specializes in iOS applications. It works out-of-the-box and has tools to retrieve class names, method names, strings, and control flow. Again, Hopper shows a great deal of valuable information but is limited, as it is only a static analysis tool.

The most popular command line dynamic analysis tool is a program called `Cycript` [10]. `Cycript` works by hooking itself to a running process and can execute commands as well as read/write to memory allocated for the process via the Objective-C runtime library [11] [10]. Dolosoft was inspired greatly by `Cycript` because of its lightweight design, lightning fast execution, and ease of use. As beneficial as it is, `Cycript` still lacks a GUI, which is some cases is advantageous, in toolchains for example, but not in this case. When it prints large amounts of data, it is difficult to parse and understand the information. Another issue is the changes made to an application by `Cycript` must be re-run every time the application is loaded into memory, which wastes a lot of time. This is expected due to the dynamic nature of Objective-C's runtime. A fix for this problem would be to write a `Cycript` daemon that hooks the application launched in iOS and applies a certain saved `Cycript` file every time a specific application is launched.

3 Methods/Approach

The goal of Dolosoft was to make software that could completely automate the analysis, both static and dynamic, and modification of an iOS app. This can be done in Dolosoft without writing any code, although the option to do so exists. The steps taken by Dolosoft to reverse apps are as follow:

1. Establish SSH connection
2. Scan filesystem
3. Select app of interest
4. Decrypt the app

5. Move the decrypted app to a computer
6. Parse the Objective-C metadata
7. Select methods to log and/or write code to attack app
8. Build and load the code to be injected
9. Run the app and inject code
10. View log

Fig. 2. A screenshot of Dolosoft after receiving application data from the device.

Dolosoft uses two shell scripts to collect properties of every iOS app installed on the device. These include the app's display name, executable name, bundle identifier, and paths related to the app, which will all be used to automate tasks like decrypting the app, writing to the log file path, etc. This data also enables Dolosoft to alphabetically list out all the installed applications as displayed in Figure 2.

Once all the apps are displayed, the researcher selects the application from the app list and clicks 'Analyze app'. When this button is clicked, a series of background tasks are performed on the computer and iOS device. First, an SSH connection is established between the computer and iOS device using `NMSSH` [13], a wrapper for `libssh2`. Second, the app is decrypted using a popular tool called `dumpdecrypted` [12] which loads the app into memory, decrypts it, and outputs a decrypted version of the app's binary. Apple, in an attempt to fight piracy, encrypts all app binaries on the iOS device. `dumpdecrypted` must be run on the phone, as every iOS app is encrypted for the device it is installed on so it would be impossible to decrypt the app outside of the iOS device. The iOS device must be jailbroken to allow access to the file system and the running of command line tools

including `dumpdecrypted`. After the decrypted binary is written, the binary is sent from the iOS device to the computer over the SSH connection.

The data in the binary is parsed by a Mach-O parser to collect Objective-C and Swift metadata used for runtime to generate header files. Inside of these header files are class names, methods, ivars (instance variable), properties, protocols, and data types. The classes and their respective methods are collected from the header files. Each method's declaration is parsed for its call syntax, return type, argument types, and master class using regex. This is the limit of static analysis on Dolosoft.

The next step is for the user to select which methods they wish to log. Logging a method logs the arguments passed into the parameters, the value or object returned by the method, in real time. All of this is logged in real time so that the researcher can view the log and then manipulate objects at specific memory addresses. If it was not logged

```
[cy# var a = new Instance(0x1024ecde0)
 #"<UILabel: 0x1024ecde0; frame = (15 15; 14 14); text = '1';
[cy# a.text
 @"1"
[cy# a.text = @"2"
 @"2"
[cy# a.text
 @"2"
```

Fig. 3. Cycript shell manipulating loaded variables.

in real time, this would not be as useful because the memory addresses change upon program execution due to ASLR [15]. Figure 3 is an example of why viewing a live log is important. Since the `UILabel` object was logged in real time, I knew the object was stored at address `0x1024ecde0`. With this information I created a new variable, `a`, from the data at address `0x1024ecde0` using `Cycript` (which is built into Dolosoft). This new object is not a copy of the original `UILabel` object but is a reference, so whenever I alter the text property of my variable it changes the original `UILabel` object as well, making this reference incredibly powerful.

Whenever the user selects the methods they want to log and click the 'Create tweak' button, an Objective-C file is automatically generated with the code necessary for logging via code injection. This file can be edited inside Dolosoft if the user needs to log additional information or change the implementation of a method to craft an attack. Pressing the 'Build/Load tweak' creates a "tweak" that is then loaded onto the iOS device through SSH using `Theos` [16]. A tweak is a debian file (similar to a `.zip`) which contains the dynamic library (`.dylib`) that will be injected into the target application's process at runtime. The dynamic library contains the code telling the application which methods to swizzle. In our case the methods to be swizzled were the selected methods, and their associated `IMP`s will be exchanged with the `IMP`s methods generated by Dolosoft. After the tweak has been loaded onto the device, `Cydia Substrate` [17] acknowledges that the

tweak has been loaded onto the device and will inject the tweak into the target app at runtime. Now, the app itself will log any methods or carry out any attacks due to the injection of the dynamic library. The logs are sent from the device to the computer over SSH and update every second. This whole flow of Dolosoft enables a user to perform all ten tasks previously listed with only three button clicks.

Dolosoft has a five additional buttons features to save the user time. The 'Cycript' button opens an interactive Cyript shell environment, shown in Figure 3, to manipulate objects stored in memory in real time. The 'SSH session' opens a SSH session in terminal to interact directly with the device. The 'Respring' button makes the iOS device "respring", which restarts the device. The 'Kill app' button kills the process of the target app. Because the code is only injected when the app first starts, both the 'Respring' and the 'Kill app' buttons are used when new code needs to be injected into the target app but the app is still open. These commands cause the app's process to be killed allowing for the new code to be injected. The last button is the 'Strings' button which displays all the strings contained inside the binary.

4 Results/Observations

Performing static analysis of iOS is made feasible because of modern disassemblers. With the abundance of graphics and diagrams produced, the static analysis that these

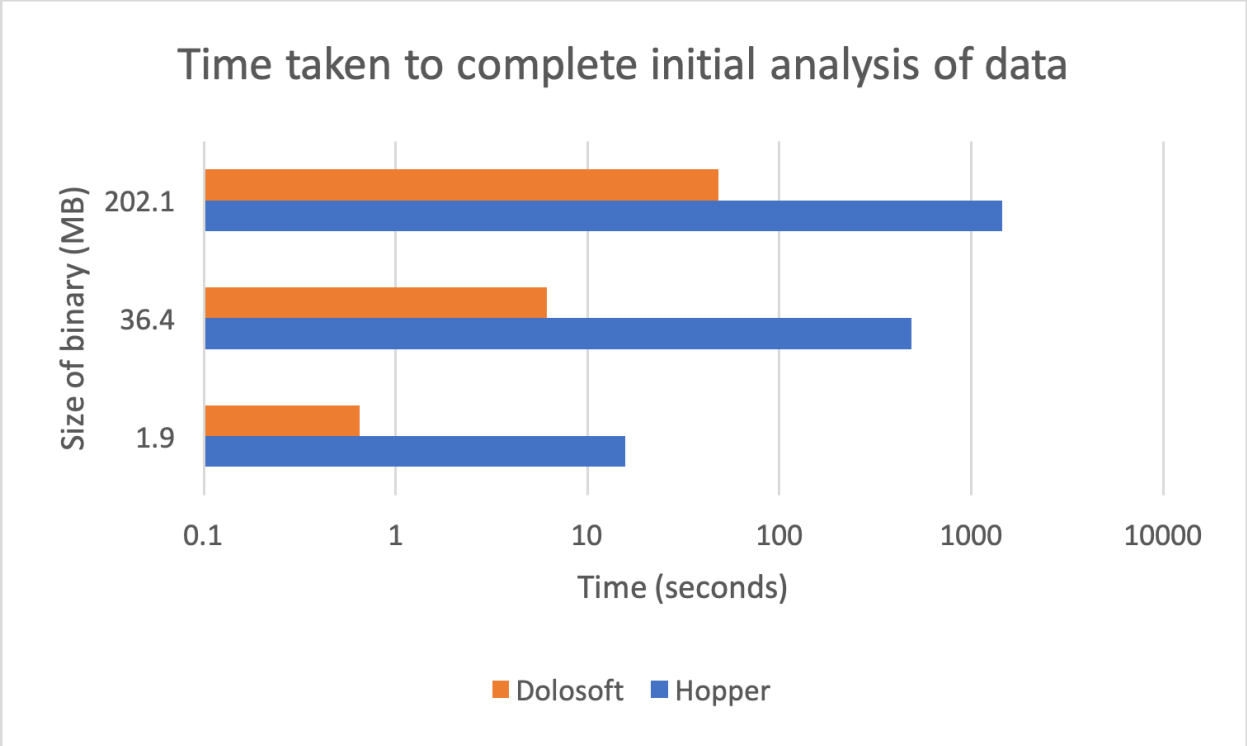

Fig. 4. Time taken to complete the initial processing and analysis of data by Dolosoft vs. Hopper tested on an Intel Core i7 CPU @ 3.3 GHz.

disassemblers perform is a resource intensive task that can push the limits of high end personal computers. I tested the performance of an established disassembler, Hopper, that supports parsing Objective-C metadata from app binaries against my own reversing software, Dolosoft. iOS app binaries of size 1.9 MB, 36.4 MB, and 202.1 MB were processed by each program to compare the performance of the programs.

First, I tested the time it took for the program to finish its initial analysis. This is important because with detailed disassemblers like Hopper and IDA, the computer running the disassembler is drained of its resources so the performance of other tasks can be slowed down. Reverse engineering is a time-consuming process, and every second that is removed from the process allows for more time to focus on the important jobs. The time taken by Hopper and Dolosoft to complete the initial analysis of the data for various sizes of binary files is presented in Figure 4 (note that the x-axis is scaled logarithmically). It is apparent that Dolosoft completes its analysis magnitudes (up to 70x) faster than Hopper. Dolosoft is able to do this by prioritizing what is necessary and opting out of long, unhelpful analysis techniques.

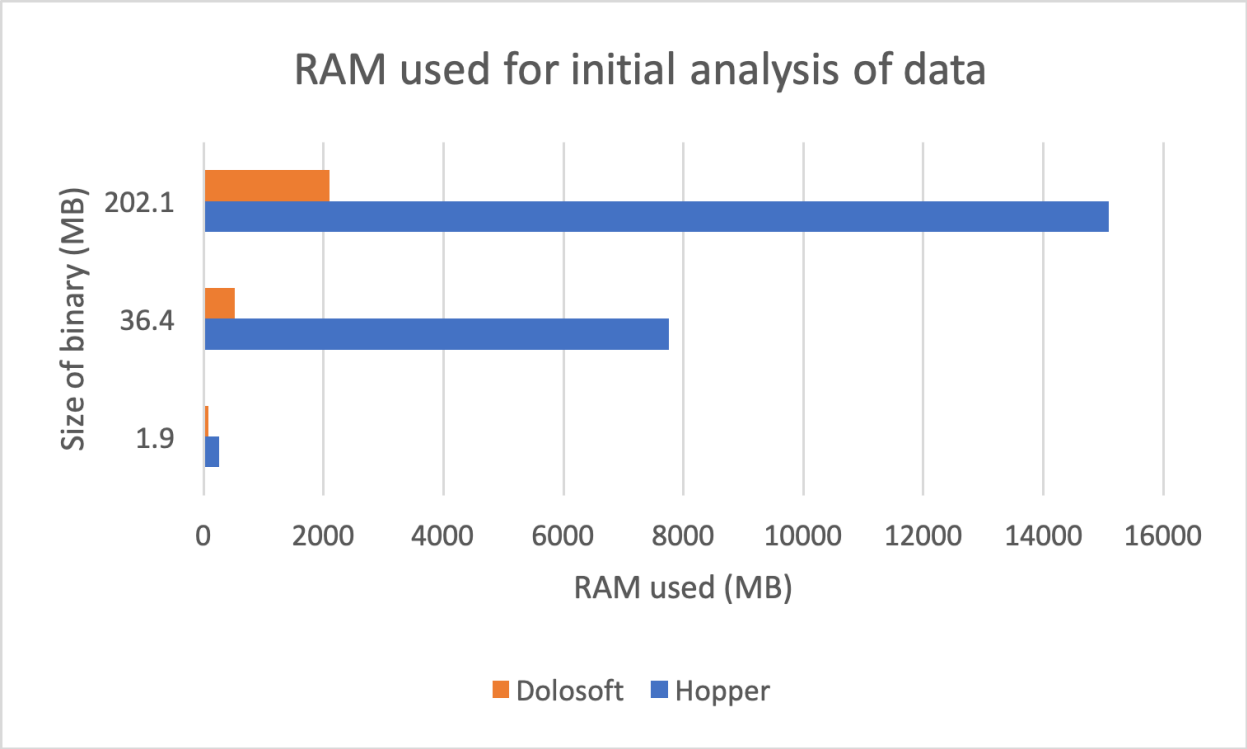

Fig. 5. RAM used to complete the initial processing and analysis of data by Dolosoft vs. Hopper tested on a computer with 16 GB of 2133 MHz LPDDR3 RAM.

Second, I took a look at memory usage. Dolosoft was designed to work on less powerful computers, so being efficient and resourceful with memory was a must. Only the information relevant for reversing and attacking the target app are saved which allows for a low amount of memory to be used. I compared the amount of RAM used by each

program when performing their initial analysis. Exhibited in Figure 5, Dolosoft used significantly (up to 15x) less RAM than Hopper to perform its analysis.

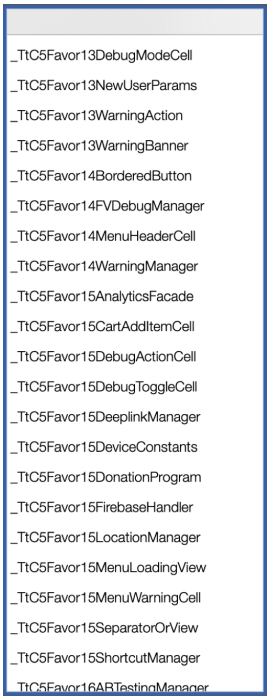

Fig. 6. Mangled class names of Swift classes.

Fig. 7. Properly formatted class names of Objective-C classes.

The functionality of Dolosoft was the most important factor throughout its development lifecycle. Dolosoft was tested on multiple applications written in completely Swift, completely Objective-C, and a mix of Objective-C/Swift. As expected, it performed best with apps written in pure Objective-C as it is easier to analyze these Mach-O binaries. There are more tools and documentation written for Objective-C provided it was released 30 years before Swift. It could decrypt all methods of classes written in Swift; however, the names of the classes were mangled as shown in Figure 6. The properly formatted class names of Objective-C classes are added for reference in Figure 7.

To test the attacking side of Dolosoft, I considered the Google Chrome browser app on iOS. I picked Google Chrome for a couple of reasons. One, Google is a worldwide technology company so security is not something they overlook when developing an app that millions will use. Two, the Google Chrome app is not a particularly small one with a download size of 70 MB. Three, it is a widely-used web browser on iOS, so this attack could be used in a real-world scenario.

After processing the Google Chrome app, I looked through the class list. There was a class called `JsPasswordManager` that was interesting. I decided to look at the methods owned by `JsPasswordManager` and noticed a method with the following declaration

```
- (void)fillPasswordForm:(id)arg1 withUsername:(id)arg2 password:
(id)arg3 completionHandler:(id)arg4;
```

Looking at the declaration syntax, it seemed that this method was responsible for automatically entering a user's password. The password should be passed through the `arg3` parameter. For example, when a user visits the Facebook login page, the Google Chrome app will automatically enter in said user's login information. I selected to log the method so that I could see the arguments passed. I loaded the tweak onto my iOS device

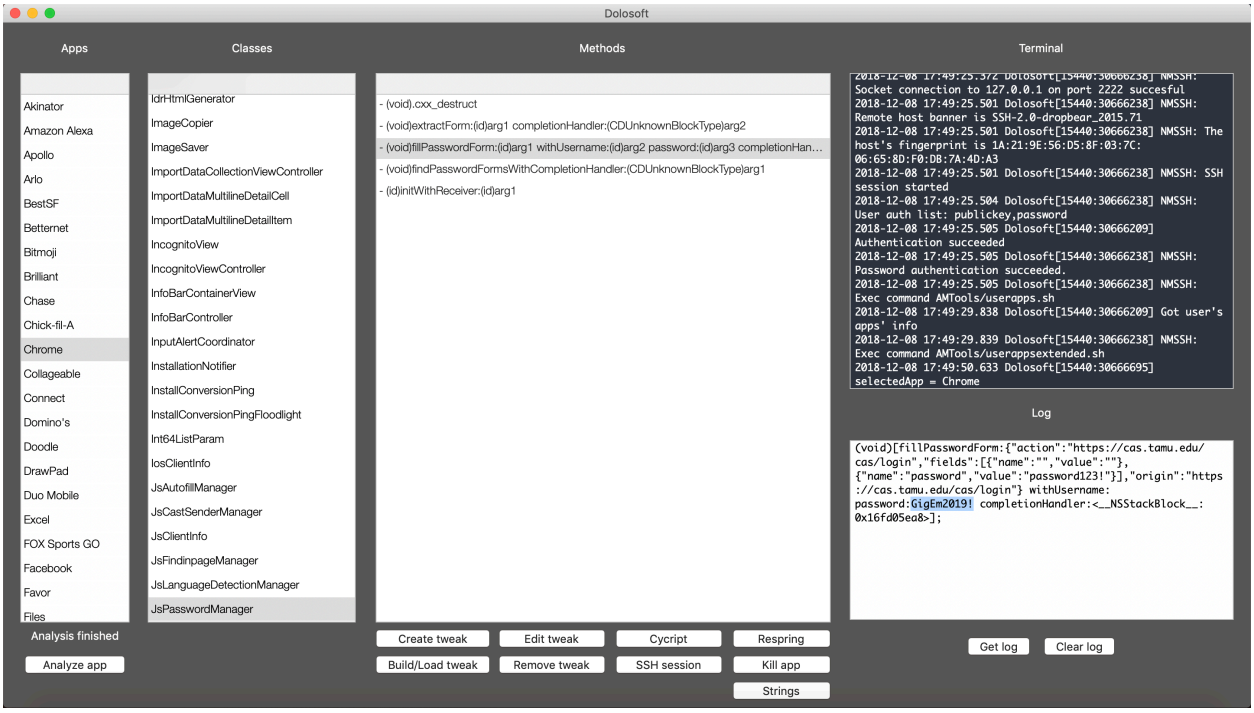

Fig. 8. A screenshot of Dolosoft where the login password was successfully logged.

and opened Google Chrome with the code now injected. Next, I went to Texas A&M's login page where Google Chrome automatically entered my login information. The resulting log, seen in Figure 8, shows that Dolosoft logged the arguments passed into the `fillPasswordForm` method, revealing my password. If this were a real attack, I would modify the generated code to upload the log to a server where I would view it. The problem with this attack is that the victim's iOS device must be jailbroken. There are many jailbroken devices where this attack would work as is, but majority of iOS devices run stock firmware. Ideally, this attack would be paired with a remote jailbreak exploit where physical access to the device would not be needed and the attack would run in the background. The result would be the victim having every password that is automatically entered with Google Chrome stolen. This is one of many possible attacks that can be carried out using Dolosoft.

5 Discussion/Recommendations

The best way to prevent an iOS app from being dynamically analyzed and attacked by programs like Dolosoft is to write iOS apps in Swift. As previously stated, Swift does not support method swizzling so swizzle based dynamic analysis is not possible. The only type of dynamic analysis possible would be using `gdb` on iOS which is not nearly as easy or effective. Method swizzling is by far the fastest, easiest way to dynamically analyze an iPhone app so this make the reverse engineer's job more challenging and time-consuming. The app can still be statically analyzed similar to an app written in Objective-C, but the function names will most likely be mangled depending on what program is being used. The reason the function names are mangled is because the tools necessary to demangle the names are still in their early stages and are not fully reliable like the tools analyzing for Objective-C based apps. Eventually these disassemblers will be able to demangle the names, but for now this is not the case.

Attacks based on dynamic code injection, like the one demonstrated with Google Chrome, are not possible with compiled Swift binaries as they do not support method swizzling. If parts of an app must be written in Objective-C, it is possible to add methods to check for jailbreaks or code injection; however, the attacker can just swizzle the method and bypass the detection. The best way to defend against method swizzling is to place the code (to detect jailbreaks or injection) inside of a method that is necessary for the app to function properly. For example, Facebook should put their detection code inside of the method that logs the user in. If the user swizzles this method to bypass the detection and changes the implementation of the method performs the check, then the login method will no longer work properly and the user will not be able to log in. The app will not function as it is supposed to, causing the dynamic analysis to fail.

Customizing the clang-llvm compiler, the compiler used to compile iOS app code, is another option that would work regardless if the code is written in Objective-C or Swift. One could customize clang compiler to obfuscate the method names during compile time so the source code would still be readable but the metadata pulled from the resulting binary would be unreadable. Again, this is just obfuscation which can always be defeated, but it would still disorient the attacker.

6 Conclusion

In this paper, I presented Dolosoft, software that fully automated dynamic analysis and attacks on iOS applications. I complemented static and dynamic analysis to implement software that can steal information from an app without any code needing to be typed. The automation that was designed turned a traditional 20 minute task to taking only a few minutes at most. Dolosoft is simple enough for a beginner to understand and use yet powerful enough for a security researcher to master and craft clever code. At the beginning of 2019, Dolosoft will be open sourced so that the iOS research community can use, learn from, and improve Dolosoft.

7 References

[1] Beschizza, R. “iPhone game dev accused of stealing players' phone numbers (UPDATED),” Nov 5, 2009. [Online] Available: https://boingboing.net/2009/11/05/iphone-game-dev-accu.html

[2] NSHipster. “Method Swizzling”, Feb 17, 2014. [Online]. Available: https://nshipster.com/method-swizzling/ [Accessed 30 Nov. 2018].

[3] Apple Inc. “Cocoa Core Competencies,” Apr 6, 2018. [Online]. Available: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Selector.html

[4] Apple Inc. “Objective-C Runtime,” Accessed Nov 30, 2018. [Online]. Available: https://developer.apple.com/documentation/objectivec/objective-c_runtime?preferredLanguage=occ

[5] Apple Inc. Accessed Dec 2, 2018. [Online] Available: https://opensource.apple.com/source/objc4/objc4-437/runtime/runtime.h

[6] Apple Developer. “method_exchangeImplementations,” Accessed Nov 30, 2018. [Online]. Available: https://developer.apple.com/documentation/objectivec/1418769-method_exchangeimplementations?language=objc

[7] Manning, T. “Reversing Objective-C Binaries With the REobjc Module for IDA Pro,” Duo Security. Accessed Dec 1, 2018. [Online]. Available: https://duo.com/blog/reversing-objective-c-binaries-with-the-reobjc-module-for-ida-pro

[8] 杨君. “Ida Swift Demangle,” Sep 4, 2016. [Online]. Available: https://github.com/tobefuturer/ida-swift-demangle

[9] Hex Rays SA. “Debugging iOS Applications With IDA,” 2016. [Online]. Available: https://www.hex-rays.com/products/ida/support/tutorials/ios_debugger_tutorial.pdf

[10] Freeman, J. “Cycript Manual,” Accessed Nov 27, 2018. [Online]. Available: http://www.cycript.org/manual/

[11] iPhoneDevWiki. “Cycript,” Accessed Nov 31, 2018. [Online]. Available: http://iphonedevwiki.net/index.php/Cycript

[12] Esser, S. “Dumpdecrypted,” Feb, 13 2014. [Online]. Available: https://github.com/stefanesser/dumpdecrypted

[13] Frugghi. “NMSSH,” Aug 16, 2018. [Online]. Available: https://github.com/NMSSH/NMSSH

[14] Apple. “Using Objective-C Runtime Features in Swift,” Accessed Dec 3, 2018. [Online]. Available: https://developer.apple.com/documentation/swift/using_objective-c_runtime_features_in_swift

[15] iPhoneWiki. “ASLR,” Oct 11, 2015. [Online]. Available: https://www.theiphonewiki.com/wiki/ASLR

[16] Kabiroberai, and Uroboro. “Theos,” Nov 22, 2018. [Online]. Available: https://github.com/theos/theos

[17] Freeman, J. "Installation Path," Accessed Dec 1, 2018. [Online]. Available: http://www.cydiasubstrate.com/inject/darwin/