UNIVERSITY OF AMSTERDAM

Automated embedding of dynamic libraries into iOS applications from GNU/Linux

Marwin Baumann
marwin.baumann@os3.nl

Leandro Velasco
leandro.velasco@os3.nl

RESEARCH PROJECT II

Supervised by:
Cedric van Bockhaven
Deloitte

July 21, 2017

Abstract

The need for mobile application security assessments has increased since more iOS applications are released every day. Multiple incidents have become public where applications violated the user's privacy, or provided functionality not allowed by the license agreement. Dynamic analysis of applications is used by security researchers because the source code is often not publicly available. Dynamic analysis enables monitoring the behavior of an application while it is being executed. A jailbroken iOS device is usually required in order to conduct the analysis. To overcome this limitation a dynamically linked library can be embedded into the target iOS application. This library enables the dynamic analysis for that application on a non-jailbroken device. This method is often used since jail-breaks for the latest iOS version are not always available. Moreover, it has become increasingly common that developers implement jail-break detection in their applications. The process of embedding dynamic libraries is mostly implemented by macOS native tools, is time consuming, and information about the inner workings of this process is scarce.

The work presented in this paper is aimed at overcoming these limitations by porting and automating this process from GNU/Linux. This would make security assessments more efficient and accessible to the open source community. Besides, by removing the requirement of macOS, security researchers could work with their preferred platform or run a GNU/Linux Virtual Machine. To accomplish our goal, we performed a theoretical analysis of the current state of the art, identified the different steps of the embedding process, and explored the way of implementing and automating each of the steps in GNU/Linux. In this report, we present an open sourced automated solution that we have developed and discuss about its applicability and limitations. We conclude that it is possible to automate from GNU/Linux the process of embedding a dynamic library into an existing iOS application.

Contents

1 Introduction

All iDevices (e.g. iPhone, iPod Touch, and iPad) can access and download applications from the App Store. Apple scrutinizes each submitted application, but not always sufficiently. Multiple incidents have become public in which applications violated the user's privacy [1] [2], or provided functionality not allowed by the license agreement [3]. Therefore, it is important for users and society to continuously monitor new applications.

Besides, with more mobile applications released every day the need for security assessments has increased [4]. In order to conduct such security assessment, dynamic analysis of an application is often used by security researchers. The reason is that the source code is not always available. Dynamic analysis enables monitoring the behavior of an application while it is being executed. Moreover, dynamic analysis is used to monitor the invocation of functions, track how data is propagated through the application, and to modify the behavior of the application [5]. It can also be useful for developers because this technique is capable of exposing subtle flaws in the source code.

To conduct the dynamic analysis of iOS applications two methods can be used. The first method consists in installing a special application on an iDevice which can monitor any application installed. The second method is to embed a dynamically linked library into the application to be monitored [6]. The first method is the easiest, but requires that a jailbroken iDevice is used. This restriction is imposed by the sandbox model of Apple [7] that enforces a mechanism in which applications are isolated from the rest of the system. The advantage of the second method is that no jailbroken iDevice is required. This method is often used since jail-breaks for the latest iOS versions are not always available, and developers increasingly implement jail-break detection [8]. In order to conduct the dynamic analysis using the second method the dynamic libraries provided by the Frida [9] and Cycript [10] project are often used. These dynamic libraries, known as "gadgets", are instrumentation tools that allow the inspection and debugging of applications during runtime [11].

In order to embed a dynamic library, four steps need to be taken as shown in Figure 1. Firstly, the iOS application needs to be extracted from the device. iOS applications are stored in the iOS Application Archive (IPA) format and are often encrypted and protected by Digital Rights Management (DRM). Therefore, the IPA file should be decrypted and the DRM needs to be removed before continuing the process. The second step is to link the dynamic library to the application and to rebuild it. The third step consists of re-signing the application because modifying the application makes the original signature invalid. It is possible to re-sign the application by a different person than the original developer. The only requirement is a provisioning profile which can be obtained from Apple using a valid Apple account. Finally, the signed IPA is pushed back to the device and then the application can be used.

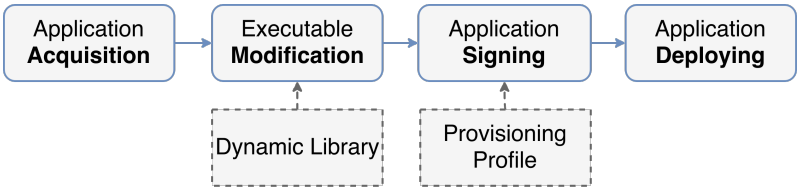

Figure 1: Overview of the process of embedding dynamic libraries into iOS applications

Due to the closed nature of Apple's ecosystem, most of the steps in this process can only be executed using a computer running macOS [8]. This limitation oblige security researchers to use an Apple computer when performing security evaluations on iOS applications. One way to overcome this limitation would be to run macOS in a Virtual Machine (VM), however there

are strict limitations to run macOS on non Apple hosts [12]. Using GNU/Linux would be more convenient, as this operating system does not have legal constraints to run as a guest in a VM. Moreover, GNU/Linux distributions are often chosen by security researchers because they are open source and because they provide a comprehensive set of security assessment tools.

The whole process of embedding dynamic libraries into iOS applications is time consuming because it consists of many steps to be executed. Moreover, the inner workings of this process are poorly documented. Exploring ways to execute all of the mentioned steps on GNU/Linux in an automated fashion is needed in order to make mobile app security assessments more efficient and accessible to the open source community. Currently there are some tools available for GNU/Linux that implement re-signing and deploying of iOS applications. But whether these tools still function after embedding a dynamic library into the application is unclear. In addition, to what extent these tools can be automated is unclear. No tools are available that enable the embedding of the dynamic library into the application without the use of macOS. Therefore, the aim of this study is to explore the feasibility of porting and automating the process of dynamic library embedding into iOS applications from GNU/Linux.

1.1 Research Question

Our main research question is: ***Is it possible from GNU/Linux to automate the process of embedding dynamic libraries into iOS applications?***

To answer this question we will investigate in depth how the different steps of dynamic library embedding in iOS applications work and subsequently if it is possible from GNU/Linux to fully port and automate these steps.

1.2 Report Structure

In Section 2, we portray the related work done on this topic. In Section 3 we describe the approach of the project in order to answer the research questions. The different steps of the dynamic library embedding process are elaborated from Section 4 until Section 8. These sections start with an in depth explanation about the step and continue by describing the work done to port this step to GNU/Linux. In Section 9 we explain how to automate all the steps from GNU/Linux. Finally, in Section 10 and Section 11 the limitations and conclusion are presented.

2 Related Work

Embedding a dynamic library into an application is not a new concept. This technique is also used for other applications running on other operating systems such as GNU/Linux, Windows, and macOS [13]. For example, to record different classes of function calls, such as API calls to the Windows API, or system calls for Linux [5]. Since jailbroken devices are not always available, during the last years this technique has been developed and refined for applications running on iOS.

In October 2014, Jonathan Zdziarksi wrote an article [14] in which he explains how an attacker can obtain an iOS application, embed external code in the binary executable, and then install the patched application on a non-jailbroken device. These techniques can be leveraged to embed dynamic libraries into iOS applications. In February 2015, Carl Livitt wrote a series of articles [15] that expanded the work done by Zdziarksi. In his articles he covers the essentials of adding dynamic libraries to iOS applications and describes the tools required to execute this process on non-jailbroken devices. In October 2016, Adrian Villa published an article [8] in which he describes the procedure to enable instrumentation of iOS applications on non-jailbroken devices. This procedure consists of embedding the Frida dynamic library into an iOS application, re-sign the patched binary with a provision profile, and then deploy the modified application to the jailed device.

During the Black Hat conference of 2016, Nishant Das Patnaik presented a framework called Appmon [16]. This software suite was designed to monitor and inspect system API calls of native apps on macOS, iOS and android. To do so, it relies on the Frida project. For the case of non-jailbroken iOS devices, Appmon provides a way to embed the Frida gadget into the target application.

A limitation that the different aforementioned publications share is that a macOS system with the Xcode framework installed is required. Xcode is an integrated development environment used to write and compile software for macOS, iOS devices, the Apple Watch, and the Apple TV [17]. The embedding process is mostly implemented by Xcode and other Apple native tools and little is documented about the inner workings of them. Our work presented in this paper is aimed at overcoming these limitations by investigating how the different steps of dynamic library embedding in iOS applications work and try to porting and automating these steps from GNU/Linux.

3 Approach and Methods

In order to answer our main research question, many challenges need to be addressed.

Firstly, a theoretical analysis of the current state of the art needs to be performed. During this analysis the different steps and the requirements for dynamic library embedding into iOS applications will be identified. Moreover, a detailed investigation of the different file formats, procedures, and protocols involved in each of the steps will be done. For instance, the macOS executable file format and the iOS App Store Package (IPA) format will be covered.

Next, the internals of the iOS application's building process will be explored. This includes provisioning profile generation, code signing and application deployment. In order to get a better understanding of the internals, we will use common analysis techniques such as reverse engineering, source code review and network packets analysis.

Once the steps and requirements of the embedding process are clear, we will analyze the different possibilities to re-implement the process in GNU/Linux. This will be done by exploring tools already ported to Linux, porting when possible the tools that are only for macOS, or writing new tools. Finnaly, we will investigate how the embedding process can be automated. In the case that automation is feasible, a proof of concept will be implemented.

In order to verify the correctness of the process and the developed proof of concept, several tests will be executed. These tests will be executed using two iOS devices as explained in the following section. Also we will verify the proof of concept using different iOS applications.

3.1 Hardware Used

For this research project we will use two laptops and two iOS devices in order to execute our experiments. The first laptop will be a MacBook Pro running macOS 10.12. This system will be used to investigate the internals of the dynamic library embedding process. Moreover, this laptop will have the Xcode 8.3.3 framework installed and will be used to analyze the components and tools involved in the process. The second laptop will be running the GNU/Linux distribution Fedora 25. This laptop will be used to explore the tools that are already ported. Furthermore, on this system we will conduct the experiments needed to port the steps that are currently bound to macOS.

Finally, in order to verify that our experiments are successful, we will use two non-jailbroken devices: an iPad Mini 3 running iOS 10.2.1 and an iPhone 6s running iOS 10.3.2. In the event of discovering that it is not possible to execute a certain experiment from a non-jailbroken device, we have limited access to a jailbroken iPhone 6 running iOS 10.2.

4 Application Acquisition

As previously mentioned, the first step of the dynamic library embedding process is the acquisition of the iOS application. iOS developers can use two formats to publish or distribute their applications. These are the app bundle (.app) format and the iOS App Store Package (IPA) format [18].

The app bundle consists of resource files and at least one executable binary file. The resources are everything an application needs besides the code itself, for example, storyboards, images, and audio files. The executable binary file, contains the machine code and it conforms to the mach-o executable file format [19]. An app bundle always contains the following resources: the `Info.plist` file, the `Base.lproj` folder, and the `_CodeSignature` folder. The `Info.plist` file, also called the information property list file, lists the metadata of the application such as the bundle name, bundle version, and the requirements [20]. The `Base.lproj` folder contains all the storyboard (.storyboard) or XML Interface Builder (.xib) files in the development language [21]. These files store the visual representation of the user interface of an iOS application [22]. The `_CodeSignature` folder contains the signature of the app bundle [23]. This is done to guarantee the integrity of the files. Finally, if the application is not distributed through the app store, but for testing purposes an `embedded.mobileprovision` file is included in the app bundle [24]. This is the provisioning profile, that specifies the permissions of the application as well as the signer identity.

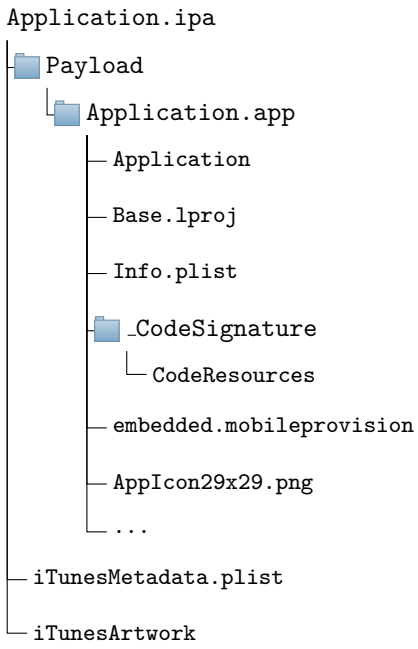

Figure 2: Tree structure of a general iOS App Store Package

The iOS App Store Package (IPA) is a compressed directory (ZIP archive) [25] containing the app bundle and additional resources needed for App Store services. The general structure of an IPA is shown in Figure 2. The IPA archive consists of the files `iTunesArtwork`, and `iTunesmetadata.plist`, and a `Payload` directory containing the app bundle. The `iTunesArtwork` file is a PNG image, containing the app's icon for showing in iTunes and the App Store [26]. The `iTunesMetadata.plist` file is used to provide extra information to iTunes about an iOS appli-

cation, such as genre, supported iOS devices, and required device capabilities. This file is not included in the IPA, if the app is distributed via ad hoc distribution, e.g. distributed for testing.

Apple uses two types of IPA; the universal and the thinned type. A universal IPA is a compressed app bundle that contains all of the resources to run the app on any device. In this case the executable file will contain multiple binaries with different ARM architectures, and is also called a "fat" binary in this case. A thinned IPA is a compressed app bundle that only contains the resources needed to run the app on a specific device type [27]. This optimization, called App Thinning, produces a IPA that only supports a single architecture and is therefore smaller. For App Store apps, the thinned IPA is downloaded to devices running iOS 9 or later and the universal IPA is downloaded to devices running iOS 8 or earlier. For example, when downloading an application on the iPhone 6 running iOS 10 the executable file will only contain the binary for the 64 bit ARMv8-A architecture [28].

Furthermore, the IPA is protected by the Digital Rights Management (DRM) technology called FairPlay [29]. When a user downloads an application from the App Store, this application is encrypted and signed by Apple. To protect the application even more, Apple injects a 4196 byte long header into the executable file within the application. This header is encrypted with the public key associated with the Apple account of the user [30]. When the application is installed the iOS device will decrypt the header with the private key of the user, which will succeed if the application was downloaded from the App Store with matching user credentials. This technique prevents users from installing the application on a device with an different Apple account associated.

4.1 iOS App Store Package Extraction

Apple provides users two ways of making a backup of their device; via iCloud or via iTunes. Using the second option makes it possible to extract the IPA files from this backup. A number of tools provide functionality to extract IPA files from iOS backups: `i-FunBox` [31], `iMazing` [32], and the archive functionality of `ideviceinstaller` [33]. All these tools can be installed on Windows and macOS, but the only one that can also be installed in GNU/Linux is `ideviceinstaller`. However, due to a change in the way applications are backed up in iOS 9 and higher, these tools can only extract the IPA files from backups from iOS 8 devices and lower.

Nowadays, only the user data is stored in the backup and not the applications themselves. When restoring the iOS device from a backup the installed apps are downloaded again and only the user data is being restored from the backup [27]. Our hypothesis is that this decision is made due to App Thinning and to improve the overall security [34] [35]. First, App Thinning would impose limitations in Apple's back-up system by preventing users from restoring applications in an iDevice with a different architecture. The second reason is that by only backing up the user data and not the application, when restoring the backup the last version of the app will be installed. Thus improving the overall security of the iOS ecosystem by avoiding the usage of outdated applications.

Since iOS 9 the only official way to obtain an IPA is through the "Download Purchased Application" functionality provided by iTunes [36]. Using this method a user can download purchased applications for backup purposes. Whereas on the iOS device the thinned IPA is downloaded and installed, using the iTunes functionality the universal IPA is downloaded. However, the universal IPA is still protected by Apple using FairPlay. Since, the IPA archive is encrypted and iTunes is not available for GNU/Linux another approach is needed to acquire the application.

To overcome the encryption limitation, the following process can be used [37] [38]. First the application needs to be installed and launched on an iOS device. When an iOS application is launched, the loader decrypts it and loads it into memory. In order to retrieve this decrypted version from memory the size of the payload needs to be calculated first. Next the memory loading address of the application needs to be found, the decrypted portion of the application

dumped using a debugger such as the GNU Project debugger, and then the encrypted area of the application executable overwritten with the dumped data. Note that only the executable file is encrypted by Apple, the resource files are never encrypted. Finally, the `cryptid` flag in the executable needs to be changed to 0. `cryptid` specifies if an application needs to be decrypted before loading into memory, when this value is 0 then iOS won't decrypt the application again. However, since normal users do not have enough rights to dump data from memory, a jailbroken device is needed to in order to execute this process.

The steps can be executed manually as described above, but are also automated by applications like Clutch [39]. Clutch can be executed on GNU/Linux, but requires a jailbroken device running iOS 8 or later. Clutch works by executing a binary on the iOS device which communicates to a host machine using SSH over USB. It hooks into the device runtime to dump the application from memory and into an unsigned application. By using Clutch we extracted five applications, using a jailbroken iPhone 6. We extracted a battery life app, two QR-code scanner apps, the 9292 travel app, and the Wikipedia app. See Section 9 for more details about the extracted applications and refer to the Appendix A for an overview of Clutch usage.

5 Executable Modification

An important step of the embedding process is the linking of the selected dynamic library (dylib) file to the main executable file. First the iOS executable file format needs to be studied, in order to know how and where to link this file.

The iOS app binary interface (ABI) uses the mach-o format as the standard to save binaries and libraries [19]. This file format consists of different regions each having a special purpose. When multi-architecture support is needed, developers can aggregate multiple mach-o files into a single executable file. These special binaries, called fat or universal binaries, contain a header that identifies the file as a fat file and indicates the amount of architectures contained in the binary. When an executable is compiled for a single architecture, this results in a thin binary file. This type of executable consist of a single mach-o file without extra headers. Figure 3 depicts a diagram of the mach-o file format.

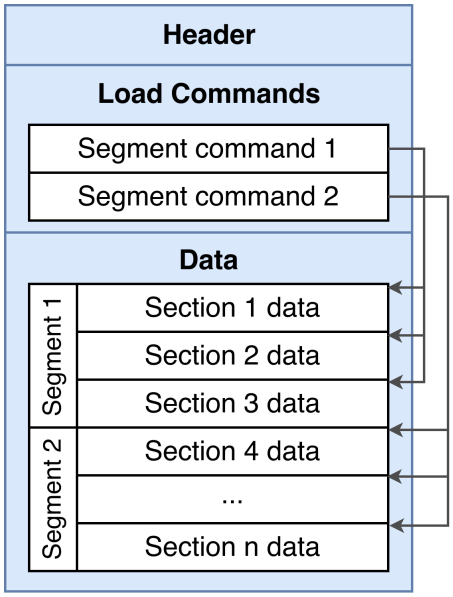

Figure 3: Mach-o file format [19]

The first region of a mach-o file is the header structure. This defines the file as a mach-o file and indicates the target architecture. Moreover, the header structure contains the binary flags and information about the file type (dylib, executable, bundle, etc). The fields that are more relevant for the embedding process are the `ncmds` and `sizeofcmds`. Where `ncmds` indicates the number of load commands, and `sizeofcmds` indicates the bytes occupied by the load commands region.

The second region is the load command region. Here, the layout and linkage properties of the mach-o file are defined. Every load command starts with the same two fields, the `cmd` that specifies the command type and the `cmdsize` that indicates the size of the command data. Depending on the command type, the rest of the command structure and total size can vary. An example of a common load command is the `LC_LOAD_DYLIB` command. This load command is used to link an executable file with a dynamic library and consists of three fields. The command type which is `LC_LOAD_DYLIB`, the command size which is set to `sizeof(dylib_command)`, and a `dylib` data structure. This data structure specifies the attributes of the shared library that an executable file links against. Moreover, the dylib data structure will be accessed by the dynamic linker at runtime, to locate the shared library.

The last region is the data region. Here the code and data are stored as specified by the load commands. This region is organized in segments that, depending on the type, can contain zero or more sections. The exact number and layout of segments and sections is specified by the load commands and the file type defined in the header structure. An important segment that can be found in every executable file is the `__TEXT` segment. This segment contains executable code and other read-only data. Additionally, it can include sections such as `__const` and `__cstring` where initialized constant variables and strings are placed.

To embed a dynamic library into an already existing binary, one needs to manipulate the mach-o file. This is done by inserting a LC_LOAD_DYLIB command into the load command region. Since the amount of commands changes, the `ncmds` and `sizeofcmds` values in the header structure are no longer valid. To fix this issue, during the mach-o manipulation, these attributes need to be recalculated.

5.1 Practical Implementation

The aforementioned steps required to embed a dynamic library are implemented by a series of open source projects. These are `node_applesign` [40], `optool` [41], and `insert_dylib` [42]. However, these three project are only implemented for macOS. Since the most popular one is `insert_dylib`, we decided to analyze it and then port it to GNU/Linux.

The `insert_dylib` project provides a command line tool for inserting the "dylib load command" into mach-o files. It does so by adding to each mach-o in a fat binary file a `LC_LOAD_DYLIB` load command to the end of the load commands region. Then it increments the mach-o header's `ncmds` and adjusts its `sizeofcmds`. Furthermore, since after the modification of the mach-o the binary signature is no longer valid, `insert_dylib` provides a mechanism to strip the code signature blob.

In order to port `insert_dylib` first we analyzed the code to identify the operations that were bound to macOS. The result of this analysis indicated that the macOS exclusive components of the code were the mach-o declaration headers and the *copyfile* function.

To implement the file manipulation needed to embed a dynamic library, `insert_dylib` requires to be aware of the internal structure of the mach-o file format. This is done by including in the program the header files *fat.h* and *loader.h* that declares the mach-o structure. Although these headers files can be downloaded from the Apple web site [1] as they are open source, not all the declarations are included. When we downloaded from Apple all the available headers and tried to compile the software in GNU/Linux, errors regarding missing declaration headers were raised. To overcome this problem, we downloaded the header files provided in the cctools GitHub project [43] and included these at compilation time. The cctools open source project provides to the GNU/Linux community a means to cross compile macOS binaries. This includes all the mach-o declarations needed to port `insert_dylib` to GNU/Linux.

Finally, since the *copyfile.h* was not included in the GitHub cctools project, the *copyfile* function could not be compiled. This issue was solved by re-implementing the functionality using GNU/Linux native operations.

Once the mach-o declaration was fixed and the copyfile function was re implemented, we were able to successfully compile `insert_dylib` on GNU/Linux. The work done to port this tool can be downloaded from our GitHub repository [2].

[1] `https://opensource.apple.com/tarballs/cctools/`
[2] `https://github.com/LeanVel/insert_dylib`

6 Re-signing the iOS App Store Package

The third step of the embedding process is resigning the IPA archive. This is needed because in the previous step the signature was invalidated by modifying the executable binary and adding a new dynamic library file to the IPA archive. Apple's code signing mechanism is mandatory and is used by iOS to verify an application's integrity and the developer's identity [23]. If an application does not pass the signature verification, the iOS kernel will prevent execution of the application [44]. This ensures that the code was not tampered with between the release of the application and the installation of it.

When developers or organizations want to install their own applications without using the App store, they need to sign the application and install the corresponding provisioning profile on their devices before the application can be started. A provisioning profile identifies the developer as a signer and indicates to iOS that applications signed by that developer are allowed to run on the device. This is not needed for applications in the App Store because those are already signed by Apple. The procedure of generating a provisioning profile is elaborated in Section 7.

When signing an application Apple distinguishes three type of components, the nested code, the main executable, and resources. The nested code are all the helper tools, dynamic libraries, plug-ins, frameworks, and other code that the main executable depends on. Everything in an application bundle that is not explicit code (nested code or the main executable) is a resource. Depending on the type of the component, the signature will be generated and stored in a different way. First the nested code is recursively signed. This is done by signing the corresponding files at the deepest level of the dependencies tree and then continuing upwards. This is because, the signature of a nested code file is used while signing a file higher in the dependencies tree. The result of this process is then stored in the file `_CodeSignature/CodeResources` within the IPA archive. Next, all the resources are individually signed and the signatures are stored alongside the nested code signatures. Finally, the main executable file is signed. However, the way this signature is generated differs significantly from the rest of the files.

For each mach-o file embedded in the main executable file, the signature is calculated and stored within each mach-o file using the `LC_CODE_SIGNATURE` load command [45]. This mach-o signature consists of a series of blobs, each having a special purpose. The first blob is the "Code Directory", in this directory the hashes of all the file pages are stored in individual slots. Moreover, the auxiliary data hashes such as the entitlements, and resource directory are added as special slots within the directory. Then, the signature has the "Requirement set" blob. This blob contains statements that will be used by iOS at the moment of verifying whether the code is validly signed and satisfies the constraints of the requirement [46]. The next blob is the "Entitlement" blob, which consists of the entitlements granted to the signed executable file. This blob will be used by iOS to decide whether to grant access to system resources. The last blob is the "Blob wrapper" where the signature of the aforementioned blobs is stored together with the corresponding certificates. Listing 1 depicts an example of the code signing blob embedded in an executable file.

```
Blob at offset: 6805776 (38521 bytes) is an embedded signature of 38521 bytes, and
    4 blobs
  Blob 0: Type: 0 @44: Code Directory (33431 bytes)
    Version:     20200
    Flags:       none (0x0)
    CodeLimit:   0x67d910
    Identifier:  com.rbtdigital.Battery-Life (0x34)
    CDHash:      8d7546dea0858cd773bb57b50542cf6923b6c39b (computed)
    # of Hashes: 1662 code + 5 special
    Hashes @191 size: 20 Type: SHA-1
      Entitlements blob:      78560f9a3aad787ebc5d177e2da8a87fc9bcd1ab
      Application Specific:   Not Bound
      Resource Directory:     3cb738270c6d2ab4d46469454eeb1146147a4d7c
      Requirements blob:      c56c48072543843ba0d272cc635d4b8bbc3c5f14
      Bound Info.plist:       817672dde672300d34c11dbe708b005dd277c939
      Slot   0 (File page @0x0000):   1f32c9fb0bef596de47db865d078783ba8327747
      Slot   1 (File page @0x1000):   958eafbf8bc9821c91f30f3206c2783a763e22ae
      ...
      Slot 1660 (File page @0x67c000): f33a2b2a4f8d669d21fda97056e4c66449c24900
      Slot 1661 (File page @0x67d000): 61c41b459b8774ee4d8942373484173d4c4ab3f7
  Blob 1: Type: 2 @33475: Requirement Set (208 bytes) with 1 requirement:
    0: Designated Requirement (@28, 168 bytes): SIZE: 168
       Ident: ("<APP␣Identifier>") AND Apple Generic Anchor
  Blob 2: Type: 5 @33683: Entitlements (474 bytes)
  Blob 3: Type: 10000 @34157: Blob Wrapper (4364 bytes)(CMS (RFC3852) signature)
    CA: Apple Certification Authority CN: Apple Root CA
    CA: Apple Worldwide Developer Relations CN: Apple Worldwide Developer Relations
        Certification Authority
    CA: Apple Certification Authority CN: Apple Root CA
    CA: Apple Certification Authority CN: Apple Root CA
```

Listing 1: Signature blob embedded in mach-o file as shown by Jtool

6.1 Code Signing Implementation

For iOS developers that use the Apple Xcode framework, the code signing mechanism is transparent. Xcode uses the tool `codesign` to perform the code signing needed. Many open source projects such as `node_applesign` [40], `iReSign` [47], and `Sigh` (part of the `fastlane` project [48]) implement code signing. However, most of these projects are built on top of the macOS native tool `codesign`. Nonetheless, there are two projects that implement Apple code signing without the usage of `codesign`.

The first project is Jtool [49]. This tool re-implements the functionality of a set of macOS native tools such as `otool`, `codesign`, `atos`, and `dyldinfo`. Since it is compiled for a variety of operating systems, including GNU/Linux, it was relevant to our research project. Although it implements some features of `codesign`, there are still some limitations in the way that code signing is done. The most important limitation is that it does not generate the "Requirement set" blob needed to evaluate the signature on the device. Moreover, it only supports signing of executable files. This means that for the nested code and the resources another tool should be used. Since the source code of this software is not available, we have raised a feature request to the developer about the empty "Requirement set" blob issue. This request was answered and in the next release of the tool the issue will be approached. Nevertheless, as this tool provides a clean view of the different blobs involved in the signature, it was a key component to understand the internals of the code signing process.

The second project is iSign [50]. This open source software is designed to re-sign iOS applications, without proprietary Apple software. This tool takes as the input an IPA file or application bundle and re-signs the complete application. In order to perform this, it requires that the signer's

private key, certificate, and provisioning profile are correctly set up in the host. Since this tool was implemented to re-sign applications, it does not support unsigned binaries. The way that iSign works is by overwriting the existing signature with the new one [51]. It reuses parts of the old signature such as the load command in the mach-o file that points to the code signature structure.

As explained in Subsection 5.1, after modifying the executable file we stripped out the existing signature as it became invalidated. This action produces an unsigned executable file that can not be signed by iSign. We overcame this limitation by using a recent fork of iSign [52] that is able to sign an executable file without reusing any pre existing structure, in other words it implements signing from scratch. Even though this version of iSign supports unsigned binaries, there was still a problem at the moment of signing the application plugins, also known as application extension (appex). An app extension, such as a widget, is used to provide extra functionality to an application. These are frequently used by iOS developers[53]. As a workaround this could have been solved by removing from the IPA the directory "Plugins". However, we troubleshooted the issue and contacted the developers. After providing error traces and further testing of the software, the issue was fixed.

7 Provisioning Profile Generation

As mentioned in Section 6, a valid provisioning profile is required in order to sign an application. A provisioning profile is a file signed by Apple that lists the certificates, devices, and the entitlements granted to applications [44]. When this provisioning profile is installed on one of the listed devices, it will present to the operating system the certificates that are allowed to sign executables.

To generate a provisioning profile many steps are needed which involve generating key pairs, issuing certificates, creating app IDs, and registering devices. There can be some limitations with the provisioning profile, depending on the Apple developer program membership [54]. Apple distinguishes three membership options. The first is the free membership, and this includes everybody with an Apple account that is not a developer. With a free membership it is possible to create a provisioning profile which has to be renewed every 7 days. Besides, the user has to provide a list of device UUIDs to which the provisioning profile will be deployed. The second membership is the individual developer. This membership requires a yearly fee, and allows the creation of provisioning profiles which expire in 365 days. The last membership is the enterprise developer. This paid membership enables companies to generate provisioning profiles that do not require a list of devices. Table 1 on the next page depicts an overview of the relevant differences between the memberships.

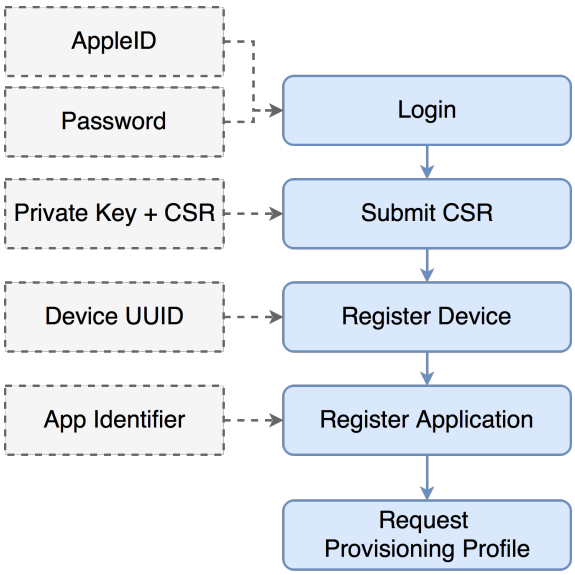

Figure 4: Overview of the provisioning profile generation process

An overview of the provisioning profile generation process is shown in Figure 4. The grey components represent the input requirements and the process itself is visualized in blue. In order to generate a provisioning profile, first the developer needs to authenticate to Apple using a valid AppleID and password. Next, a 2048 bit RSA key pair needs to be generated by the developer, which consists of a private key and a public key. The private key needs to be stored by the developer and is never sent to Apple. The key pair is then used to generate a Certificate Signing Request (CSR) [55]. A CSR is a combination of the public key and identifying information such as the organization name, common name (domain name), locality, and country. The CSR is then submitted to Apple which acts in this case as the Certificate Authority. This CSR is used by Apple to generate an identity certificate for the developer that proves the ownership of the public key. After the CSR has been submitted, the Universally Unique Identifier (UUID) of an iDevice needs

to be registered. These are the devices in which the provisioning profile will be installed. It is important to note that this is not necessary for enterprise developers, since this type of developer is not restrained to a limited amount of devices.

Next, the developer needs to generate an `AppID` value, in order to register the application. This can be done by using an explicit name or a "wildcard" name. When using a wildcard name the provisioning profile can also be used for other applications on the registered device. The wildcard functionality is only available for individual and enterprise developer accounts, free Apple users need to generate a new provisioning profile for every new application. Finally, the provisioning profile can be requested and used for signing.

Membership Type	**Expiration of provisioning profile**	**Devices in provisioning profile**	**Access to Developer Portal**
Free Apple account	7 days	List of Devices UUIDs	No
Individual Developer	365 days	List of Devices UUIDs	Yes
Enterprise Developer	365 days	-	Yes

Table 1: Relevant differences between Apple developer subscriptions

7.1 Individual / Enterprise Developer account

As shown in Table 1 individual and enterprise developers have access to the Apple Developer Portal. This portal can be accessed via the web browser, and can be used to generate the provisioning profile as described in Figure 4. It is also possible to directly use the Apple developer API, used by the Apple Developer Portal and Xcode. We identified one open source project able to use these API directly using a HTTP client. The `fastlane` project [48] provides a Ruby library called `spaceship` which is able to do this. By using this library we developed a script called `genProvisioningProfileDev` which can automatically perform all the steps mentioned in Figure 4. First the API `https://idmsa.apple.com` is used to authenticate and get a valid session. Then the API `https://developerservices2.apple.com` is used to register devices. Submitting the CSR, registering the application, and requesting the provisioning profile is done using the API `https://developer.apple.com`. Currently, the script can only be used for individual and enterprise developer accounts, because `spaceship` can only handle accounts that are enrolled into a team with an active membership, thereby having an `TeamID`.

7.2 Free Apple account

No open source tools were identified which could be used to retrieve the provisioning file using a free Apple account. Moreover, the online developer portal is not available for users with a free account. The only tool besides Xcode, which is able to sign applications using a free Apple account is Cydia Impactor. Therefore, we explored the way Xcode 8.3.3 (the current version at moment of writing this report) and Cydia Impactor are able to generate the mobile provisioning file. In order to do so, we used the Burp suite to setup a proxy and monitor the incoming and outgoing traffic. In addition, we installed the Burp's Certificate Authority (CA) certificate as a trusted root in the computer, thereby performing a man-in-the-middle attack which allowed us to inspect the encrypted HTTP traffic.

Only when Xcode is opened for the first time, the user has to authenticate using a valid AppleID and password. To capture this moment we created a new AppleID and tried to authenticate for the first time. However, while the Burp proxy was enabled Xcode refused the authentication and

alerted that “an unknown error has occurred”. Since after disabling the proxy, authentication took place without issues, our hypothesis is that Xcode 8 uses certificate pinning during the authentication. Afterwards, we re-enabled the proxy and generated a provisioning profile through Xcode. This resulted in a number of calls to the Apple API `developerservices2.apple.com`. First the TeamID is obtained. Because Apple requires that every developer is part of a team, Xcode enrolled our test account into the Xcode Free Provisioning Program, and added it to a new team called “<firstname> <last name> (personal team)”. Next all active development certificates, and the currently associated applications are listed. If a new application needs to be associated, a new AppID is generated and is associated with the AppleID. Finally, the mobile provisioning file is downloaded using the TeamID and AppID. Since we could not capture the network traffic during the authentication process, it is unknown how the development certificates are generated and how Xcode enrols the new Apple accounts to the Xcode Free Provisioning Program.

In order to explore the missing information we also monitored the traffic of Cydia Impactor in the same way we monitored Xcode. Cydia Impactor uses the same API used by Xcode in order to register and retrieve the information needed to download the provisioning profile. First the user has to provide a valid AppleID and password. Using this information in combination with a static `AppIDKey` value, the authentication takes place via the `idmsa.apple.com` API. It is important to note that the AppIDKey value is bound to the application that is authenticating to the API, and is not the identifier of an iOS application. In this case the application is Cydia Impactor. The result of the authentication procedure is a 575 character string called `myacinfo`. This string is subsequently used as an authentication token each time a message is sent to the `developerservices2.apple.com` API. This API is used for submitting the CSR, registering the device, registering the application, and retrieving the provisioning profile. Listing 2 depicts an example of a HTTP header sent to the `developerservices2.apple.com` API. This listing shows that Cydia Impactor presents itself as a Xcode version 7.0 client.

```
Host: developerservices2.apple.com
Content-Type: text/x-xml-plist
X-Xcode-Version: 7.0 (7A120f)
Cookie: myacinfo=DAWTKNV2b444b2323aa07dff9d559ae5fa86b63abfa653e889280ce69...
Accept-Language: en-us
Accept: text/x-xml-plist
Content-Length: 1145
Connection: close
User-Agent: Xcode
```

Listing 2: Example header of a captured Cydia Impactor HTTP packet while generating a mobile provisioning profile

By using the packets captured from Xcode and Cydia Impactor we implemented a bash script called `genProvisioningProfileFree` that mimics the provisioning profile generation process for free apple accounts. This helper script uses the AppleID and password of the user, in combination with the UUID of the iDevice to request the provisioning profile.

The helper scripts implemented for both developer and free Apple accounts can be downloaded from our GitHub repository [3].

[3] `https://github.com/LeanVel/iInject`

8 Install the iOS App Store Package

The final step of the embedding process is installing the modified IPA on the iDevice. In order to provide communication between the host machine and an iDevice, generally iTunes is used. This software can be run on a macOS or Windows machine, which is in charge of establishing the connection by means of the `lockdown` protocol. The `lockdown` protocol, provides pairing, activation, FairPlay certificate handling, and delegates communications to other services [56] [57]. `lockdown` runs on port 62078 and can accept connections via USB or via WiFi over TCP.

When connecting the iDevice via USB, the general USB protocol is used to provide generic access to the iDevice. On top of this protocol the `USBmux` protocol provides the multiplexing of several TCP connections over one USB pipe [58] [59].

After the `lockdown` service establishes the pairing, other protocols are used to provide access to different areas of the iDevice. For example the `AFC` (Apple File Connection) protocol can be used to exchange files between the iDevice and iTunes. Another example is the `installation_proxy` protocol, which is used to install and list applications. All the protocols are run as daemons on the iDevice and a client program is used on the host machine to connect with the corresponding daemons. Jonathan Zdziarski has published an overview of the known protocols involved [56].

8.1 Deployment Implementation

Besides iTunes itself, the aforementioned protocols are also implemented by the libimobiledevice open source project [60]. Libimobiledevice is a cross-platform software library that supports Windows, macOS, and GNU/Linux. This library allows access to the filesystem, and can be used to retrieve details about the connected device, create backups, manage installed applications, and synchronize music, video's, address books, calenders, notes, and bookmarks from and to the iDevice. An shown in Figure 5, the aforementioned protocols are implemented by the libimobiledevice library.

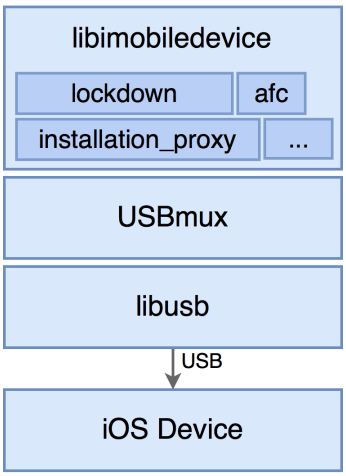

Figure 5: Overview of communication between iDevice and Host Machine

The libimobiledevice library only implements the protocols. In order to make use of the protocols provided by libimobiledevice, other tools are provided by the project which are built upon this library. For instance, to install, upgrade, uninstall and enumerate installed applications `ideviceinstaller` is used. This tool interacts with the `installation_proxy` daemon of an iOS device.

Another well known tool for installing applications is Cydia Impactor [61]. This is a cross-platform tool that supports Windows, macOS, and GNU/Linux. The main differences with ideviceinstaller are that Cydia Impactor is closed source, provides only an Graphical User Interface (GUI), and can also sign applications besides installing. By running the Linux `strings` command on the binary of Cydia Impactor, it was possible to see that Cydia Impactor also uses the libimobiledevice library in order to communicate with the iDevice and to install applications. Even though Cydia Impactor can sign applications, it does so using "Entitlements" that do not allow application debugging. If an application does not need to be started in debugging mode, this is not an issue, but this may impose a severe limitation for applications that need to be started in this mode. An example of a scenario where debugging mode is needed is when the Frida Gadget dynamic library is embedded in a application. Although the Gadget will be loaded when the application is started, to begin the instrumentation of the application the Gadget first needs to attach to it. If the application is not started in debugging mode, iOS will not allow any process to attach to the application. Since the goal of our project is to embed dynamic libraries like the Frida Gadget, Cydia Impactor is not a suitable solution. Therefore we use ideviceinstaller as the solution to install the modified IPA to the iDevice.

8.2 Running the modified application

Before an application can be run, the provisioning profile needs to be deployed to the iDevice. The tool ideviceinstaller will automatically install this profile, which makes it possible for iOS to run the security checks needed before an application will run. If the installation goes well, the app should launch on the device by tapping the application icon.

As mentioned before, some dynamic libraries, such as the Frida dynamic library need to be started in debug mode [62] [8]. In order to activate this mode, the debugging symbols need to be loaded first on the iDevice. These can be loaded by mounting the developer disk image for the right iOS version. The `DeveloperDiskImage.dmg` file can be copied from any macOS system with Xcode installed. This file can be found under `/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/<iosversion>/`. This file can also be downloaded from various sources on the internet. After this file is retrieved, macOS is not needed anymore. On GNU/Linux, the tool `ideviceimagemounter`, part of the libimobiledevice project, can be used to mount this file to the iDevice. After this file is mounted the application can be started in debugging mode by using the tool idevicedebug, also part of the libimobiledevice project.

9 Automation

In order to answer our last research question, we developed a proof of concept tool which we call `iInject`. This command line tool takes as input an IPA file and a dynamic library file. The tool performs the application modification, code signing, and deployment in an automated fashion. In Figure 6 we present a diagram of the inner workings of iInject. Since it is not possible to acquire the application from a non-jailbroken device, this step of the process in not included in iInject. Therefore, an unencrypted IPA is required as part of the input.

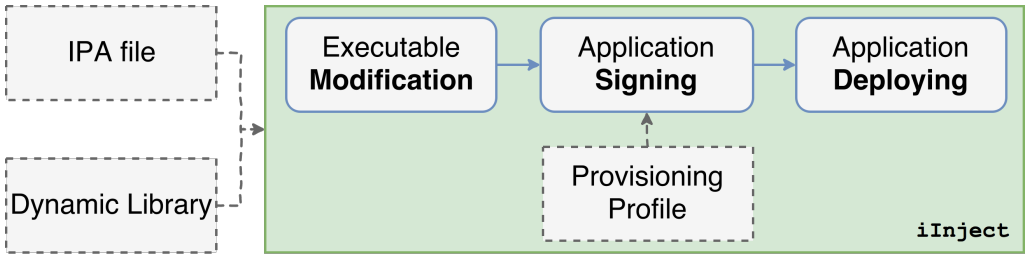

Figure 6: Diagram of the input requirements and inner working of iInject

First of all, iInject verifies if a valid signing identity and provisioning profile is set up on the host system. In other words, it checks if there is a valid private key with the corresponding certificate signed by Apple, and it verifies that the provisioning profile is not expired and includes the target device in the device list. If this check fails, iInject will suggest to the user to generate a provisioning profile using one of the helpers scripts introduced in Section 7.

Once the provisioning profile is correctly set-up, iInject uncompresses the IPA archive in a work directory. After the IPA is correctly uncompressed, the tool copies the selected dynamic library file into the "<application name>.app" directory. Then, the program `insert_dylib` is called to insert the `LC_LOAD_DYLIB` load command into the application's executable. Since the modification of the executable invalidates the old signature, we use the "code signature stripping" feature of `insert_dylib` to remove the old signature from the binary.

The next step is signing the application with the installed provisioning profile. Before signing iInject creates a new IPA file by compressing the "Payload" directory. This directory by now contains a modified executable and the selected dynamic library file. When the new IPA is ready, the program `iSign` is called. As explained in Subsection 6.1, iSign signs the application's nested code, resources, and main executable file. Moreover, it adds to the IPA archive the "embedded.mobileprovision" file required by iOS to verify the code signature. This file is the provisioning profile used during the signing process.

Finally, iInject deploys the new IPA to the target device. This is done via the program `ideviceinstaller`. This program implements the protocols needed to communicate with the device, pushes the IPA file, and launches the installation proxy on the device to install the application. The installation instructions, as well as additional technical details of the tool, can be found in our GitHub repository [4].

To verify the correctness of iInject we have tested the tool using two non-jailbroken devices: an iPhone 6s running iOS 10.3.2 and an iPad mini 3 running iOS 10.2.1. Moreover, we used ten different IPAs during the tests. These applications were acquired from either a jailbroken iPhone running iOS 10.2 using Clutch or downloaded from the `iosninja.io` website [5]. `iosninja.io` provides IPA files without Fairplay protection.

[4] `https://github.com/LeanVel/iInject`
[5] `https://iosninja.io/ipa-library`

In Table 2 we present the BundleID of the applications, the origin of them, and the result of the tests. Both the IPAs that did not pass the test successfully, had problems in the code signing step. Our hypothesis is that some of the framework files is in a format that is not supported by iSign. Therefore, we have escalated the issues to the iSign developers to find out the root cause of the problem.

Name	**Application Bundle ID**	**Origin**	**Result**
BatteryLife	com.rbtdigital.Battery-Life	Clutch	Success
QR Scanner	com.wenstudio.free.f3.Scanner6	Clutch	Success
QR Free	com.ihandysoft.barcode.qr.free	Clutch	Failure
9292	nl.9292.9292	Clutch	Success
Wikipedia	org.wikimedia.wikipedia	Clutch	Failure
YouTube++	com.google.ios.youtube.noads	Web Site	Success
FilesBrowser	com.highcaffeinecontent.Files	Web Site	Success
Kodi 16 Ja	rvisorg.xbmc.kodi-ios	Web Site	Success
BatteryLifeApp	com.rbt.batteryLifeApp	Web Site	Success
FlappyBird	com.dotgears.flap	Web Site	Success

Table 2: Applications details and test results

10 Discussion and Future Work

In this project we have shown that the process of embedding a dynamic library into an existing iOS application can be performed from a GNU/Linux system in an automated way. Nevertheless, some limitations need to be addressed.

First, as explained in Subsection 4.1, to acquire the target IPA we need a jailbroken device or a non-jailbroken device running iOS 8 or lower. Currently, the last jailbreak was released in January 2017 for iOS 10.2, and this could be one of the last jailbroken iOS versions [63]. The reason for this is that it has become increasingly difficult to crack an iOS release, due to the security enhancements rolled out by Apple. Besides, even when a vulnerability is found which could be used for a jailbreak, this vulnerability is often sold to Apple or other high paying third parties. In the context of this project, we assumed that the security researcher interested in embedding a dynamic library has the means to acquire the IPA file. Hence, this limitation could also be overcome by contacting the developers of the application and retrieving the IPA this way.

Secondly, the scope of this project did not cover the development of dynamic libraries. This is not always an issue, since projects such as Frida and Cycript provide pre-compiled dynamic libraries. But, in order to develop and compile self-written dynamic libraries from a GNU/Linux system, further research needs to be done to port the full tool-chain for cross compilation.

Thirdly, we implemented a helper script called `genProvisioningProfileFree` that is able to retrieve the provisioning profile for free Apple accounts. Since we could not capture the authentication traffic from Xcode version 8.3.3, we mimicked the inner workings of Xcode version 7.0 in this script. If Apple decides to stop the support for Xcode 7.0, the script would not be functional anymore. For the script that we have developed to generate provisioning profiles using individual and enterprise developer accounts, this is not a limitation since it uses the Apple Developer Portal.

Furthermore, iInject and all the helper scripts were developed as a proof of concept. This means that to keep the implementation simple, we did not take all scenarios into account and we assumed that the requirements of our tool were properly fulfilled. However, we implemented basic checks to guide the user of the iInject tool to establish the right setup. Additionally, the tool can be improved to make the it more secure and efficient.

Although the tool was designed to embed any dynamic library into any iOS application, during the implementation of the tool only the Frida Gadget dynamic library was tested. Nonetheless, there are no indications that the tool would fail at the moment of embedding another dynamic library.

Finally, the tool was tested with two non-jailbroken devices running iOS 10.2.1 and 10.3.2. Since all the steps performed are independent from iOS 10, from a theoretical point of view iInject should work with lower iOS versions. During the development of iInject, ten different iOS applications were used to verify the behaviour of the tool. In order to improve the robustness of the tool, a more representative population of iDevices, iOS versions, and IPAs should be tested.

11 Conclusion

Our research focused on automating from GNU/Linux the process of embedding dynamic libraries into iOS applications. This process is mostly implemented by Apple native tools and little is documented about the inner workings of them. Therefore, to accomplish our goal, we performed a theoretical analysis of the current state of the art, identified the different steps of the embedding process, explored the way to implement each of the steps in GNU/Linux, and implemented a proof of concept that executes the steps in an automated fashion.

To begin with, we identified four steps in the embedding process: application acquisition, executable modification, application signing, and application deployment. Then, we studied in detail the different files and procedures involved in each of the steps. This includes the IPA archive structure, the mach-o binary file format, the code signing procedure, the provisioning profile generation, and the communication protocol between a host and an iOS device. The theoretical analysis revealed to us the requirements needed to embed a dynamic library into an already compiled iOS application.

After the theoretical analysis, we identified the tools that could implement each of the aforementioned steps. During this investigation many tools were found, however most of them were implemented for macOS. With the knowledge gained in the previous analysis, and by using additional open source projects, we ported and adapted the tools needed to implement from GNU/Linux all steps involved in the embedding process. Furthermore, by analyzing the network packets interchanged between Xcode and the Apple Developer Portal API, we identified the requirements and procedure to generate provisioning profiles without a macOS system.

Finally, we explored different ways of automating the complete embedding process. By leveraging the functionality of the tools identified in the practical investigation, we implemented a command line tool called iInject. This proof of concept takes as an input a target IPA and a dynamic library file and then performs the executable modification, application signing, and application deployment to the iDevice. Since the application acquisition requires a jailbroken device, we did not to integrate this step in the proposed automated solution. Thereby allowing iInject to work on non-jailbroken devices. To fine-tune the parameters passed to the underling tools we performed several tests on different iOS versions with different IPA files. Furthermore, we collaborated with the corresponding tools' developers to fix features and functionality needed for this project. In addition to implementing iInject, we developed two standalone scripts capable of generating a provisioning profile by just providing a valid free Apple account or paid developer account.

To conclude, the process of embedding dynamic libraries into iOS applications can be performed from a GNU/Linux system. We have shown with our proof of concept that this process can be automated and can be executed on non-jailbroken devices.

References

[1] Rob Beschizza. iPhone game dev accused of stealing players' phone numbers. https://boingboing.net/2009/11/05/iphone-game-dev-accu.html, November 2009. [Online; accessed 8-June-2017].

[2] Sean Hollister. Here's why we're not downloading Meitu, the red-hot anime photo app. https://www.cnet.com/news/meitu-app-privacy-issues-why/, January 2017. [Online; accessed 8-June-2017].

[3] Wired. Apple Approves, Pulls Flashlight App with Hidden Tethering Mode. https://www.wired.com/2010/07/apple-approves-pulls-flashlight-app-with-hidden-tethering-mode/, July 2010. [Online; accessed 8-June-2017].

[4] Android Open Source Project. Android Security 2015 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2015_Report_Final.pdf, April 2016. [Online; accessed 7-June-2017].

[5] Martin Szydlowski, Manuel Egele, Christopher Kruegel, and Giovanni Vigna. Challenges for dynamic analysis of ios applications. In *Open Problems in Network Security*, pages 65–77. Springer, 2012.

[6] Andreas Kurtz, Andreas Weinlein, Christoph Settgast, and Felix Freiling. Dios: Dynamic privacy analysis of ios applications. Technical Report CS-2014-03, Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, June 2014.

[7] Topher Kessler. What Apple's sandboxing means for developers and users. https://www.cnet.com/news/what-apples-sandboxing-means-for-developers-and-users/, November 2011. [Online; accessed 8-June-2017].

[8] Adrian Villa. iOS instrumentation without jailbreak. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2016/october/ios-instrumentation-without-jailbreak/, October 2016. [Online; accessed 7-June-2017].

[9] Ole André V. Ravnòs. Frida - A world-class dynamic instrumentation framework - Inject JavaScript to explore native apps on Windows, macOS, Linux, iOS, Android, and QNX. https://www.frida.re, June 2017. [Online; accessed 06-July-2017].

[10] Jay Freeman (saurik). Cycript - allowing developers to explore and modify running applications on either iOS or Mac OS X. http://www.cycript.org/, June 2017. [Online; accessed 06-July-2017].

[11] Ole André V. Ravnòs. Frida A world-class dynamic instrumentation framework. https://www.frida.re/docs/home/, July 2017. [Online; accessed 7-June-2017].

[12] Apple Support Community. Macintosh virtual machine hosted by Windows. https://discussions.apple.com/thread/5785112?tstart=0, January 2014. [Online; accessed 8-June-2017].

[13] Hammond, Richard P. Dynamic injection of execution logic into main dynamic link library function of the original kernel of a windowed operating system, October 2002. US Patent No 6,463,583.

[14] Jonathan Zdziarksi. How App Store Apps are Hacked on Non-Jailbroken Phones. https://www.zdziarski.com/blog/?p=4002, October 2014. [Online; accessed 7-June-2017].

[15] Carl Livitt. Rethinking & Repackaging iOS. https://www.bishopfox.com/blog/2015/02/rethinking-repackaging-ios-apps-part-1/, February 2015. [Online; accessed 7-June-2017].

[16] Nishant Das Patnaik. Appmon: runtime security testing & profiling framework for native apps. https://github.com/dpnishant/appmon/wiki/2.-Introduction, November 2016. [Online; accessed 06-July-2017].

[17] Apple Inc. Xcode - Apple Developer. https://developer.apple.com/xcode/, June 2017. [Online; accessed 8-June-2017].

[18] Apple Inc. Technical Q&A QA1795 - Measure Your App. https://developer.apple.com/library/content/qa/qa1795/_index.html, April 2017. [Online; accessed 26-June-2017].

[19] Apple Inc. OS X ABI Mach-O File Format Reference, 2009.

[20] Apple Inc. About Information Property List Files. https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/AboutInformationPropertyListFiles.html, June 2017. [Online; accessed 26-June-2017].

[21] Apple Inc. Internationalization and Localization - Managing Strings Files. https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPInternational/MaintaingYourOwnStringsFiles/MaintaingYourOwnStringsFiles.html, September 2017. [Online; accessed 26-June-2017].

[22] Apple Inc. Cocoa Application Competencies for iOS - Storyboard. https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html, September 2017. [Online; accessed 26-June-2017].

[23] Apple Inc. Code Signing Guide - Understanding the Code Signature. https://developer.apple.com/library/content/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html, September 2016. [Online; accessed 27-June-2017].

[24] Apple Inc. Creating Your Team Provisioning Profile. https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppStoreDistributionTutorial/CreatingYourTeamProvisioningProfile/CreatingYourTeamProvisioningProfile.html, April 2017. [Online; accessed 26-June-2017].

[25] PKWARE Inc. .ZIP File Format Specification. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT, October 2014. [Online; accessed 26-June-2017].

[26] Apple Inc. Technical Q&A QA1686 - App Icons. https://developer.apple.com/library/content/qa/qa1686/_index.html, December 2016. [Online; accessed 26-June-2017].

[27] Apple Inc. App Distribution Guide - App Thinning (iOS, tvOS, watchOS). https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/AppThinning/AppThinning.html, September 2017. [Online; accessed 27-June-2017].

[28] Empirical Magic Ltd. ios support matrix. http://iossupportmatrix.com/, June 2016. [Online; accessed 12-July-2017].

[29] Wikipedia. Fairplay — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=FairPlay&oldid=789684844, July 2017. [Online; accessed 12-July-2017].

[30] Josh Lieberman. How are Apple App Store Apps encrypted? https://reverseengineering.stackexchange.com/questions/14704/how-are-apple-app-store-apps-encrypted, February 2017. [Online; accessed 26-June-2017].

[31] Appholly Technology Co. iFunbox — the File and App Management Tool for iPhone, iPad & iPod Touch. http://www.i-funbox.com/, June 2017. [Online; accessed 06-July-2017].

[32] DigiDNA SARL. iMazing — iPhone, iPad & iPod Manager for Mac & PC. https://imazing.com/, June 2017. [Online; accessed 06-July-2017].

[33] Contributors of libimobiledevice. libimobiledevice/ideviceinstaller: Manage apps of iOS devices . https://github.com/libimobiledevice/ideviceinstaller, June 2016. [Online; accessed 06-July-2017].

[34] DigiDNA Team. Why did Apple kill App extraction in iOS 9? https://imazing.com/blog/why-did-apple-kill-app-extraction-in-ios-9, September 2015. [Online; accessed 27-June-2017].

[35] Jeffery Battersby. Why 'Transfer Purchases' doesn't copy apps to iTunes when backing up your iPhone. http://www.macworld.com/article/3125509/iphone-ipad/more-on-backing-up-your-ios-device.html, September 2016. [Online; accessed 27-June-2017].

[36] Apple Inc. Redownload music, movies, TV shows, apps, and books from the iTunes Store, iBooks Store, and App Store. https://support.apple.com/en-gb/HT201272, May 2017. [Online; accessed 26-June-2017].

[37] Infosec Resources. Penetration Testing for iPhone Applications âĂŞ Part 5. http://resources.infosecinstitute.com/penetration-testing-for-iphone-applications-part-5/, May 2016. [Online; accessed 27-June-2017].

[38] iPhone Development Wiki. iOS - Reverse Engineering Tool. http://iphonedevwiki.net/index.php?title=Reverse_Engineering_Tools&oldid=4970, April 2017. [Online; accessed 27-June-2017].

[39] Kim Jong-Cracks. KJCracks/Clutch: Fast iOS executable dumper. https://github.com/KJCracks/Clutch, June 2017. [Online; accessed 06-July-2017].

[40] NowSecure. nowsecure/node-applesign: NodeJS module and commandline utility for re-signing iOS applications. https://github.com/nowsecure/node-applesign/, June 2017. [Online; accessed 06-July-2017].

[41] Alex Zielenski. optool - Command Line Tool for interacting with MachO binaries on OSX/iOS. https://github.com/alexzielenski/optool, March 2017. [Online; accessed 06-July-2017].

[42] Asger Hautop Drewsen. insert_dylib - Command line utility for inserting a dylib load command into a Mach-O binary. https://github.com/Tyilo/insert_dylib, August 2016. [Online; accessed 06-July-2017].

[43] Fabian Renn. cctools - Apple toolchain ported to linux. https://github.com/hogliux/cctools, June 2014. [Online; accessed 06-July-2017].

[44] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. *iOS Hacker's Handbook*. John Wiley & Sons, 2012.

[45] Jonathan Levin. Code signing - hashed out. http://www.newosxbook.com/articles/CodeSigning.pdf, April 2015.

[46] Apple Inc. macos code signing in depth. https://developer.apple.com/library/content/technotes/tn2206/_index.html#//apple_ref/doc/uid/DTS40007919-CH1-TNTAG4, September 2016. [Online; accessed 12-July-2017].

[47] NowSecure. maciekish/iReSign: allowing iDevice app bundles files to be signed or resigned with a digital certificate from Apple for distribution. https://github.com/maciekish/iReSign, June 2015. [Online; accessed 06-July-2017].

[48] Fastlane contributors. Fastlane - iOS and Android Automation for Continuous Delivery. https://fastlane.tools/, July 2017. [Online; accessed 06-July-2017].

[49] Jonathan Levin. jtool - Taking the O out of otool. http://newosxbook.com/tools/jtool.html, July 2017. [Online; accessed 06-July-2017].

[50] Sauce Labs. saucelabs/isign: Code sign iOS applications, without proprietary Apple software or hardware. https://github.com/saucelabs/isign/, April 2017. [Online; accessed 06-July-2017].

[51] Neil Kandalgaonkar. isign issue #1 - saucelabs/isign. https://github.com/saucelabs/isign/issues/1, March 2016. [Online; accessed 13-July-2017].

[52] Mark Wang. ryu2/isign: Code sign iOS applications, without proprietary Apple software or hardware. https://github.com/ryu2/isign, June 2017. [Online; accessed 06-July-2017].

[53] Apple Inc. App extensions essentials. https://developer.apple.com/library/content/documentation/General/Conceptual/ExtensibilityPG/index.html#//apple_ref/doc/uid/TP40014214-CH20-SW1, November 2016. [Online; accessed 13-July-2017].

[54] Apple Inc. Choosing a Membership. https://developer.apple.com/support/compare-memberships/, 2017. [Online; accessed 08-July-2017].

[55] Wikipedia contributors. Certificate signing request — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Certificate_signing_request&oldid=769412577, March 2017. [Online; accessed 11-July-2017].

[56] Jonathan Zdziarski. Identifying back doors, attack points, and surveillance mechanisms in ios devices. *Digital Investigation*, 11(1):3–19, 2014.

[57] Mathieu Renard. Hacking apple accessories to pown iDevice. http://2013.hackitoergosum.org/presentations/Day3-04.Hacking%20apple%20accessories%20to%20pown%20iDevices%20%E2%80%93%20Wake%20up%20Neo!%20Your%20phone%20got%20pwnd%20!%20by%20Mathieu%20GoToHack%20RENARD.pdf, May 2013. [Online; accessed 28-June-2017].

[58] iPhone Dev Team. usbmux. https://www.theiphonewiki.com/w/index.php?title=Usbmux&oldid=49715, July 2008. [Online; accessed 28-June-2017].

[59] Héctor Martín Cantero. iPhone syncing on Linux. https://marcan.st/2009/10/iphone-syncing-on-linux/, October 2010. [Online; accessed 28-June-2017].

[60] Contributors of libimobiledevice. libimobiledevice - cross-platform software protocol library and tools to communicate with iOS devices natively. http://www.libimobiledevice.org/, April 2017. [Online; accessed 06-July-2017].

[61] Jey Freeman (Saurik). Cydia Impactor - GUI tool for working with mobile devices. http://www.cydiaimpactor.com/, March 2017. [Online; accessed 06-July-2017].

[62] Bernhard Mueller. Vantage point security - patching and re-signing ios apps. http://www.vantagepoint.sg/blog/85-patching-and-re-signing-ios-apps, February 2017. [Online; accessed 13-July-2017].

[63] Buster Hein. Jailbreaking pioneers say iPhone jailbreaking is dead. https://www.cultofmac.com/490594/jailbreaking-pioneers-say-iphone-jailbreaking-dead/, June 2017. [Online; accessed 10-July-2017].

Appendices

A Clutch

In order to obtain an unencrypted .ipa, Clutch can be used. This tool can be downloaded from the GitHub repository: `https://github.com/KJCracks/Clutch`. Although the building of the binary requires macOS, pre-built binaries are released along every major update: `https://github.com/KJCracks/Clutch/releases/latest`. Using the pre-built binary GNU/Linux can be used to execute all the required steps. In order to communicate with the binary `iProxy` can be used. `iProxy` is part of the `usbmuxd` project: `https://cgit.sukimashita.com/usbmuxd.git/` and is also forked to the `libimobiledevice` project: `https://github.com/libimobiledevice/usbmuxd`. `usbmuxd` (USB multiplexing daemon) is a socket daemon to multiplex connections over USB from and to iOS devices. `usbmuxd` can be build from source using the repository or can be installed using a package manager by installing the following two packages: `libusbmuxd.x86_64` and `libusbmuxd-utils.x86_64`. In order to retrieve an .ipa file, we executed the following steps:

1. Download or build the Clutch binary.
2. Install OpenSSH onto the iDevice (e.g. using Cydia).
3. Connect the iDevice to the host machine via USB.
4. Start the iProxy by running the following command in the terminal: `iproxy 2222 22`. This will forward all traffic from port 2222 to port 22 over USB.
5. Copy the Clutch binary to `/usr/bin/` on the device by running the following command in a terminal on the host machine: `scp /path/to/Clutch root@localhost:/usr/bin/`
6. Open another terminal and connect to the iDevice by running ssh: `ssh -p 2222 root@localhost`
7. As shown in Listing 3, Command `Clutch -i` lists the installed apps and shows their bundleID.
8. As shown in Listing 4, Command `Clutch -d <bundleID>` will retrieve the .ipa files of an app.

```
Host:/usr/bin root# ./Clutch-2.0.4 -i
Installed apps:
1:   QR Code <com.wenstudio.free.f3.Scanner6>
2:   9292 <nl.9292.9292>
3:   Battery Life: check device's runtimes <com.rbtdigital.Battery-Life>
4:   Termius - SSH Shell / Console / Terminal <com.crystalnix.ServerAuditor>
```

Listing 3: List of installed apps and their bundleID

```
Host:~ root# Clutch-2.0.4 -d com.wenstudio.free.f3.Scanner6
Zipping Scanner6.app
ASLR slide: 0x100010000
Dumping <Scanner6> (arm64)
Patched cryptid (64bit segment)
Writing new checksum
DONE: /private/var/mobile/Documents/Dumped/com.wenstudio.free.f3.Scanner6-iOS7.0-(
    Clutch-2.0.4).ipa
Finished dumping com.wenstudio.free.f3.Scanner6 in 2.6 seconds
```

Listing 4: Retrieving the .ipa of QR-code scanner app