Securing applications from commit to execution for Android apps and Jar files.

Year: 2017/18
Module: Final Year Project (IY3821)
Supervisor: Dr Jorge Blasco Alis

Contents

Chapter 1

Introduction

1.1 Motivation

A lot of applications are now signed to prevent tampering with them, this usually consists of a created executable that is signed by the company or group releasing some software, allowing the end user or their system to verify the source of the software. The signature usually covers the executable code, but also the metadata, and the resource files, allowing prompts to show who signed a piece of software, and stopping manipulating the resource files to for example swap out error messages for security issues with benign looking messages.

For some classes of software (device drivers on windows for example), this is a mandatory step, as 64 bit editions need settings changes to allow the use of unsigned drivers[1]. This was introduced to try to prevent malware from getting deep into the system, as once it is at such a layer not only does it have the ability to take over the system, but also hide itself from any software looking for it.

Unfortunately this is just moving the insertion of malicious payloads to before the signature, such as the CCLeaner[2] attack, which exploited exactly this loophole (This has not been confirmed to be how the attack took place, but analysis by Cisco[3] point to this, and no other explanation has been provided). This kind of attack generally works by injecting malicious code into the repositories used to build the executables that are then signed for release holding their payload.

Trying to defend against this attack is difficult, but there are some things that can help. One example of this is that git allows for signed commits[4], which can help to prove the author of the commit. This does introduce the issue of validating all commits are signed though, and the storage of keys for developers, although yubikey style devices [5] or smartcards can help reduce the difficulty of this.

As I would like to gain more experience using Kotlin, and JGit is written in Java, I will write the majority of the code for this project in Kotlin, which is cross compatible with Java (Java code can access all methods and fields of

Kotlin code and vice versa, and they can run together in the same JVM. This is possible as Kotlin compiles to Java bytecode, rather than having its own format. Kotlin is also a first class language on Android[6], so the apps that are needed can be developed in it too, without needing to be translated to Java.

By having code that can run on both computers and Android devices, I can make simple versions of the systems, package them as JARs and test them locally, and once they work I can integrate the logic into an Android app, as an APK file. By using this 2 stage pattern I can test locally, which is more convenient than needing to use a device for testing, and also as a computer can have more resources I can run tests that are more exhaustive in less time.

1.2 Objectives

This project looks to create a proof-of-concept defence against such an attack with the following features:

1. Developers should not need to install new software on their own devices if possible (although this may be necessary in some cases, this should be avoided if possible).

2. The releases are signed in such a way that end user devices can be sure who committed and released an application, and there should be no breaks in the chain of custody of the source or artefacts created from it.

3. The releases should not be able to be tampered with in an undetectable way.

4. It should not be possible to include an entity (person, group or company) in the list of people who released an application without their permission.

1.3 Limitations

SHA1 limitations
: Git only supports SHA1 commit IDs, despite Shattered breaking SHA1[7], which means that for identifying a commit this project only allows the use of SHA1. There is some argument [7][8] about if this attack is practical against a git repository, but as a defensive measure the systems designed in this project are designed to be able to be used with the output of different hash algorithms, so it will support the new algorithms as soon as git does.

Mallicious developers
: This project only deals with trying to verify that the author of the code is the one that the entity that the user believes it is, and that they can trust them, it does not deal with trying to make sure the author is not writing malicious code.

Build Server compromise

We assumes the build server is a secure environment, as otherwise we can't guarantee our code is running, as the build system could be replaced with one that signs a trojan with the securely generated chain.

Trust Establishment

This project uses Keybase as a secure link from identity to keys. This can be used for either a social user, or a website, which can relate to a single person, a group or another kind of entity like a company.

The use of keybase is abstracted away, so in theory any other trust mechanism could be used, such as one of:

- The local GPG keychain
- A keychain relayed over a secure medium (such as via a diplomatic bag, or pre-installed into a device)
- TXT records relayed via DNSSEC (for linking a key to a website)

Chapter 2

Background Theory

2.1 Trust

This project allows for trusting the releasing entity for a given project, this means that a company can for example lock down their machines, only allowing software from approved vendors to be executed. There are a set of properties that may want to be asserted about an executable:

- Who released an executable.

  This would be used to assert that a trusted company made a piece of software such as a driver, rather than it being a fake version, which may be used for malicious purposes.

- Which developers have been involved in the creation of an executable.

  This is used for example to say who made an open source library, so that if a malware author injects code into a project they cannot then claim to have not done it.

- That a developer has not been involved in the creation of an executable

  This would be used so that a malicious party cannot create malware, and then claim it was created by an uninvolved party instead of themselves.

- That an executable has not been modified since its release.

  This is used to check that malware has not been injected into an executable after it was built.

2.2 Digital Signatures and PKI

Digital signatures allow an entity to sign or encrypt a message with a public key. This can be used to sign code, so that the entity which released the code. It can also be used to chain trust, so you can trust a root certificate, which can

then trust intermediates, which can then trust leaf certificates which are used. This is how certification authorities are able to sign certificates for sites which browsers will trust.

2.2.1 GPG

GPG is a tool which allows for signing and encrypting of messages, and can armor the resulting message, making it suitable for sending via email.

2.3 Code signing

Code signing is the process by which an entity can assure that an executable was released by them, and has not been modified after that point. There are many uses of this in real world situations, but some of the more common are Android apps, IOS apps, Authenticode and signed JARs.

2.3.1 Android Apps

For android apps a certificate is used to sign an application, but the certificate does not need to be part of a trust chain, it is just trusted the first time it is seen, and while the app is installed this certificate cannot change. This does reduce the overhead needed to create an app, as the developer does not need to get their certificate signed, but it has 2 main downsides, firstly the app could be a fake, and as long as it is not already installed the fake certificate will be trusted, and second if the certificate is lost the app can no longer be updated, and to make a new version the old version must be uninstalled and an app with a new name and cert must be released.

2.3.2 Authenticode

Authenticode works similarly to Android apps, with but the cerificate must be signed and trusted. This causes issues for developers of open source projects, as code signing certificates usually cost at least $100 a year, and can be a lot more expensive. Another issue is that the amount of CAs that are trusted for code signing and issue code signing certificates is relatively small, so there is not much competition.

2.4 Reproducible builds

There are multiple projects which are trying to make their builds reproducible, such as debian[9], and even some which aim to allow multiple parties to sign their build result and then use this to try to validate the build, one example of which is gitian[10].

While these try to solve the same issue I am trying to, they rely on the end user being able to compile the project, which in some environments may not

be possible (embedded devices and mobile for example). It is possible a future project could look at a hybrid of the 2 strategies, with an aim of getting the best of both worlds, where multiple releasing entities could sign a reproducible build with the signature scheme designed in this project.

Chapter 3

Project Diary

As the original plan was changed, a lot of these points do not relate directly to the points in the plan. There were also a lot of delays added, both because of delays in designing the signature scheme, and also in trying to make the git connector, which was originally just going to be a facade around JGit, but which needed a lot more work than expected, as JGit is missing many key features. Further problems with the output of git delayed the project even further, as it turns out that git returns incorrect data in some situations.

25th September
: Project was chosen.

5th October
: Created git connector module, which allows the project to check the status of a git repo and get individual commit information. This is later than originally planned, because it turned out that JGit does not support signed commits, which added considerable delays to the project.

9th October
: Got travis working with the repo, so that builds can be automatically ran against commits.

10th October
: Successfully read commit details from the repository.

13th October
: Whole roject was moved to gradle to simplify the build process, rather than just 1 module in a directory, allowing for adding future models.

18th October
: Created the jarInfo and demoJar module. The demo jar was originally scheduled for being done by the 10th November, but that did also include the Android app, which has not been created yet. The jarInfo project is a key part of the signer, and the validator.

During this time gap I was working on the signature design, and dealing with other assignments.

24th November
: Created the JarInjector module, also created the GPG connector module

25th November
: Reworked the build enviroment setup script to make it easier to use, and allow for more scripts later on. This added delays but it should reduce delays later on.

26th November
: Created the Keybase connector, also reworked the whole build system so everything is only configured in the root project, so that all the modules follow a common set of standards.

27th November
: Create the tools that check the repo setup and detect if the repo is in a state that would previously allow it to fail silently.

28th November
: Fixed issue with git error messages being different on the teaching server, which broke the error detector. Also created an issue on the Keybase issue tracker to record the issue with the API where a request that should get back the list of linked accounts for a given domain returns nothing.

Chapter 4

Software Engineering

4.1 Key points

4.2 Patterns

4.2.1 Singleton

The use of Kotlin simplifies using patterns, the following classes objects, which in Kotlin are implemented as thread safe singletons[11].

- uk.co.jrtapsell.fyp.gpgWrapper.GpgWrapper
- uk.co.jrtapsell.fyp.processTools.extensions._Thread
- uk.co.jrtapsell.fyp.keybaseConnector.Keybase
- uk.co.jrtapsell.fyp.keybaseConnector.dataclasses.KeybaseMapper

4.2.2 Factory

Using companion objects in Kotlin factories can be implemented in relatively small amounts of code, one example of how this works is the companion object of uk.co.jrtapsell.fyp.gitWrapper.data.Commit, which creates Commits from their JSON representation.

4.3 Automation

This project makes heavy use of automation, all of the modules have automatic tests which are called on every commit by travis, which allows for validation and detection of regressions.

Normally, this would be used with GitHub protected branches to prevent broken code getting onto master, but as the merge commits need to be signed

this cannot be used, in the final project there will be a web UI for GitHub which allows for fast forward merging only passing commits, which will prevent broken code ending up on master, without requiring its own signing key, as branches are not signed in git.

4.4 TDD

I used TDD throughout the project, with tests being written first and then the code being written to pass the tests. This meant that it was easy to tell if the code was broken or not. I also used travis to allow commits on GitHub to be signed.

4.5 Signed Commits

All commits to the repository were signed with a GPG key stored in a smart card, which allows me to be certain no party could inject fake commits into the GitHub repository without somehow getting them signed with my key, which shouldn't be possible, unless they can create a commit with a matching hash. This would be important in a real environment as attackers have been known to attack suppliers in order to get access to their clients.

Chapter 5

Timeplan

5.1 Original plan

- Stage 12nd Oct to 13th Oct
- Stage 216th Oct to 27th Oct
- Stage 330th Oct to 10th Nov
- Stage 413th Nov to 26th Nov
- Stage 527th Nov to 10th Dec
- Stage 68th Jan to 19th Jan

5.2 Current plan

Due to delays and unforeseen issues, it was better to create the utility parts first, and connect them together afterwards, rather than try to make the parts in their original planned order.

5.3 Statuses

- Signature design
  A base of how the signatures will be calculated and stored, this has to be completed first in order for the other parts of the project to be done, as they all will use the signature format.

  Completed, the signature design is shown in the architecture diagram above.

- Jar Signer
  A tool that takes a working repo and signs it with a given key or set of keys. This will be the first test of the Signature design, so it needs the signature design completed first.

  Created, needs to be packaged, this works as part of jarInjector, but needs to be packaged as part of a program in order to be fully useful.

- Jar Validator
  A tool that validates a jar file and shows the details of the keys that wrote and released it.

  Created, needs to be packaged, this is the validator of jarInfo, but it needs to be extended to be able to test against arbitrary keys.

- Jar Launcher
  A tool that launches a jar file after asking a user if they trust the developers and/or breleasers.

  To be created

- Demo App
  A basic app with all commits signed.

  JAR version complete, APK version to be made

- App Signer
  Signs an app and creates a signed release.

  JAR version needs packaging, APK version to be made, the jar version is part of jarInjector, but it needs a UI added to make it fully usable, the APK version needs to be modified to be made useful.

- Signed App
  An app signed which proves the developers and/or releasers.

  Needs to be made

- App Validator
  Checks if an app is valid and shows the signature details. **Needs to be made**

- Validating Installer
  An app which replaces the android installer, and shows the signature details

Chapter 6

Parts completed

demoJar
: This is a simple project used to make a clean testing JAR, it is a simple application written in Kotlin, which simply prints a message and exits. It is build into a jar, which can then be signed, injected into and can be used for other tests.

gitWrapper
: Wraps git, and allows for getting repo status and commit information back without each part that uses git having to call out to the git executable manually.

gpgWrapper
: This module wraps the GPG executable, and allows the system to sign messages with keys stored in memory, on disk, or on a smart card. It can also validate messages against either the default keyring, or against a set of known keys.

jarInfo
: Allows for reading of information from JAR files, including from their manifest and other files, and it also allows for getting the signature status of the files from the commit.

jarInjector
: This tool allows for manipulation of jar files that are stored on disk, this includes being able to sign them, unsign them, and add and remove files. This will be used by the signature system to clean any existing signatures from the jar file, before it injects the relevant information into the jar and signs it with the RSK.

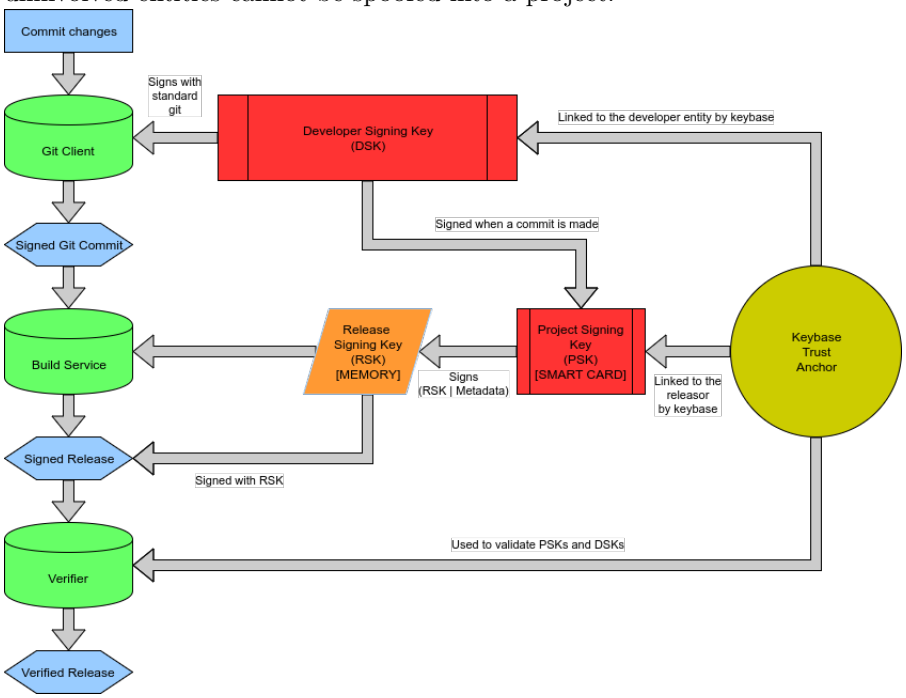
keybaseConnector
: Connects to Keybase via their API and allows for details of an identity to be retrieved. This acts as a trust anchor for the project, and also means that if a key is compromised the entity can just remove it from keybase,

and releases signed with that key will then become invalid, which reduces the amount of harm that a compromised key can cause to a company.

processTools
: This helps with calling external processes, communicating with them, and checking exit codes, so that all the modules do not need to repeat this, causing duplicated code and allowing for inconsistent behaviour.

projectValidator
: This checks the project is setup correctly, all tests are hooked correctly and that the packages are correct. This makes sure that there are no configuration errors that would make the project appear to be ok, while it is actually broken.

Signature design
: I have designed the signature scheme below, so that the entities involved in the release of an artefact can be proven to be involved, and so that uninvolved entities cannot be spoofed into a project.



Chapter 7

Apendix

7.1 Folders

documents
: Contains the documents (report and presentation)

project
: Contains the actual project code

7.2 Tests

To run the tests:

- Open a terminal in the project directory.
- Run the 2 scripts in setup
- Run ./gradlew test

Chapter 8

Bibliography

[1] "Driver signing policy." https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy–windows-vista-and-later-, 2017.
This lists the signing policy for windows drivers, which shows it is mandatory on recent versions of windows using secure boot, which is a mandatory part of the windows hardware specs[12].

[2] A. Greenberg, "Software has a serious supply-chain security problem." https://www.wired.com/story/ccleaner-malware-supply-chain-software-security/, 2017.
This was a good example of malicious code being injected before signatures, getting around the current protections.

[3] L. Tung, "Hackers hid malware in ccleaner pc tool for nearly a month." http://www.zdnet.com/article/hackers-hid-malware-in-ccleaner-pc-tool-for-nearly-a-month/, 2017.
This is second source about the CCleaner hack.

[4] "7.4 git tools - signing your work." https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work, 2017.
This shows that git supports signed commits, proving their author or commiter.

[5] "Git signing." https://developers.yubico.com/PGP/Git_signing.html, 2017.
This shows yubikeys can be used to store keys used for signing commits, proving that they were created by the person that was expected, this helps to simplify managing keys, and excluding an attack against the yubikey keeps the key safe even when inserted into malicious machines.

[6] M. Shafirov, "Kotlin on android. now official." https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/, 2017.

This shows that Kotlin is now a first class language on android, which allows it to be used for this project, although before it would have required additions to the toolchain it is now fully supported.

[7] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "Shattered." https://shattered.io/, 2017.
SHAttered attack, demonstrating a pair of colliding PDFs.

[8] L. Torvalds, "Re: Sha1 collisions found." https://marc.info/?l=git&m=148787047422954, 2017.
Linus Torvalds' thoughts on the effects of the SHA1 collion on git.

[9] "Reproduciblebuilds." https://wiki.debian.org/ReproducibleBuilds/About.

Debian wiki on reproducible builds.

[10] "Gitian: a secure software distribution method." https://gitian.org/.
A project that allows for signing and trusting signatures to validate a build.

[11] "Objects in kotlin: Create safe singletons in one line of code (kad 27)." https://antonioleiva.com/objects-kotlin/. This shows how Kotlin objects are thread safe Singletons.

[12] "Hardware compatibility specification for systems for windows 10, version 1607." https://docs.microsoft.com/en-gb/windows-hardware/design/compatibility/systems#systemfundamentalsfirmwareuefisecureboot, 2017.
This shows that secure boot is a mandatory part of the requirements for windows 10.