Klaus Horn

Code Signing Android and iOS Applications

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Thesis

9 November 2018

Metropolia

Author Title Number of Pages Date	Klaus Horn Code Signing Android and iOS applications 37 pages + 5 appendices 9 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Lauri Alakuijala, Technical Manager, Rovio Entertainment Corp. Erja Nikunen, Principal Lecturer, Metropolia UAS

Code signing Android and iOS mobile applications is an integral part of the mobile application development process. It is needed for verifying the developer of an application and is also tightly linked to the installation of a mobile application on a device.

This bachelor's thesis aims to primarily provide a general handbook for introducing the process of code signing Android and iOS mobile applications. The information gathered in this thesis should be enough to develop a command line application that uses tools provided by both Apple and Google, as well as some open source ones.

Secondarily, this thesis outlines the development process of a proprietary Python library, which provides a unified interface for code signing both Android and iOS applications. The Python library was developed to replace an existing code signing service used by Rovio Entertainment Corporation.

Keywords	android, apk, code signing, entitlements, ios, ipa, keystore, zipalign, provisioning profile

Metropolia

Tekijä Otsikko Sivumäärä Aika	Klaus Horn Android and iOS sovellusten digitaalinen allekirjoitus 37 sivua + 5 liitettä 9.11.2018
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tieto- ja viestintätekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	Lauri Alakuijala, Technical Manager, Rovio Entertainment Corp. Erja Nikunen, Yliopettaja, Metropolia Ammattikorkeakoulu

Android ja iOS mobiilisovellusten digitaalinen allekirjoitus on keskeinen osa mobiilisovellusten kehitysprosessia. Digitaalista allekirjoitusta tarvitaan sovelluksen kehittäjän todentamiseen ja on myös oleellinen osa mobiilisovellusta asennettaessa laitteelle.

Tämän työn päätavoite on tarjota yleinen käyttöohje Android ja iOS sovellusten digitaalisen allekirjoitusprosessin perusteisiin. Työssä kerätyn tiedon pitäisi olla riittävä komentorivisovelluksen kehittämiseen käytettäessä Applen, Googlen ja avoimen lähdekoodin tarjoamia työkaluja.

Työn toissijainen tavoite on kuvailla Python kirjaston kehitysprosessia, jossa kirjasto tarjoaa yhtenäisen rajapinnan Android ja iOS sovellusten digitaaliseen allekirjoitukseen. Python kirjasto kehitettiin korvaamaan Rovio Entertainment Corporationin olemassa oleva allekirjoituspalvelu.

Työssä havainnollistetaan Applen ja Googlen ohjeissa ja työkaluissa olevia puutteita ja täydennetään puutteellisia tietoja tutkimuksen ja kokeilun kautta kerätyillä kuvauksilla ja selostuksilla. Python kirjaston kehitysprosessin kuvaus toimii oivallisena esimerkkinä, kuinka Android ja iOS mobiilisovellukset digitaalisesti allekirjoitetaan, ja kuinka muun muassa erinäisten prosessiosien todentaminen tehdään.

Python kirjasto kehitettiin ATDD menetelmää sivuten ja riippuvuusinjektiomallia hyväksikäyttäen. ATDD menetelmästä hyödynnettiin pääasiassa käyttötapauskuvauksia ja niiden vaatimuksia, joiden kautta pysyttiin ja jakamaan työtehtävät järkeviin kokonaisuuksiin. Riippuvuusinjektiomalli mahdollisti kirjaston, joka toimi pelkästään funktioiden kautta. Tulokseksi saatiin toimiva, selkä ja ylläpidettävä kirjasto, jota pystyy muokkaamaan ja täydentämään käyttötarpeiden mukaan.

Keywords	android, apk, code signing, entitlements, ios, ipa, keystore, zipalign, provisioning profile

Metropolia

Contents

List of Abbreviations

1 Introduction 1

2 Code signing 2

2.1 Certification Authorities 3
2.2 Issues with mobile application code signing. 4
2.3 Available third-party tools and their limitations 5

3 Android code signing 6

3.1 The Android application package 6
3.2 The keystore 8
3.2.1 Creating keystores 8
3.3 Key hashes and SHA1 signatures 11
3.4 Code signing an APK 13
3.4.1 Zipaligning 13
3.4.2 Code signing using jarsigner 14
3.4.3 Code signing using apksigner 15
3.5 Routine flow 15

4 iOS code signing 16

4.1 The iPhone application package 17
4.2 The certificate 18
4.3 Entitlements 19
4.4 Provisioning profiles 21
4.5 Code signing an iPhone Archive 27
4.5.1 Validation 27
4.5.2 Setup 28
4.5.3 Code signing 29
4.5.4 Packaging 30
4.6 Routine flow 30

5 The solution 31

5.1 Issues with the current tools 32
5.2 Use cases 32

5.3 Requirements 33
5.3.1 Environment requirements 33
5.3.2 Implementation requirements 33
5.3.3 Testing requirements 34
5.3.4 Documentation requirements 34

6 Results 35

7 Summary 36

References 38

Appendices

Appendix 1. Certificate parsing using Python

Appendix 2. Generic use cases

Appendix 3. Android use cases

Appendix 4. iOS use cases

Appendix 5. Parsing provisioning profile with a Python regular expression

List of Abbreviations

API	Application programming interface. A set of subroutine definitions, protocols, and tools for building application software.
APK	Android Package Kit. The package file format used by the Android operating system for distribution and installation of mobile apps and middleware.
CA	Certificate authority. An entity that issues digital certificates.
CLI	Command-line interface. A means of interacting with a computer program where the user issues commands to the program in the form of successive lines of text.
DER	Distinguished Encoding Rules. A widely used digital certificate format.
DRM	Digital rights management. A set of access control technologies for restricting the use of proprietary hardware and copyrighted works.
GPGS	Google Play Games Services. A proprietary service and API package for Android devices.
HTTPS	Hypertext Transfer Protocol Secure. An adaptation of the Hypertext Transfer Protocol for secure communication over a computer network.
IDAT	Image Data. A part of the Portable Network Graphics format.
iOS	iPhone OS. A mobile operating system created and developed by Apple Inc. exclusively for Apple's hardware.
IPA	iPhone Archive. The package file format used by the iOS operating system for distribution and installation of mobile apps.
JAR	Java Archive. A package file format typically used to aggregate many Java class files and associated metadata and resources into one file for distribution.

JDK	Java Development Kit. A development kit for the Java Platform.
OSX	Mac OS X. A series of graphical operating systems developed and marketed by Apple Inc. since 2001.
PC	Personal Computer. Computer running either Windows or Linux operating system.
PEM	Privacy-Enhanced Electronic Mail. A de facto file format for storing and sending cryptographic keys, certificates, and other data, based on a set of 1993 IETF standards defining “privacy-enhanced mail”.
PGP	Pretty Good Privacy. An encryption program that provides cryptographic privacy and authentication for data communication.
PKI	Public key infrastructure. A set of roles, policies, procedures needed to create, manage, distribute, use, store, and revoke digital certificates and manage public-key encryption.
PNG	Portable Network Graphics. A raster graphics file format that supports lossless data compression.
RSA	Rivest-Shamir-Adleman. A public-key cryptosystem widely used for secure data transmission.
SDK	Software development kit. A set of development tools that allows the creation of applications for a certain software package, software package, software framework, hardware platform, computer system, operating system, or similar development platform.
SHA1	Secure Hash Algorithm 1. A cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value known as a message digest, typically rendered as a 40 digits long hexadecimal number.
SHA256	Secure Hash Algorithm 256. A cryptographic hash function computed with 32-bit words.

UUID	Universally Unique Identifier. A 128-bit alphanumeric string.

1 Introduction

Rovio Entertainment Corporation is a Finnish games-first entertainment company that creates, develops and publishes mobile games and acts as a brand licensor in various entertainment and consumer product categories. Rovio Entertainment Corporation is best known for the global Angry Birds brand, which started from a popular mobile game in the year 2009. Nine years later the Company offers multiple mobile games for Google's Android and Apple's iOS devices.

Code signing is a small but critical part of Android and iOS application development. Both Apple and Google offer tools for performing code signing, but naturally, each of them has a very different approach to the procedure. Code signing does not require any source code to be compiled and can thus be performed as part or outside a build process. However, building a mobile application can be very time consuming and can therefore create bottlenecks in the mobile application development process. Code signing is a complex but relatively fast procedure, compared to a full build of an application. Thus, performing code signing independently prevents or alleviates bottlenecks by reducing the amount of builds created, errors and time used.

For a company such as Rovio Entertainment Corporation, which develops multiple applications simultaneously, it is of vital importance that parts of the development process, which can be either generalized or standardized, are offered as reliable services that can be used by all development teams. Code signing is one such service. The used code signing service was found lacking especially in scalability, maintainability and reliability.

The primary purpose of this thesis was to provide a general handbook for code signing Android and iOS mobile applications. The secondary purpose of this thesis was to develop a proprietary Python library for code signing mobile applications by using acceptance test driven development principles. This proprietary Python library was to replace the existing code signing service used by Rovio Entertainment Corporation.

2 Code signing

Code signing is the process of digitally signing applications and scripts, in order to confirm the software author, as well as to guarantee that the code has not been altered or corrupted since it was signed [Microsoft 2017]. Code signing is effectively a public key encryption, where all files of an application package are hashed one by one using a hash function (SHA1, SHA256). The resulting hashes are then encrypted individually using the private key of the private/public key-pair, also known as the digital signature. A code signed application package consists of a manifest file of digital signatures mapped to each file, the original unmodified files as well as the public key, for verifying the package contents. Anyone in possession of the public key of a code signed package can verify the package's origin. (Apple 2016a.)

Sealing an envelope with a personal wax seal can be seen as an analogy to code signing. The message or envelope can be opened by anyone, while the unique seal authenticates the sender.

A fundamental problem with public key cryptography is verifying the authenticity of a public key [Xu & Miller-Osborn 2014]. Does the public key belong to the claimed person or entity? Or has a malicious third party perhaps tampered or replaced the public key? While no fully satisfactory solution exists, the usual approach to this problem is to use a public key infrastructure (PKI). The PKI consists of one or more third parties, known as certificate authorities (CA) that certify the ownership of key pairs.

The use of CAs is very common among web applications due to the requirements set by the HTTPS protocol. CAs make it easy to look up the information about a certificate holder as well as revoking outdated or compromised certificates. When it comes to native applications no such system or entity exists that would force registration and management of certificates.

In the world of mobile applications this has been solved using different distribution platforms or stores for applications. Apple's AppStore and Google's Google Play, are both application distribution platforms that offer very similar capabilities, while still placing very different demands on a distributed application and its code signing.

Code signing is limited to only verifying the author of the application or scripts. It does not offer any guarantee whatsoever that the software is free of security vulnerabilities. Nor does code signing guarantee that the software will not load unsafe or altered code from third parties during execution, such as untrusted plug-ins. Code signing does not provide digital rights management (DRM) or copy protection technologies, and, as such, code signing does not in any way hide or obscure the content of the signed software.

2.1 Certification Authorities

Mobile application developers have a few options to consider when code signing applications for distribution. One of these options is the certification authority. The CA keeps track and verifies keys created and registered by the developer. A problem, however, arises when the developer and publisher are not the same. Should the developer be the one registering the key pairs under the CA or should the publisher be doing this? In most cases the legal owner of the software should be the registering party.

Out of the two major mobile platforms, iOS and Android, iOS can be argued to be the “safer” one in terms of code signing. This is mainly due to Apple forcing a centralized PKI strategy where Apple itself is the only valid CA and all developer information needs to be registered with Apple. While developers do not need to generate the keys manually themselves, they still need to issue a certificate signing request to Apple and store the key safely in the build pipeline, in order to sign their application bundles or packages for distribution. Once a package has passed Apple’s AppStore submission process, Apple modifies and re-signs the package using a separate key.

The same is true for Android. Google, however, provides two different strategies. The first one, shown in Figure 2, is a decentralized PKI strategy somewhat similar to Pretty Good Privacy (PGP), where the developer generates the key pair themselves and takes the sole responsibility for the validity of the keys. The second approach is similar to Apple’s, shown in Figure 1, where Google maintains two sets of keys for the developer, a distribution key and an upload key, on a per project basis. The upload key is used to verify the identity of the uploading party, after which Google uses the distribution key to re-sign the Android package.

While Google's second approach and Apple's approach potentially offer better security for distribution keys, it raises suspicion over a package's integrity, as nothing hinders Apple or Google from injecting code into a package before re-signing it.

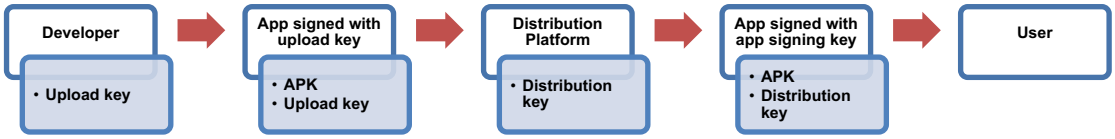

Figure 1. Two step code signing process, where developer first signs a package using an upload key after which the distribution platform re-signs the package using a distribution key.

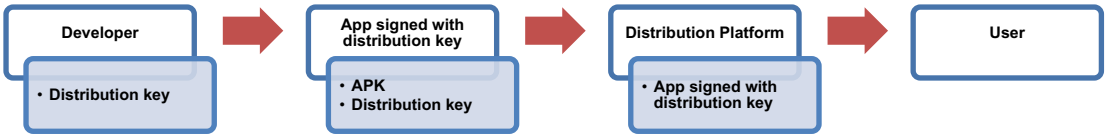

Figure 2. Single step code signing process, where the developer takes the sole responsibility of code signing the package intended for distribution.

2.2 Issues with mobile application code signing.

The most apparent problem with code signing, when it comes to mobile applications, is the process itself. Neither Apple nor Google have particularly clear documentation regarding the technicalities of the code signing procedure.

Apple relies heavily on developers to carry out their code signing of iOS packages through their development framework Xcode [Apple 2016b]. While it is possible to carry out the code signing using tools that come bundled together with Xcode, becoming familiar with these tools takes an excessive amount of time, due to the confusing documentation. Furthermore, the code signing process is complicated because the process requires certificates to be stored in OSX's keychain, which in itself becomes problematic when maintaining several different projects and their certificates. iOS code signing is explained in greater detail in chapter 4.

On Android the process is quite a lot easier to understand, and the command line interfaces (CLI) for tools provided by the Android SDK are quite straightforward to use. The Android SDK's tools work on all platforms (Linux, OSX, Windows), but suffer to some extent from the same issue as OSX when it comes to maintaining several projects and their certificates/keys. Android code signing is explained in greater detail in chapter 3.

A secure approach to code signing is to have a single machine dedicated for distribution code signing, while so called build machines are installed with only development certificates. This limits the risk of exposing the code signing private key, as stricter access control can be enforced on the distribution code signing machine.

2.3 Available third-party tools and their limitations

There are very few third-party tools that offer code signing and re-signing capabilities for mobile applications, and understandably so. Is it really safe to rely on a third party tool for such a sensitive task?

The only noteworthy tool is Fastlane. Fastlane is an open source platform aimed at simplifying Android and iOS deployment, originally developed by Felix Krause [Fastlane 2018a]. The Fastlane platform, can among other things, generate localised screenshots of an application for each language and device, distribute beta builds, publish applications on Apple's AppStore or Google's Google Play, and code sign iOS applications.

Fastlane, however, requires the developer to provide quite a lot of sensitive information for it to work, and thus might repel developers that are more security aware [Fastlane 2018b]. Fastlane does, offer code signing strategies that are less security intrusive, essentially doing the same as Xcode or the Android SDK, which begs the question: is the integration investment worth it or would it be better to rely on the tools provided by Apple and Google? It should be noted that Fastlane does not currently contain any functionality for Android code re-signing and heavily relies on Gradle to carry out the code signing during the building of an application package.

3 Android code signing

3.1 The Android application package

The Android application package called Android Package Kit (APK) is the package file format used by the Android operating system. An APK file contains the compiled executable for the application, resources, assets, certificates, and a manifest file. APK file format is an extension of the Java Archive (JAR) file format, and as such is built on the ZIP format [Oracle 2017]. There is no naming convention in place for APK files, provided that the file always ends with the extension ".apk".

APK files can be installed on Android operating systems, similar to installing software on Windows or OSX, by downloading and installing it from official and unofficial sources. Installing APKs from unofficial sources is by default disabled for security reasons on most Android devices. Users can enable installation of unofficial APKs by setting the option "Unknown Sources" in the devices application manager. APK files that have not been code signed cannot be installed.

APK files, similar to JAR files; contain a META-INF directory. The META-INF directory in turn contains the MANIFEST.MF, CERT.SF, and CERT.RSA. MANIFEST.MF holds the file paths relative to the root of the APK for each file in the APK package as well as Base64 encoded hash digest of its contents. The MANIFEST.MF should not be confused with the AndroidManifest.xml, which can be found at the root of an APK. The AndroidManifest.xml describes essential information about an app to the Android build tools, the Android operating system, and Google Play [Google 2018c]. A sample MANIFEST.MF file is shown in Listing 1.

```
Manifest-Version: 1.0
Created-By: 1.8.0_144 (Oracle Corporation)
Name: res/drawable-mdpi-v4/abc_textfield_search_default_mtrl_alpha.9.png
SHA1-Digest: D6dilO+UMcglambujyMOhNbLZuY=
Name: res/drawable-hdpi-v4/abc_list_longpressed_holo.9.png
SHA1-Digest: KQunCQh0E4bP0utgN0cHdQr9OwA=
Name: res/drawable-ldrtl-xxhdpi-v17/abc_spinner_mtrl_am_alpha.9.png
SHA1-Digest: LRWlMCeV4/Lq+wNklDndVFmtRHU=
Name: res/drawable-xxxhdpi-v4/abc_btn_switch_to_on_mtrl_00012.9.png
SHA1-Digest: AppXOewxC/QI7e8k90KnlawH5MA=
```

Listing 1. Partial contents of a MANIFEST:MF file.

CERT.SF is a ‘signature file’, but does not technically contain any signatures in the cryptographical sense. A sample of a partial CERT.SF file is shown in Listing 2.

```
Signature-Version: 1.0
SHA1-Digest-Manifest-Main-Attributes: /6NgQZa0nSl/UOmCh2TNsE48qxY=
SHA1-Digest-Manifest: z6A4gMlyKV8wYXIT+jcvqNNT2zY=
Created-By: 1.8.0_144 (Oracle Corporation)
Name: res/drawable-mdpi-v4/abc_textfield_search_default_mtrl_alpha.9.p
 ng
SHA1-Digest: IKTGIARYbmam8Kefq6089z/0k98=
Name: res/drawable-hdpi-v4/abc_list_longpressed_holo.9.png
SHA1-Digest: xcQ0bHWRc+R9tuxQ3wgY1a2eY0k=
Name: res/drawable-ldrtl-xxhdpi-v17/abc_spinner_mtrl_am_alpha.9.png
SHA1-Digest: IsziyF2OBzAviNmnk3+DRoNCmAA=
```

Listing 2. Partial contents of a CERT.SF file.

The SHA1-Digest-Manifest in the first section specifies the SHA-1 digest of the manifest file. In the following sections the SHA1-Digest attribute specifies the SHA-1 digest of the corresponding section in the manifest file.

The CERT.RSA file is the “signed signature file”. The CERT.RSA file is a binary file, containing the certificate, the public-key of the code signing key pair, as well as a section containing the hash digest of the CERT.SF file encrypted with the private key of the code signing key-pair.

Note that due to the unique nature of the above files, each of them should be removed prior to re-signing an APK file. This ensures that only the new signature applies. Removal of the files can be achieved in a multitude of ways; the example in Listing 3 utilizes the zip program found on most Unix systems.

```
zip -d infile.apk META-INF/*.SF META-INF/*.RSA META-INF/*.MF
```

Listing 3. Removing the all .SF, .RSA and .MF files from an APK’s META-INF directory.

Because there is a possibility the META-INF directory contains other files than the above it is imperative that only files with the extensions .SF, .RSA, and .MF are deleted. Removing the whole META-INF directory may have undesirable application specific side-effects.

3.2 The keystore

Android code signing relies on a generated certificate and digital “key” which provides a unique, encrypted, and reasonably “unhackable” signature [Miracle 2014]. On Android, this is achieved by using a keystore.

A keystore is a simple file with a large block of encrypted data. The file can be stored anywhere on a machine’s file system, which requires developers to develop conventions storing the file. Because there is no standard location in which to store keystore files, it is easy to misplace them.

There are two types of keystores that need to be considered: debug and release. Functionally and technically these two types do not differ from one another. The debug keystore should be used while developing an Android application, as it allows manual installation of applications on local Android devices. The debug keystore should not, however, be used for an application destined for Google Play, as this requires the release keystore.

A keystore is identified by two aspects: the filename and an alias. A keystore file can potentially store multiple keystores, thus each keystore is identified by an alias. In most cases a keystore only contains one certificate/key pair but will still require an alias.

Keystore files are protected by a pair of passwords: one for the keystore file itself and another for each keystore/alias pair within the file. While the passwords should ideally be unique, it is common to use the same password for both.

3.2.1 Creating keystores

Because an APK has to be signed in order for it to be installable on a device, a keystore is always required when developing an Android application. The Android SDK will automatically create a debug keystore, or use a previously created debug keystore, if a keystore is not explicitly defined. If a developer opts to handle the code signing without using Google Plays code signing service, he or she will have to generate a release keystore him- or herself.

While it would seem logical to create a unique keystore for each application, Google actually recommends using the same keystore for all applications published by the same entity. Google lists three main reasons to do so [Google 2018f]:

> "App upgrade: When the system is installing an update to an app, it compares the certificate(s) in the new version with those in the existing version. The system allows the update if the certificates match. If you sign the new version with a different certificate, you must assign a different package name to the app—in this case, the user installs the new version as a completely new app.
>
> App modularity: Android allows APKs signed by the same certificate to run in the same process, if the apps so request, so that the system treats them as a single app. In this way you can deploy your app in modules, and users can update each of the modules independently.
>
> Code/data sharing through permissions: Android provides signature-based permissions enforcement, so that an app can expose functionality to another app that is signed with a specified certificate. By signing multiple APKs with the same certificate and using signature-based permissions checks, your apps can share code and data in a secure manner."

If an application is to receive future upgrades, the developer needs to ensure that the signing key has a validity period that exceeds the expected lifespan of the application. Google recommends a validity period of 25 years or more. When the validity period of a key expires, users will no longer be able to seamlessly upgrade to new versions of an application signed with the key.

An application published on Google Play, requires that the key's validity period ends after the 22nd of October 2033. This is enforced by Google Play and an application submitted to Google Play, code signed with a validity period ending before the aforementioned date, will be rejected.

Creation of a keystore is achieved through a Java Development Kit (JDK) utility program called keytool. An example for generating a keystore is shown in Listing 4.

```
keytool -genkey -v -keystore yourkeystore.keystore -alias yourkeyalias \
 -keyalg RSA -validity 999999
```

Listing 4. Generating a keystore using the JDK utility program keytool.

The options in Listing 4 being:

- -genkey, generate a key
- -v, verbose output
- -keystore, filename to store the keystore under
- -alias, the alias for the keystore
- -keyalg RSA, use the RSA method for keystore generation
- -validity 999999, set keystore validity period to 999999 days

After executing the command, the user is prompted for more information in a routine appearing as shown in Listing 5 in the command window.

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  YourFirstName YourLastName
What is the name of your organizational unit?
  [Unknown]:  Indie
What is the name of your organization?
  [Unknown]:  Your Company Name
What is the name of your City or Locality?
  [Unknown]:  YourCity
What is the name of your State or Province?
  [Unknown]:  ST
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=YourFirstName YourLastName, OU=Indie, O=Your Company Name, L=YourCity,
ST=ST, C=US correct?
  [no]:  yes
Generating 1,024 bit RSA key pair and self-signed certificate (SHA1withRSA)
with a validity of 999,999 days
     for: CN=YourFirstName YourLastName, OU=Indie, O=Your Company Name,
L=YourCity, ST=ST, C=US
Enter key password for
     (RETURN if same as keystore password):
Re-enter new password:
[Storing yourkeystore.keystore]
```

Listing 5. Prompt sequence initiated by keytool.

The first password is the password for the keystore file itself. Next, the user is asked for his/her first and last name. The prompt for “organizational unit” is for companies with multiple departments like “Engineering” or “Development”. In addition, the user is prompted for his/her city, state/province, and country code, after which the user is prompted to type “yes” or “no” confirming the provided information is correct. Finally, if desired, the user can supply a different password to the individual alias entry, or simply press return or enter to use the same password associated with the keystore file, after which the keystore can be found in the current working directory.

Great care should be exercised when managing keystores. A misplaced or compromised keystore, will prohibit a developer for releasing new versions of an application as updates, and forces the developer to distribute a “new” application on Google Play. Google Play App Signing tries to solve this problem by providing a service that manages the app signing key for the developer [Google 2018e].

3.3 Key hashes and SHA1 signatures

When working with third parties like Facebook or Google Play Games Services (GPGS), the developer is sometimes asked to provide a hash from a keystore. For GPGS, the release keystore must be used for this task. For Facebook, the developer can develop or test with the debug keystore but will eventually have to provide Facebook with information related to the release keystore. (Miracle 2014.)

Both a keyhash (used by Facebook) and a SHA1 signature (used by GPGS) are hash digests calculated from the much larger keystore file [Miracle 2014]. Even though these two are different values, the concept is the same: some standard math is performed on the values in the keystore generating a unique value that cannot be easily reversed, thus ensuring integrity.

Generating a keyhash requires two tools, keytool and openssl, as showcased in Listing 6.

```
keytool -exportcert -alias yourkeyalias -keystore yourkeystore.keystore | \
 openssl sha1 -binary | openssl base64
```

Listing 6. Generating a keyhash using the JDK utility program keytool and openssl.

The three piped commands in Listing 6 have been separated from one another in Listing 7, Listing 8 and Listing 9.

```
keytool -exportcert -alias yourkeyalias -keystore yourkeystore.keystore
```

Listing 7. JDK utility program keytool command exporting a certificate called “yourkeyalias” from a keystore file “yourkeystore.keystore” while prompting the user for the keystore password.

```
openssl sha1 -binary
```

Listing 8. Openssl command that calculates a signature using the SHA1 digest function and outputting in binary format from standard input.

```
openssl base64
```

Listing 9. Openssl command converting the standard input into Base64 encoded format.

If an incorrect password is issued when issuing the piped command in Listing 6, it will lead to the second commands input being “Invalid Password”. This results in the final SHA1 digest being not of the keystore, but the string “Invalid Password”. The commands can naturally be issued one at a time by providing the output of the first and second command as the input of the second and third command respectively. The commands will finally output something similar to string shown in Listing 10.

```
tZRNBKXmYKOa22HvFl57za4gvU0=
```

Listing 10. An example keyhash.

One or more ‘=’ characters at the end of a keyhash are important, as it signifies the end of a Base64 encoded string. The length of a Base64 encoded string should always be divisible by four, and as such multiple ‘=’ characters serve as padding to the string in order to fulfil the length requirement.

GPGS, in contrast to Facebook, requires the text representation of the SHA1 output. Fortunately, the keytool program can output this without relying on additional tools, as shown in Listing 11.

```
keytool -exportcert -alias yourkeyalias -keystore yourkeystore.keystore -list
-v
```

Listing 11. JDK utility program keytool command for listing certificate information.

After providing the password(s), output similar that shown in Listing 12 is displayed in the command window.

```
Alias name: youralias
Creation date: Aug 24, 2014
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=YourFirstName YourLastName, OU=Indie, O=Your Company Name,
L=YourCity, ST=ST, C=US
Issuer: CN=YourFirstName YourLastName, OU=Indie, O=Your Company Name,
L=YourCity, ST=ST, C=US
Serial number: 53fa57f7
```

```
Valid from: Sun Aug 24 17:24:07 EDT 2014 until: Sun Jul 20 17:24:07 EDT 4752
Certificate fingerprints:
      MD5:  66:22:E9:94:EA:14:EA:4A:06:EB:98:8B:DA:2B:25:D2
      SHA1: B5:94:4D:04:A5:E6:60:A3:9A:DB:61:EF:16:5E:7B:CD:AE:20:BD:4D
      Signature algorithm name: SHA1withRSA
      Version: 3
```

Listing 12. Certificate information generate by the JDK utility tool keytool.

The string of hex digits after SHA1, in Listing 12, is what a developer needs to provide GPGS and Google when setting up their application.

3.4 Code signing an APK

The documentation for Android Studio by Google, provides basic examples for code signing an APK manually [Google 2018d]. The process is carried out in two main steps: zipaligning the APK and code signing the APK.

The second step, code signing, can be done using two different tools: the JDK utility tool jarsigner and the Android SDK utility tool apksigner. While both tools are capable of performing the code signing task, it should be noted that Google introduced the APK Signature Scheme v2 with Android 7.0 [Google 2018a], which makes apksigner the recommended option considering the future of Android.

3.4.1 Zipaligning

The first step inherits its name from the Android SDK tool zipalign [Google 2018g]. Zipalign is an archive management tool for optimizing APKs, by ensuring uncompressed data within the APK, such as images and raw files, start with a particular alignment relative to the start of the APK file with 4-byte boundaries. This allows all portions to be accessed directly with mmap() even if they contain binary data with alignment restrictions, which results in a reduction in the amount of RAM consumed when running an application. Zipalign has to be used each time a package is to be distributed, be it for development purposes or releasing it to end-users.

Listing 13 depicts example usage, where an infile.apk is aligned and saved as outfile.apk

```
zipalign -p 4 infile.apk outfile.apk
```

Listing 13. Zipalign command for zipaligning a APK package.

The “-p” option stands for memory page alignment for stored shared object files. The “4” is the alignment in bytes. Listing 14 shows the command for checking whether an APK is zipaligned or not.

```
zipalign -c outfile.apk
```

Listing 14. Zipalign command for checking the zipalignment of APK.

3.4.2 Code signing using jarsigner

Code signing of the infile.apk using jarsigner can be done by issuing the command shown in Listing 15 [Oracle 2018].

```
jarsigner -keystore yourkeystore.keystore -storepass:env YOUR_KEYSTORE_PW_ENV
-keypass:env YOUR_ALIAS_PW_ENV infile.apk yourkeystorealias
```

Listing 15. JDK utility program jarsigner command for signing an APK package.

The options used in the command shown in Listing 15 are as follows:

- -keystore. location of the keystore to be used
- -storepass:env, use the keystore password from an environment variable
- -keypass:env, use the alias password from an environment variable, required when the alias password differs from the keystore password

Verification of an APK’s digital signature can be done using jarsigner as shown in Listing 16.

```
jarsigner -verify outfile.apk
```

Listing 16. JDK utility program jarsigner command for verifying an APK digital signature.

Zipaligning of an APK code signed using jarsigner should be performed after code signing the APK [Google 2018g].

3.4.3 Code signing using apksigner

In contrast to performing the zipaligning after the code signing, when using jarsigner, the zipaligning should be performed before code signed an APK using apksigner [Google 2018g]. Code signing and verifying an APK using apksigner is shown in Listing 17 and Listing 18 respectively [Google 2018b].

```
apksigner sign --ks yourkeystore.keytore --ks-pass env:YOUR_KEYSTORE_PW_ENV --
ks-key-alias yourkeystorealias --key-pass env:YOUR_ALIAS_PW_ENV outfile.apk
```

Listing 17. Android SDK utility program apksigner command for code signing an APK.

```
apksigner verify outfile.apk
```

Listing 18. Android SDK utility program apksigner command for verifying an APK is signed.

3.5 Routine flow

A simplified flow diagram of code signing an Android APK package can be derived from the explanations in chapter 3, as shown below in Figure 3. The input of every routine is always checked for failure, where a failure causes the routine to clean up the working directory and exit.

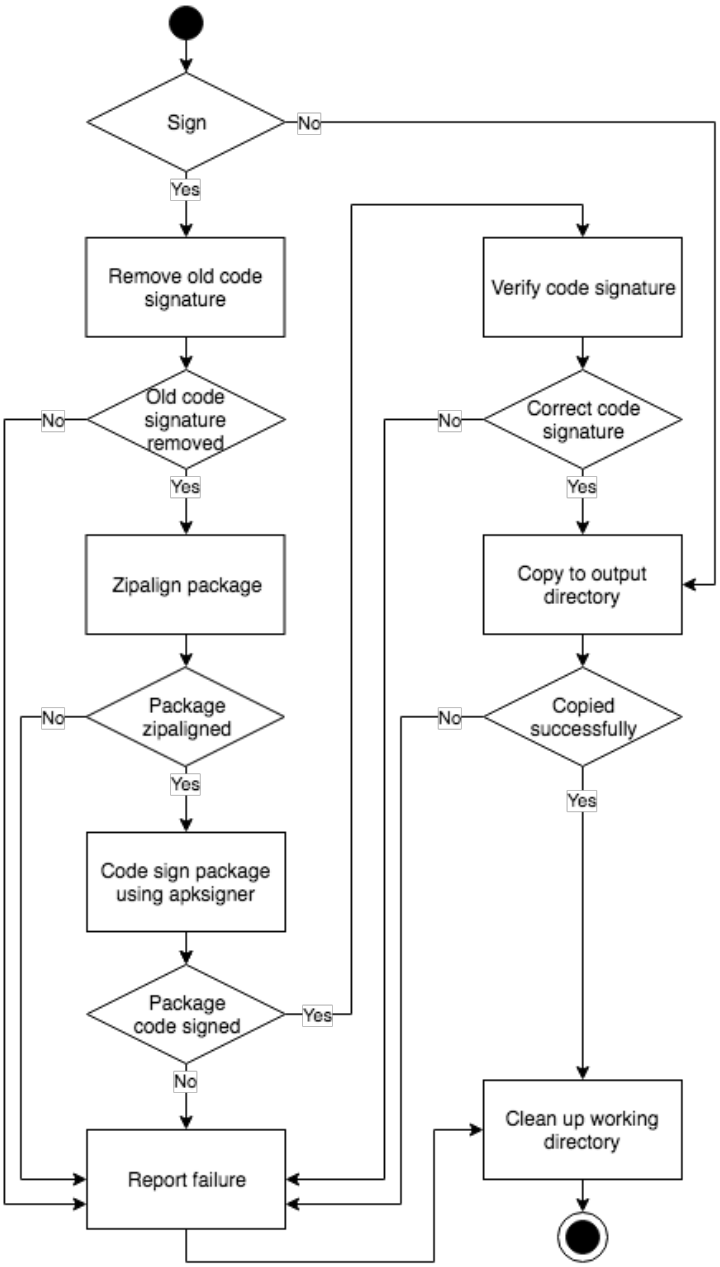

Figure 3. Routine flow of code signing an Android APK package.

4 iOS code signing

While iOS codes signing is essentially similar to Android code signing, the practical side of iOS code signing is vastly different from Android code signing. The tools used in iOS code signing are different and the code signing has to be executed on a machine running OSX, because no official tools exist for other platforms. There are also more distribution types than the two used in Android code signing (development, production),

which each require their own project specific provisioning profiles, resulting in a potentially unmaintainable amount of provisioning profiles. Apple has improved the functionality for maintaining provisioning profiles in Xcode, but it is still quite unclear how to automate the process.

4.1 The iPhone application package

The application package file format for iOS has no official definition but is referred to as iPhone Application or iPhone Archive (IPA) by developers. Similarly, to the Android APK the IPA is also an extension of the ZIP archive. The IPA can be extracted the same way as any other ZIP archive, however, portable network graphics (PNG) images found in the archive, e.g. application icons, are sometimes in a proprietary variant of the PNG format (pngcrush). A so called pngcrushed PNG is usually of reduced size compared to the standard version of the PNG, thanks to various combinations of compression methods and delta filters applied to the image data (IDAT) stream. The additional compression of the IDAT stream makes the pngcrushed PNG files unreadable by most image viewers found on PCs. Additionally, the application binary found in an IPA is encrypted to enforce DRM, when the IPA is downloaded from Apple's AppStore, effectively making examination of the binary impossible. IPAs distributed Ad Hoc outside of Apple's AppStore do not have the binary encrypted.

While some command line tools exist for installing IPAs on an iPhone or iPad on Linux and OSX, the only viable way to install IPAs on Windows is through iTunes. Invalid provisioning of an IPA can still fail an installation regardless of method of installation. This cannot be disabled, contrary to the similar functionality available on Android.

The IPA usually only contains one directory called Payload on the root level of the archive. However, possible application extensions can be found on the root level as well. The Payload directory contains the actual application package file with the extension .app. The application package contains among other files the Info.plist property-list file, a _CodeSignature directory, the embedded.mobileprovision file and the actual application binary file.

The Info.plist is similar to the AndroidManifest.xml as it contains both metadata about the application as well as preferences and information about resources used by the

application [Apple 2016d]. The _CodeSignature directory also bares resemblance to the APK equivalent META-INF directory. The _CodeSignature directory contains the CodeResources file, which is a property-list file containing a mapping of all the packages files as well as Base64 encoded hash digests of their contents, similar to an APK's MANIFEST.MF and CERT.SF.

The embedded.mobileprovision file is commonly referred to as the provisioning profile. The provisioning profile contains among other things, the public key of the certificates signing key pair, team-identifier, a list of universally unique identifiers (UUID) of devices that can install the application and the expiration time of the provisioning profile itself. (Apple 2018b; Abhimuralidharan 2018.)

4.2 The certificate

The digital certificate that is used for code signing an IPA is an X.509 certificate. The certificate is a collection of data used to distribute the public half of a private/public key pair. Along with structural information, the certificate contains name and contact information for its issuer and its owner, or subject, as well as the owner's public key. The certificate also contains a date range that indicates when the certificate is valid. A certificate may also contain extensions, which provide additional information and conditions, e.g. acceptable uses for the public key. (Apple 2018c.)

When the certificate is assembled, the issuer of the certificate signs the certificate using the issuer's own identity, private key and certificate, to vouch for the certificates integrity. iPhone developer certificates are issued only by Apple, and as such are always signed by Apple using Apple's own identity.

Certificates used for code signing IPA packages are usually stored in an OSX keychain together with the private key, forming a code signing identity, and given a name. The name of this code signing identity is used during the actual code signing process, rather than the individual file names of the private key and certificate. This adds an extra layer of security, as not all users of a machine have access to all OSX keychains.

Due to the nature of the code signing identities found in an OSX keychain, the keychain should be backed up remotely in a secure place, in case of hardware or software fail-

ure. Apple does not keep a copy of the private key, as that would compromise the signing identity of an iPhone developer's own certificate. If a code signing identity, which has been used for signing an IPA package intended for distribution, is lost or compromised, the developer certificate needs to be revoked. In order to re-enable an application's services after a certificate revocation, a new certificate needs to be obtained from Apple and a new version of the application, signed with the new certificate, submitted to Apple's AppStore. Certificates may be obtained from Apple's developer portal. (Apple 2018a)

4.3 Entitlements

The Entitlements of an IPA package can be thought of as application permissions, similar to permissions set in an APK's AndroidManifest.xml file. However, an IPA's Entitlements are strings written into the code signature of an application. An Entitlement grants the application a specific capability or access to a specific service. The operating system (OS) inspects an application's Entitlements before allowing an application access to certain features. For instance, an application must have the iCloud entitlement before it is allowed to access iCloud APIs at runtime. The OS enforces this by checking the application for an iCloud entitlement such as the one shown in Listing 19.

```
<key>com.apple.developer.ubiquity-kvstore-identifier</key>
<string>123ABC.com.example.app</string>
```

Listing 19. An example iCloud entitlement.

There are two possible locations where entitlements are defined: entitlements embedded in a provisioning profile and entitlements defined in a code signing entitlements file. According to Apple's documentation [Apple 2015], the values of the code signing entitlement file are used to fill in any wildcard, asterisk portions of the entitlements that might exist in the code signing provisioning profile. However, if entitlements embedded in a provisioning profile and entitlements file have the same key, that key's value has to be equal to the one defined in the provisioning profile. The codesign utility tool, which comes bundled with Xcode, performs some validation of the entitlements before performing the code signing.

Listing 20, Listing 21, Listing 22 and Listing 23 depict typical entitlement related errors.

```
The executable was signed with invalid entitlements. The entitlements speci-
fied in your application's Code Signing Entitlements file do not match those
specified in your provisioning profile.
```

Listing 20. Entitlement mismatch error

```
Could not install the application. Your code signing/provisioning profiles are
not correctly configured ... you have an entitlement not supported by your
current provisioning profile ... (error: 0xe8008016).
```

Listing 21. Installation error

```
Invalid Code Signing Entitlements. The entitlements in your app bundle
signature do not match the ones that are contained in the provisioning
profile. According to the provisioning profile, the bundle contains a key val-
ue
that is not allowed: '[ A1B2C3D4E5.com.example.MyGreatApp ]' for the key
"keychain-access-groups" in "Payload/MyGreatApp.app/MyGreatApp
```

Listing 22. Submission error

```
entitlement 'keychain-access-groups' has value not permitted by a provisioning
profile
```

Listing 23. Invalid values error

Inspections of an iOS application's entitlements can be done by running the command shown in Listing 24.

```
codesign -d --ent :- /path/to/the.app
```

Listing 24. Xcode utility tool codesign command for inspecting the entitlements of iOS or OSX application.

Where the parameters are as follows:

- -d - Display information about the code at the path
- --ent - Extract any entitlement data from the signature and write it to the path given. "-" writes to standard output. By default, the binary "blob" header is returned intact, prefixing the path with a colon ":" will automatically strip it off. If the signature has no entitlement data, nothing is written.

A code signing entitlements file is a simple property-list file containing a single dictionary similar to the one found embedded in the provisioning profile. The contents of an example code signing entitlements file is show in Listing 25.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
     <key>application-identifier</key>
     <string>ABCD1234EF.com.example.app</string>
     <key>com.apple.developer.team-identifier</key>
     <string>ABCD1234EF</string>
     <key>get-task-allow</key>
     <false/>
     <key>keychain-access-groups</key>
     <array>
          <string>ABCD1234EF.*</string>
     </array>
</dict>
</plist>
```

Listing 25. Example contents of a code signing entitlements file.

4.4 Provisioning profiles

A provisioning profile is a collection of digital entities that uniquely ties developers and devices to an authorized iPhone development team and enables a device to be used for testing of the application under development. A development provisioning profile must be installed on each device on which application code is to be run. Each development provisioning profile will contain a set of iPhone development certificates, unique device identifiers and an app ID. Devices specified within the provisioning profile can be used for testing application under development locally only by those individuals whose iPhone development certificates are included in the profile. A single device can contain multiple provisioning profiles. Provisioning profiles can be obtained from Apple via the iOS provisioning portal.

There are five distinct variants of provisioning profiles [Umbaugh & Dunn 2018]:

- Development: Typically used when running locally on a phone plugged in to a computer. Allows debugger access and uses development Game Center and development push notifications.
- Team: A kind of development profile that is managed by Xcode. Xcode automatically adds everyone's certificate to it and all UDIDs from devices as well.
- Ad-Hoc: Often used when sending application to third party testers. It does not allow debugger access and uses development Game Center, but production push notifications.
- Enterprise: A provisioning profile that runs on any device. It uses production Game Center and production push notifications.

- Distribution: A provisioning profile that is intended for packages intended for distribution via Apple's AppStore. The provisioning profile has no ProvisionedDevices or ProvisionsAllDevices key value pair. As such, a package containing a distribution provisioning profile cannot be installed on a device.

A provisioning profile file usually has the extension .mobileprovision or .provisionprofile and is a Cryptographic Message Syntax (CMS) encrypted property-list file. The encryption ensures that the provisioning profile was created by Apple and not some malicious third party. Due to the encryption, the provisioning profile needs to be decoded before inspection. The command for decoding a provisioning profile is shown in Listing 26 [Solodovnichenko 2017]:

```
/usr/bin/security cms -D -i embedded.mobileprovision
```

Listing 26. An OSX program security command that decodes a provisioning profile.

The above command is, however, time consuming when decoding multiple files in sequence. A faster approach is to simply read the file and capture all the characters in between and including the strings <plist and </plist>, as the property-list part of the file is not always encrypted. A python regular expression of the form `(?=\<plist)\<plist(?:.*\s)*\<\/plist\>` can be used to achieve this. Appendix 5 showcases the parsing of a provisioning profile using a Python regular expression. The sample contents of the property-list segment of a provisioning profile is shown in Listing 27.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
     <key>AppIDName</key>
     <string>App ID Name</string>
     <key>ApplicationIdentifierPrefix</key>
     <array>
          <string>ABC123DEF4</string>
     </array>
     <key>CreationDate</key>
     <date>2042-10-15T15:20:42Z</date>
     <key>Platform</key>
     <array>
          <string>iOS</string>
     </array>
     <key>DeveloperCertificates</key>
          <array>
          <data>Long human-unreadable Base64 encoded certificate=</data>
          </array>
     <key>Entitlements</key>
     <dict>
          <key>keychain-access-groups</key>
          <array>
               <string>ABC123DEF4.*</string>
          </array>
          <key>get-task-allow</key>
          <false/>
          <key>application-identifier</key>
          <string>ABC123DEF4.com.your.bundle.id</string>
          <key>com.apple.developer.team-identifier</key>
          <string>U1R23TEAMID</string>
          <key>aps-environment</key>
          <string>production</string>
          <key>beta-reports-active</key>
          <true/>
     </dict>
     <key>ExpirationDate</key>
     <date>2044-10-15T15:20:42Z</date>
     <key>Name</key>
     <string>Profile Name</string>
     <key>TeamIdentifier</key>
     <array>
          <string>U1R23TEAMID</string>
     </array>
     <key>TeamName</key>
     <string>Your team name</string>
     <key>TimeToLive</key>
     <integer>364</integer>
     <key>UUID</key>
     <string>thats-your-profile-uuid-string</string>
     <key>Version</key>
     <integer>1</integer>
</dict>
</plist>
```

Listing 27. An example provisioning profiles intended for distribution via Apple's AppStore.

The decoded property-list contains its own Entitlements dictionary, which contains among other things values that have to be reflected in the Entitlements.plist file. The entitlements of the provisioning profile always take precedence over other entitlements

provided during the code signing process. Thus, any entitlement keys that are present in an Entitlements.plist file that are also present in the provisioning profile have to be equal, as explain in section 4.3. The following list of key value pairs were found to commonly appear in both the provisioning profile and Entitlements.plist file:

- com.apple.developer.team-identifier
- keychain-access-groups
- get-task-allow
- aps-environment
- beta-reports-active
- com.apple.developer.ubiquity-kvstore-identifier

Additionally, the certificate for signing the package needs to exist in the DeveloperCertificates array as Base64 encoded data. The developer certificates are X.509 distinguished encoding rules (DER) certificates and can be decoded by using the openssl command line tool. Decoding of a Base64 encoded X.509 DER certificate is shown in Listing 28 [Solodovnichenko 2017].

```
cat your_cert.base64 | \
 base64 -D | \
 openssl x509 -noout -inform DER -issuer -subject -dates -fingerprint -pubkey
```

Listing 28. Cat, base64 and openssl commands piped together to produce human readable output of a Base64 encoded certificate string.

The cat tool simply reads the your_cert.base64 file and pipes the output to the Base64 tool which decodes the Base64 encoded string through the -D parameter. The decoded output is then piped further to the openssl tool which produces human readable information depending on the issued command line arguments. The command line arguments used for openssl in Listing 28 are as follows [Openssl 2017]:

- x509 - specifies X.509 standard
- -noout - disable certificate output, enabled by default
- -inform DER - specifies X.509 DER format
- -issuer - output issuer information
- -subject - output subject information
- -dates - output certificate notBefore and notAfter dates
- -fingerprint - output SHA1 fingerprint of the certificate

- -pubkey - output the public key of the certificate

The information produced by the command in Listing 28, should be enough to determine the state of a developer certificate, specifically, whether it has expired or has changed in any way. A provisioning profile may contain invalid or expired developer certificates.

All provisioning profiles have a set time to live defined by the TimeToLive key in the property-list, as well as a CreationDate and ExpirationDate keys. Apple usually sets the TimeToLive value to 365, i.e. one year, which means provisioning profiles need to be renewed on an annual basis. The CreationDate and ExpirationDate should not be confused with the developer certificate notBefore and notAfter values, as they are unrelated and are to be validated separately.

Table 1. A typical code signing identity and provisioning profile distribution on a per project basis.

Code signing identity / Project (p_i)	**Distribution (x_1)**	**Development (x_2)**	**Enterprise (x_3)**	**Extra Certificate (c_1)**
Project 1	Distribution	Ad-hoc Development Team	Enterprise	Ad-hoc
Project 2	Distribution	Ad-hoc Development Team	Enterprise	Ad-hoc
Project (N)	1	3	1	m_k, $0 < m_k < 4$

Assuming an entity is developing two (2) iOS projects, said entity would need at least three code signing identities (certificates); development, distribution and enterprise, as well as a unique provisioning profile for each needed distribution type (development, ad-hoc, team, enterprise, distribution) for each certificate on a per active project basis. A typical scenario would thus lead to a situation where two active projects would use all of the three (3) certificates and have at least five (5) unique provisioning profiles each. If said entity would add an additional certificate the amount of provisioning profiles would increase by at least two (2) as depicted in Table 1. Thus, we have the following function for the total minimum amount of provisioning profiles f

$$f(c, m, p, x) = \sum_{j=1}^{p} \left(\sum_{k=0}^{c_j} m_k + \sum_{i=1}^{3} x_i \right),$$

$$\{p \in \mathbb{Z}^+, x_i \in [0:3], m_j \in [1:3] \wedge m_0 = 0, c \in [0:+\infty[\}$$

where p is the number of projects, m_k is the number of additional provisioning profiles for an additional certificate k and c_j is the number of extra certificates for project j. x_i are the counts of provisioning profiles for the baseline code signing identities (distribution, development, enterprise). Assuming a baseline of 5 provisioning profiles per active project, the above equation can be further simplified as

$$f(p) = p(5 + m),$$

$$\{p \in \mathbb{Z}^+, m \in [0:+\infty[\}$$

where p is the number of projects and m is the average count of extra provisioning profiles per active project.

Applying the above equations to the values given in Table 1, we receive the following:

$$f = \big((0 + 1) + (1 + 3 + 1)\big) + \big((0 + 1) + (1 + 3 + 1)\big) = 12$$

or

$$f(2) = 2 \times (5 + 1) = 12$$

The above function is a reasonably good approximation of the minimum amount of provisioning profiles needed for a set of active projects. The above function for a set of 20 active projects and 5 baseline provisioning profiles gives us a total of 100 unique provisioning profiles. Each of these files has to be maintained separately. The above equations do not serve any practical purpose beyond approximating the amount of provisioning profiles needed for a set of active projects but gives a clear view of how many files need to be annually updated.

It is possible, to develop iOS applications with fewer provisioning profiles using wildcard entitlements, resulting in a set of generic provisioning profiles that are valid for all active projects. This can, however, lead to false positive issues due to improper code signing, because some entitlements might not be set correctly, because a wildcard-provisioning profile's entitlements being too generic. iOS Code signing issues are noto-

riously subtle and are an unnecessary waste of development time and should therefore only be handled by entities that are autonomous or highly specialised in the process of code signing.

4.5 Code signing an iPhone Archive

Code signing an iPhone Archive (IPA) package consists of four main stages: validation, setup, code signing and packaging. The process requires an IPA package, a provisioning profile as well as an available code signing identity in an OSX keychain. The following sections will cover each stage in more detail, by explaining their function and how to carry out each of them in practice.

4.5.1 Validation

While the codesign utility tool does some validation, it allows you to codesign IPA packages with expired certificates and provisioning profiles. An IPA package submitted to Apple's AppStore code signed with expired certificates or provisioning profiles will be rejected by Apple, but it can also have other side effects. Push notifications or in app purchases (IAP) might not work while testing the application or the application fails to install on a device. Thus, some additional validation of the certificate and provisioning profile is warranted. When validating the certificate and provisioning profile at least the following should be considered:

- Ensure that the code sign identity can be accessed from in the OSX keychain
- Ensure that the provisioning profile exists and is readable
- Ensure that the certificate is available among the developer certificates defined in the provisioning profile
- Ensure that neither the certificate or provisioning profile have expired

A code signing identity stored in an OSX keychain can be inspected using the security utility tool. The command in Listing 29 will output an X.509 certificate in Privacy-enhanced Electronic Mail (PEM) format, where the parameters are as follows [Apple 2012]:

- find-certificate - Find a certificate item

- -c - Match on name when searching
- -p - Output certificate in PEM format. Default is to dump the attributes and keychain the certificate is in.

```
/usr/bin/security find-certificate -c "name of the code signing identity" -p
```

Listing 29. An OSX program security command for finding code signing identities and outputting them in PEM format.

The obtained certificate can then be inspected using the openssl tool, by piping the command as shown Listing 30, and producing output similar to the example output shown in Listing 31.

```
/usr/bin/security find-certificate -c "name of the code signing identity" -p |
/usr/bin/openssl x509 -noout -inform PEM -issuer -subject -dates -fingerprint
-pubkey
```

Listing 30. Decoding a PEM formatted certificate produced by the OSX program security using the openssl program.

```
issuer= /C=US/O=Apple Inc./OU=Apple Worldwide Developer Relations/CN=Apple
Worldwide Developer Relations Certification Authority
subject= /UID=12345ABCDEF/CN=iPhone Developer: MyUserName (XYZ-
AB12345)/OU=X1Z2Y3ABCD/O=My Organization Ltd/C=US
notBefore=Jan 1 00:00:00 2018 GMT
notAfter=Jan 1 00:00:00 2019 GMT
SHA1 Fingerprint=A1:B2:C3:D4:E5:F6:A1:B2:C3:D4:E5:F6:A1:B2:C3:D4:E0:F0:A0:00
-----BEGIN PUBLIC KEY-----
APublicKeyHash==
-----END PUBLIC KEY-----
```

Listing 31. Example output produced by the commands shown in Listing 30.

This output can easily be parsed further using any programming or scripting language. An example using Python can be viewed in Appendix 1.

4.5.2 Setup

Code signing with the codesign utility tool requires an unarchived IPA package. Additionally, in order for entitlements to be set correctly, the previous entitlements of a signed package should be read, in order to ensure that the entitlements are set correctly, as per section 4.4, when re-signing the package with a new certificate and/or provisioning profile. It is also useful to parse the Info.plist file in case any form of modification of the property-list needs to be done, for instance changing the CFBundleIdentifier key value pair.

If required, the Info.plist and the embedded.mobileprovision of the processed should be replaced with updated versions respectively. Updated entitlements should be stored in a file, e.g. Entitlements.plist.

4.5.3 Code signing

Once the Info.plist and embedded.mobileprovision files have been updated and the Entitlements.plist file has been created, code signing the package is as simple as issuing the command shown in Listing 32, where the parameters are as follows [Apple 2011]:

- -f - Replace any existing signature on the path(s) given. Without this option, existing signatures will not be replaced, and the signing operation fails.
- -s - Sign the code at the path(s) given using this identity.
- --entitlements - Take the file at the given path and embed its contents in the signature as entitlement data. If the data at path does not already begin with a suitable binary (“blob”) header, one is attached automatically.
- --keychain - During signing, only search for the signing identity in the keychain file specified. This can be used to break any matching ties if you have multiple similarly-named identities in several keychains on the user’s search list. Note that the filename will not be searched to resolve the signing identity’s certificate chain unless it is also on the user’s keychain search list.

```
/usr/bin/codesign -f -s “My code signing identity” --entitlements \
 /path/to/Entitlements.plist --keychain “MyKeyChain” \
 /path/to/myapp/Payload/my.app
```

Listing 32. Xcode utility program codesign command for code signing an iOS or OSX application.

It should also be noted that packages containing nested code content such as helpers, frameworks, and plugins, should each be signed in turn. The deepest nested code content should always be signed first, due to the files created or modified during code signing. Additionally, the same code signing rules i.e. entitlements, provisioning profile, bundle identifier, will most likely not apply among the nested code content packages and main package, and as such need to be defined and set separately [Apple 2016c].

4.5.4 Packaging

After executing the code signing successfully the code signed files need to be repackaged. Assuming a code signed package has the path ./MyApp/Payload/my.app/*, everything under the MyApp directory should be packaged, using the zip tool, resulting in an IPA package, e.g. MyApp.ipa.

4.6 Routine flow

A simplified flow diagram of code signing an IPA package can be derived from the explanations in chapter 4, as shown below in Figure 4. The input of every routine is always check for failure, where a failure causes the routine to clean up the working directory and exit. The iOS routine flow is significantly more complex compare to the Android counterpart.

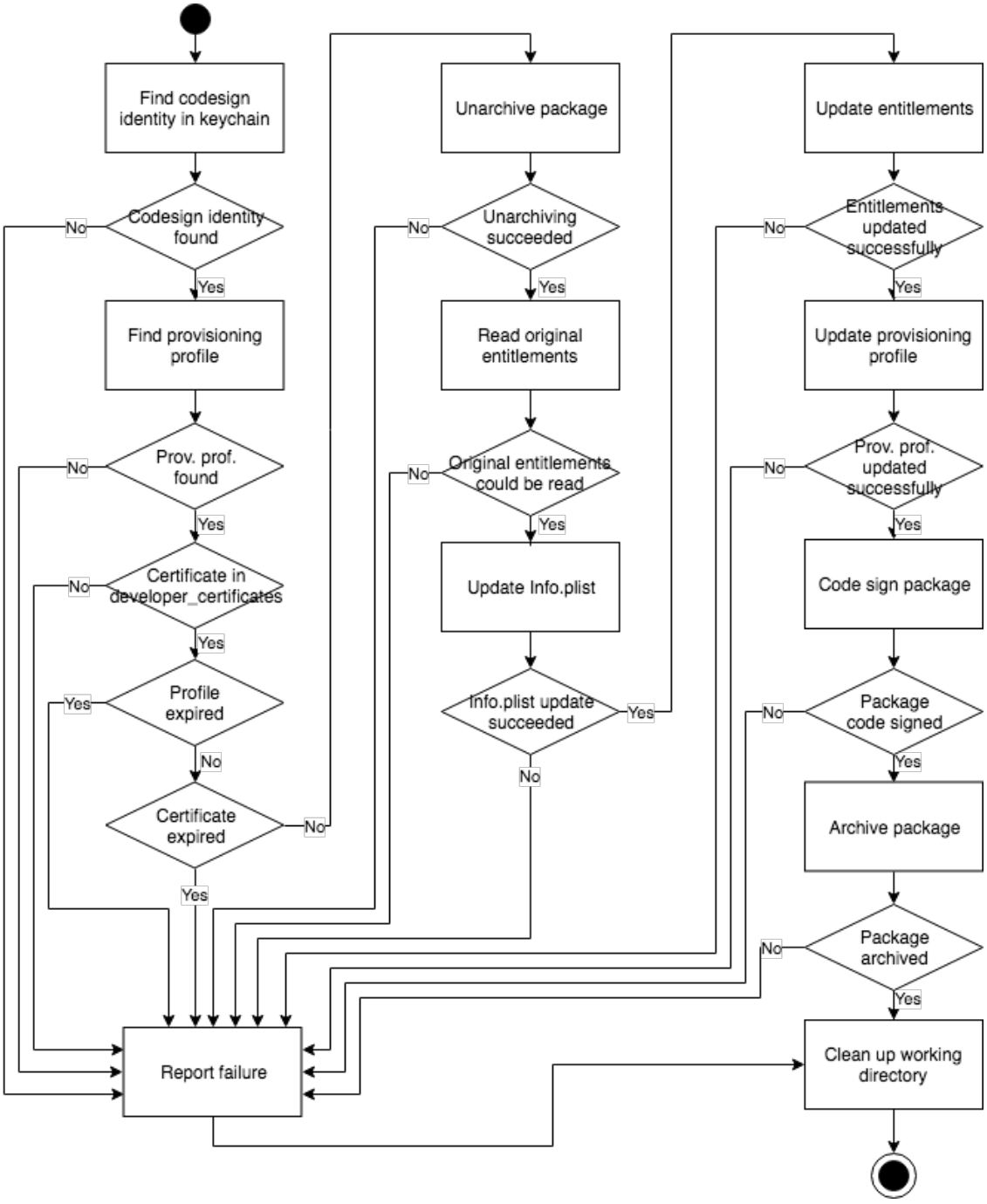

Figure 4. Routine flow of code signing an iOS IPA package.

5 The solution

Development of a new solution was agreed to be done in stages, where the first stage would consist of a library implementation of the actual code signing logic. The library should have unit and functional test coverage of at least 80% and provide a simple CLI

for basic manual testing. The following stages would consist of the actual service implementation development and further enhancement of the first stage's core library by adding new functionality. The following sections cover the design of first stage's core library. The new solution was to be identified as BuildResigner.

5.1 Issues with the current tools

The previous code signing solution was created as a proof of concept in 2014. It, however, gradually ended up in production, due to other services depending on its functionality, first through basic build jobs and later through complicated automated file uploads through file servers. Because of the very fragile implementation of and lack of proper documentation of the previous code signing solution, it soon became close to impossible to maintain and develop.

The previous solution was a single python file consisting of around 1200 lines of code with functions as long as 250 lines, handling both iOS and Android code signing, while also sending email notification and distributing signed packages to file servers. The solutions had integration tests that turned out to be more of a hindrance than help when maintaining the previous code signing solution. The previous code signing solution also heavily relied on another project, making the checkout from version control problematic.

While the implementation of the previous solution had its flaws, the main issues with the solution were its outbound interfaces, scalability, file verification and logging. This led to failures under heavy load and made debugging a nightmare.

5.2 Use cases

The use cases defined for the solution were to follow an acceptance test driven development (ATDD) style, where each use case was to be accompanied by a set of requirements. All requirements were to be fulfilled in order for a use case to be seen as implemented. The use cases were divided into three categories: Generic, Android and iOS. The generic use cases mainly consisted of the development and testing related

use cases, whereas, the Android and iOS use cases contained implementation requirements for each platform.

Users were seen as individuals that use the CLI and developers as individuals that would take care of development and maintenance of the solution. The solution was to be referred to as the application or BuildResigner.

The use cases are listed in Appendices 2-4.

5.3 Requirements

The generic requirements were defined by four (4) separate sub-requirements: environment, implementation, testing and documentation.

5.3.1 Environment requirements

Due to the nature of the code signing of iOS applications the tool would only have to work on the OSX operating system. The new solution was to be implemented using the third major version of the Python programming language (python3), thus python3 would have to be installed on a system before developing or running the new solution. No support for the second major version of Python was considered.

5.3.2 Implementation requirements

The new solution should aim to be as modular as possible, allowing easy testing and extension. Log messages of the info, warning, error and exception levels were to always be logged, and the exception level messages were to also contain the stack trace for easier debugging. An optional CLI flag for logging debug messages was also to be provided.

5.3.3 Testing requirements

Unit and functional testing were to be executed using the py-test testing library [Krekel 2018a]. Data and assets required by the tests were to be available alongside the source code.

The source code was to include a tox.ini configuration file, enabling running of py-test tests through the tox virtual environment management tool [Krekel 2018b]. As tox can be run both locally or through a CI system, all environment related setup was to be done by tox. This in turn would allow testing of the testing environment setup locally, without having a CI system installed, against different python versions in the same test run.

The source code was also to include a Jenkinsfile allowing CI through a Jenkins CI server. The Jenkinsfile's instructions were to simply set up the workspace and run tox within it. This would result in minimal changes to the Jenkinsfile throughout the development of the tool.

A developer was to be able to execute testing using any of the following commands from the solution's source root directory:

- `pytest .`
- `python3 setup.py test`
- `tox`

5.3.4 Documentation requirements

The documentation was to consist of three major areas: technical information, usage and source code comments. The technical information was to be covered by this thesis, while the usage was to be written as Markdown in a text file called README.md for better visualisation while viewing the file in version control. All source code functions were to have descriptive names as well as contain a comment explaining the functionality.

6 Results

Based on the requirements set during the initiation of the project in June 2017, the research and prototyping phase for the library was started in July 2017. The focus was on finding an implementation that would both work on its own as well as provide a clear interface for implementing the library into existing technology stacks. Ultimately a clear vision for the library implementation was found, as well as strategy for how the library could be incorporated into existing technology stacks.

The flow diagrams in section 3.5 and section 4.6 were derived from work done during the prototyping phase. Understanding the routine flow of the previous code signing solutions was the most time-consuming part of the prototyping phase. It was not always clear when reading the source code, how branches were chosen or what the parameters should be, as there were no examples or unit tests in place. The review also identified some dead code as well as some oversights in exception handling. It also became abundantly clear that the lack of validation in the previous code signing implementation had to be rectified in the new implementation. Thus, leading to a revision of the use cases and the addition of validation and verification of all input given.

The implementation phase was started in September 2017, starting with the general use cases. A simple CLI was set up to accommodate the need for manual testing, as well as functionality for validating all user input, including schemas for each type of configuration file. Logging functionality was copied from another proprietary Python project to speed up development. During the implementation phase it was also realized that the library would not necessarily need to follow a typical object-oriented programming approach, thanks to the flexibility of the Python programming language, but could use a dynamic call stack using higher order functions. Implementing all of the general use cases, tests included, took approximately a week for a single developer.

Implementation of the Android use cases was started in late September 2017 after the general use cases had been implemented. It was decided that Android would be implemented before iOS, as the Android routine flow was clearer. Implementing the functionality for codesigning Android APK packages took three weeks, as extra effort was put in to ensure that test coverage was sufficient.

The functionality for iOS IPA package codesigning was started in October 2017 but had to be revised a couple of times due to misunderstandings regarding the IPA entitlements. The work also took twice as long compared to the Android implementation (4 weeks), due to the complexity of the test cases.

The technical part of the documentation, i.e. the technical part of this thesis, was started in February 2018 and worked on throughout the spring of said year. During this time the stable library was also used in a proof of concept, for an upcoming feature in a technology stack utilizing the Django web framework. The proof of concept not only helped further enhance the library itself, but also proved a hypothesis regarding the utilization of Apple Mac Minis for OSX specific functionality using asynchronous worker processes communicating over web sockets. It proved beneficial to write the technical documentation after the implementation was already in a stable state, as it forced a re-revision of the implementation, which identified some minor flaws.

One identified flaw was the reading of iOS certificates. Initially the reading was done through an unnecessarily complex binary conversion of the PEM formatted certificate in order for it to be readable in DER format. The re-revision led to a realization that the same could be achieved by simply reading the certificate in the PEM format, removing the unnecessarily complex binary conversion altogether. Additionally, the usage of higher order functions proved in some cases somewhat clumsy. However, changes to this functionality were deferred until something more versatile and flexible would be needed.

7 Summary

The project to improve and simplify the code signing of Android and iOS applications was a success on many fronts.

Firstly, the project highlighted the importance of identifying and planning the needed functionality and scope of a project, before any implementation was to be done. This was something that had not been done by the development team to the same extent before. The project scope could have included a production ready service, but the scope was deemed too broad, and would have required a system upgrade of existing services.

Secondly, the adoption of acceptance test-driven development, while not fully utilized, helped sprint planning and improved task completion time of coding tasks significantly. It also made the requirements for each use case clear, which also sped up the code review process, as the requirements could be used as a reference there as well.

Thirdly, the technological documentation of this thesis will serve as a good entry point for anyone who needs a hands-on guide to codesigning of Android and iOS applications.

Fourthly, the new library not only works, but also serves as an example for future Python projects, as the Python package configurations can be mimicked easily, speeding up development and reducing configuration mistakes.

References

Abhimuralidharan. 2018. What is a provisioning profile & code signing in iOS?. Online source. Medium.com. <https://medium.com/@abhimuralidharan/what-is-a-provisioning-profile-in-ios-77987a7c54c2>. Retrieved 26.7.2018.

Apple. 2011. codesign -- Create and manipulate code signatures. BSD General Commands Manual. Apple.

Apple. 2012. security -- Command line interface to keychains and Security framework. BSD General Commands Manual. Apple.

Apple. 2015. Entitlements Troubleshooting. Online source. Apple. <https://developer.apple.com/library/content/technotes/tn2415/_index.html>. Retrieved 26.7.2018.

Apple. 2016a. About Code Signing. Online source. Apple. <https://developer.apple.com/library/content/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html>. Retrieved 11.1.2018.

Apple. 2016b. Code Signing Tasks. Online source. Apple. <https://developer.apple.com/library/content/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html>. Retrieved 31.7.2018.

Apple. 2016c. Ensuring Proper Code Signatures for Nested Code. Online source. Apple. <https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html#//apple_ref/doc/uid/TP40005929-CH4-TNTAG201>. Retrieved 27.7.2018.

Apple. 2016d. Understanding the code signature. Online source. Apple. <https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/AboutCS/AboutCS.html>. Retrieved 26.7.2018.

Apple. 2018a. Certificates. Online source. Apple. <https://developer.apple.com/support/certificates/>. Retrieved 26.7.2018.

Apple. 2018b. Download manual provisioning profiles. Online source. Apple. <https://help.apple.com/xcode/mac/current/#/deva899b4fe5>. Retrieved 26.7.2018.

Apple. 2018c. Manage Digital Certificates. Online source. Apple. <https://developer.apple.com/documentation/security/certificate_key_and_trust_services/certificates>. Retrieved 26.7.2018.

Fastlane. 2018a. Fastlane. Online source. Google. <https://fastlane.tools/>. Retrieved 12.1.2018.

Fastlane. 2018b. Is this secure?. Online source. Google. <https://docs.fastlane.tools/actions/match/#is-this-secure>. Retrieved 12.1.2018.

Google. 2018a. APK Signature Scheme v2. Online source. Google. <https://developer.android.com/about/versions/nougat/android-7.0#apk_signature_v2>. Retrieved 18.7.2018.

Google. 2018b. apksigner. Online source. Google. <https://developer.android.com/studio/command-line/apksigner>. Retrieved 18.7.2018.

Google. 2018c. App Manifest Overview. Online source. Google. <https://developer.android.com/guide/topics/manifest/manifest-intro>. Retrieved 24.7.2018.

Google. 2018d. Build an unsigned APK and sign it manually. Online source. Google. <https://developer.android.com/studio/publish/app-signing#sign-manually>. Retrieved 18.7.2018.

Google. 2018e. Manage your key. Online source. Google. <https://developer.android.com/studio/publish/app-signing#manage-key>. Retrieved 21.7.2018.

Google. 2018f. Signing considerations. Online source. Google. <https://developer.android.com/studio/publish/app-signing#considerations>. Retrieved 18.7.2018.

Google. 2018g. zipalign. Online source. Google. <https://developer.android.com/studio/command-line/zipalign.html>. Retrieved 18.7.2018.

Krekel, Holger. 2018a. pytest: helps you write better programs. Online source. Holger Krekel and pytest-dev team. <https://docs.pytest.org/en/latest/>. Retrieved 27.7.2018.

Krekel, Holger. 2018b. tox: standardize testing in Python. Online source. Holger Krekel and others. <http://tox.readthedocs.io/en/latest/index.html>. Retrieved 27.7.2018.

Microsoft. 2017. Introduction to Code Signing. Online source. Microsoft. <https://msdn.microsoft.com/en-us/library/ms537361(v=vs.85).aspx>. Retrieved 10.1.2018.

Openssl. 2017. Openssl v.0.9.8 2017-01-18. System documentation. Openssl.

Oracle. 2017. Using JAR Files: The Basics. Online source. Oracle. <https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html>. Retrieved 18.7.2018.

Oracle. 2018. jarsigner. Online source. Oracle. <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html>. Retrieved 18.7.2018

Miracle, Rob. 2014. Tutorial: Understanding Android App Signing. Online source. Corona Labs. <https://coronalabs.com/blog/2014/08/26/tutorial-understanding-android-app-signing/>. Retrieved 26.7.2018

Solodovnichenko, Mikhail. 2017. Extracting stuff from provisioning profiles. Blog post. Mikhail Solodovnichenko. <http://maniak-dobrii.com/extracting-stuff-from-provisioning-profile/>. Retrieved 27.7.2018.

Umbaugh, Brad & Dunn, Craig. 2018. Xamarin.iOS App Distribution Overview. Online source. Microsoft. <https://docs.microsoft.com/en-us/xamarin/ios/deploy-test/app-distribution/>. Retrieved 26.7.2018.

Xu, Zhi & Miller-Osborn, Jen. 2014. Bad Certificate Management in Google Play Store. Online source. Palo Alto Networks. <https://researchcenter.paloaltonetworks.com/2014/08/bad-certificate-management-google-play-store/>. Retrieved 11.1.2018.

Certificate parsing using Python

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
# Filename: ios_cert_parser_example.py
# Usage: python3 ios_cert_parser_example.py

from datetime import datetime
from datetime import timezone
from collections import namedtuple

from dateutil.parser import parse as dateutil_parse  # Third party package

output = ("issuer= /C=US/O=Apple Inc./OU=Apple Worldwide Developer "
          "Relations/CN=Apple Worldwide Developer Relations Certification "
          "Authority\n"
          "subject= /UID=12345ABCDEF/CN=iPhone Developer: MyUserName"
          "(XYZAB12345)/OU=X1Z2Y3ABCD/O=My Organization Ltd/C=US\n"
          "notBefore=Jan 1 00:00:00 2018 GMT\n"
          "notAfter=Jan 1 00:00:00 2019 GMT\n"
          "SHA1 Fingerprint="
          "A1:B2:C3:D4:E5:F6:A1:B2:C3:D4:E5:F6:A1:B2:C3:D4:E0:F0:A0:00\n")

Certificate = namedtuple('Certificate', [
    'issuer',
    'subject',
    'notBefore',
    'notAfter',
    'fingerprint', ])

Name = namedtuple('Name', [
    'countryName',
    'organization',
    'organizationalUnit',
    'commonName',
    'UID', ])


class InvalidCertificateError(Exception):
    pass


def parse_name(name):
    # Using the dict method let's us default to None if the key
    # does not exist.
    return Name(countryName=name.get('C'),
                organization=name.get('O'),
                organizationalUnit=name.get('OU'),
                commonName=name.get('CN'),
                UID=name.get('UID'))


def parse_pkix(data):
    result = {}
    pkix = data.strip().split('/')
    for field in pkix:
        # Parse each PKIX key-value pair
        pair = field.split('=')
        key, value = pair[0], pair[1:]
        if key and value:
            result[key] = ''.join(value).strip()
    return result
```

```
def parse_certificate(output):
    # The output from OpenSSL is in the format of
    #
    #   issuer= CN=Acme Corp Certificate/OU=Acme Corp/...
    #   subject= ...
    #
    # The last data is in PKIX form. Parse this output into a dictionary.
    certificate_dict = {}
    for line in output.strip().splitlines():
        # Separate the entity and PKIX data. Note that
        # partition is used instead of split, since it
        # does not remove '=' characters from the rest
        # of the string.
        what, _, data = line.partition('=')

        if '/' in data:
            # data is for all intents and purposes a PKIX value
            certificate_dict[what] = parse_pkix(data)
        else:
            certificate_dict[what] = data

    if not all(key in certificate_dict for key in (
         'issuer', 'subject', 'notAfter', 'notBefore', 'SHA1 Fingerprint')):
        raise InvalidCertificateError("{}".format(certificate_dict))
    # dateutil_parse will make sure the parsed timestamp is not naive
    return Certificate(issuer=parse_name(certificate_dict['issuer']),
                       subject=parse_name(certificate_dict['subject']),
                       notAfter=dateutil_parse(certificate_dict['notAfter']),
                       notBefore=dateutil_parse(
                       certificate_dict['notBefore']),
                       fingerprint=certificate_dict.get('SHA1 Fingerprint'))


def certificate_expired(cert):
    t_delta = cert.notAfter - datetime.now(timezone.utc)
    if t_delta.days < 0:
        raise InvalidCertificateError("Certificate '{}' has expired.".format(
            cert.subject.commonName
        ))


if __name__ == '__main__':
    cert = parse_certificate(output)
    certificate_expired(cert)
    print(cert)

# Outputs
# > Certificate(issuer=Name(countryName='US', organization='Apple Inc.',
# > organizationalUnit='Apple Worldwide Developer Relations', common-
Name='Apple
# > Worldwide Developer Relations Certification Authority', UID=None),
# > subject=Name(countryName='US', organization='My Organization Ltd',
# > organizationalUnit='X1Z2Y3ABCD', commonName='iPhone Developer: MyUserName
# > (XYZAB12345)', UID='12345ABCDEF'), notBefore=datetime.datetime(2018, 1, 1,
0, 0,
# > tzinfo=tzutc()), notAfter=datetime.datetime(2019, 1, 1, 0, 0, tzin-
fo=tzutc()),
# > fingerprint='A1:B2:C3:D4:E5:F6:A1:B2:C3:D4:E5:F6:A1:B2:C3:D4:E0:F0:A0:00')
```

Generic use cases

Case 1	**As a developer, I want to develop the application using python 3, so I can use the latest libraries.**
Requirements	The application should be developed using python 3 The application should not support python 2
Case 2	**As a developer I want to be able to run the BuildResigner's unit tests from the CLI**
Requirements	The application should use the `py-test` testing library. The application should use `py-test` for running the unit tests. The application should use `py-test` for running functional tests. The application should have a mechanism for extending/suppressing pre-defined `py-test` commands. The test coverage should not be reported by default. Test data and assets required by the tests should be available in the source code. The tests should be runnable using the `python setup.py test` command. The tests should be runnable using the `pytest .` command.
Case 3	**As a developer, I want to have the option to run tests in a virtual environment, so I can test against different python versions.**
Requirements	The application should use `tox` for setting up virtual python testing environments Tox should by default use the installed python 3 version for setting up the testing environment Tox should run the tests using the `pytest` command The testing environment should be setup and run using the `tox` command.
Case 4	**As a developer, I want the source code to be tested through CI, so I can be sure the application does not only work on my development system.**
Requirements	Continuous integration should be setup for the source code using Jenkins The source code should include a Jenkinsfile which specifies the tasks to be performed by Jenkins
Case 5	**As a user I want to install the tool using python's package manager pip, so I can access the individual modules as I see fit.**
Requirements	The application should be installable using the python package manager pip The installation process should handle installation of all required dependencies The version of the application should be retrievable through the CLI
Case 6	**As a user, I want to configure the application, so I can store signing keys and files where ever I want.**
Requirements	The application should be configurable through JSON-file. The application should offer a CLI parameter that allows definition of the JSON-file location. The application should validate the JSON-file on start up using a predefined schema. The application should validate the JSON-file on a platform basis. The application should stop execution in case of an invalid configuration. The application should provide a user-friendly error message in case of a mis-configuration. Configuration should be allowed, but not limited to the following: - Working directory

	- Output directory - Log file location - (Android) Keystore location - (Android) Android SDK build-tools location - (iOS) provisioning files directory - (iOS) keychain name - (iOS) location of JSON-file containing certificate-provisioning file mapping
Case 7	**As a user, I want to configure the signing, so that I get a signed package matching the given configuration.**
Requirements	The application should be able to read configuration files in JSON-format. The application should be able to validate configuration files using a predefined schema. The application should stop execution if the configuration is invalid The application should validate that the configured package file is a valid ZIP-file The application should stop execution if package is not a valid ZIP-file The package is resigned according to the given configuration. The user is informed with a user-friendly error message, if the configuration is invalid.
Case 8	**As a user, I want the application to validate the certificate-provisioning JSON-file configuration, so I can rectify possible misconfigurations.**
Requirements	The application should validate the certificate-provisioning JSON-file using a predefined schema. The application should stop execution if the configuration is invalid. The application should provide a user-friendly error message in case of a mis-configuration. The schema should include the following: - profile - name of the provisioning profile to use (not filename) - string - certificate - name of the certificate as defined in the keychain - string - provision_in_filename - suffix to be added to the resigned file's filename - string - regex validation
Case 9	**As a user, I want the application to unarchive the package being code signed, so that it can be modified.**
Requirements	The application should unarchive the package to the working directory. The application should retain the basename of the original package while processing the unarchived package.
Case 10	**As a user, I want the application to log what it is doing, so I can see if there are any manual actions I need to take.**
Requirements	The application should log info, warning, error and exception level log messages by default.

Android use cases

Case 1	**As a user, I want an Android .apk file code signed as defined in the configuration, so I can distribute it to Google Play.**
Requirements	The application should have support for using a signing keystore The application should ensure the code signed package is zipaligned The application should produce an .apk file The produced .apk file should be found in the output directory
Case 2	**As a user, I want to process an Android .apk file without code signing it.**
Requirements	Code signing of Android packages should be disableable in the configuration The application should not change the signature of the package The application should not modify the contents of the package The application should produce an .apk file The produced .apk file should be found in the output directory

iOS use cases

Case 1	**As a user, I want an iOS .ipa file to be code signed as defined in the configuration, so I can test I on a provisioned device.**
Requirements	The iOS subschema should contain at least the following optional attributes: - bundle_id - string - version_number - string - regular expression validation - short_version_string - string - regular expression validation - associated_domains - array of strings - provisioning_id - string (a key to look for in the provisioning/certificate mapping)
Case 2	**As a user, I want the application to update an iOS .ipa file's Entitlements.plist based on the existing Entitlements.plist and the one defined in the provisioning profile, so that the build functions as expected.**
Requirements	The application should be able to read the Entitlements.plist in the unarchived package. The application should be able to read provisioning profiles. The application should create a new Entitlements.plist based on the old implementation. The application should be able to write .plist files
Case 3	**As a user, I want the application to add the provisioning profile defined in the configuration to the .ipa package, so that it can be installed on provisioned devices.**
Requirements	The application should be able to validate provisioning profiles The application should stop execution if the configured provisioning profile is found invalid. The application should produce a user-friendly error message if the provisioning profile is invalid. The application should forcefully copy the provisioning profile to unarchived .ipa package
Case 4	**As a user, I want an iOS .ipa file's bundle identifier to be updated as defined in the configuration, so that the bundle identifier can be changed as needed.**
Requirements	The application should be able to read the Info.plist file in the unarchived package. The application should be able to edit the CFBundleIdentifier property of the Info.plist file. The application should be able to write the Info.plist file in binary format.
Case 5	**As a user, I want iCloud entitlements in iOS .ipa files to be updated by combining entitlements from the original .ipa file and the new provisioning profile, so the application has access to the required iCloud services.**
Requirements	If and only if the key “com.apple.developer.icloud-services” exists in both the original entitlements and the provisioning profile entitlements set the iCloud entitlements: - Set “com.apple.developer.icloud-services” as the value in original entitlments. - Set “com.apple.developer.ubiquity-container-identifiers” as the value in

	the provisioning profile entitlements if the key exists. - Set “com.apple.developer.icloud-container-environment” as the value in the original entitlements if the key exists.
Case 6	**As a user, I want the associated domains related entitlement property to be a combination of the original .ipa file and the provisioning profile, so that ...**
Requirements	If and only if the key “com.apple.developer.associated-domains” is defined in the provisioning profile and the optional attribute “associated_domains” is defined and truthy in the configuration, set the ”com.apple.developer.associated-domains” entitlment as the value defined in configuration.
Case 7	**As a user, I want to update an iOS .ipa's build number when code signing, so that the package does not have to be remade.**
Requirements	The iOS subschema should contain the optional attributes “version_number” and “bundle_version_string”. The “version_number” and “bundle_version_string” attributes should be strings of the form “x.x.x” where x is an integer. If the “bundle_version_string” or “version_number” are defined in the configuration they should be updated in the Info.plist file accordingly.
Case 8	**As a user, I want the application to archive the code signed package, so that it can be installed on iOS devices.**
Requirements	The application should archive the code signed package without adding the working directory as the root directory of the archive. The application should name the archived package based on the postfix defined in the provisioning/certificate mapping.

Parsing provisioning profile with a Python regular expression

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

import plistlib
import re

EXAMPLE_PROFILE = """<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
     <key>AppIDName</key>
     <string>App ID Name</string>
     <key>ApplicationIdentifierPrefix</key>
     <array>
          <string>ABC123DEF4</string>
     </array>
     <key>CreationDate</key>
     <date>2042-10-15T15:20:42Z</date>
     <key>Platform</key>
     <array>
          <string>iOS</string>
     </array>
     <key>DeveloperCertificates</key>
          <array>
          <data>Long human-unreadable Base64 encoded certificate=</data>
          </array>
     <key>Entitlements</key>
     <dict>
          <key>keychain-access-groups</key>
          <array>
               <string>ABC123DEF4.*</string>
          </array>
          <key>get-task-allow</key>
          <false/>
          <key>application-identifier</key>
          <string>ABC123DEF4.com.your.bundle.id</string>
          <key>com.apple.developer.team-identifier</key>
          <string>U1R23TEAMID</string>
          <key>aps-environment</key>
          <string>production</string>
          <key>beta-reports-active</key>
          <true/>
     </dict>
     <key>ExpirationDate</key>
     <date>2044-10-15T15:20:42Z</date>
     <key>Name</key>
     <string>Profile Name</string>
     <key>TeamIdentifier</key>
     <array>
          <string>U1R23TEAMID</string>
     </array>
     <key>TeamName</key>
     <string>Your team name</string>
     <key>TimeToLive</key>
     <integer>364</integer>
     <key>UUID</key>
     <string>thats-your-profile-uuid-string</string>
     <key>Version</key>
     <integer>1</integer>
</dict>
```

```
</plist>
Some garbled signatures...
Some garbled signatures...
Some garbled signatures...
"""

# Compiling a regular expression consumes a lot of resources so let's use it
# globally in the module's namespace
PROFILE_MATCHER = re.compile(r"(?=\<plist)\<plist(?:.*\s)*\<\/plist\>")


def from_string(content):
     """Parse the <plist></plist> content of a CMS-formatted string."""
     match = PROFILE_MATCHER.search(content)
     if match:
          try:
               return plistlib.loads(match.group(0).encode(),
use_builtin_types=False)
          except plistlib.InvalidFileException:
               raise ValueError("Invalid provisioning profile content")


def from_file(path):
     """Parse a provisioning profile from a file."""
     with open(path, 'r', encoding='unicode_escape') as f:
          return from_string(f.read())


if __name__ == '__main__':
     print(from_string(EXAMPLE_PROFILE))
```