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

James Tapsell

Contents

Overview

- It used to be common to see injection of code into executables and then these malicious executables being distributed.
- Code signing has made these attacks less common, but has caused the malicious code to be injected to the codebase before it is signed insead.
- This project aims to extend this trust chain from the commit all the way to execution.
- As APK files are similar internally to JAR files, and Kotlin is a first class language in both environments, the approaches used for JAR files should also apply to APK files, with some modifications.

Limitations

SHA1 limitations

Git only supports SHA1 commit IDs, despite Shattered breaking SHA1, so this project has to use SHA1.

Mallicious developers

As this project does not do static analysis it cannot detect bad code.

Build Server compromise

We assume the build server is a secure environment, as otherwise we cannot guarantee our code is running.

Trust Establishment

This project uses Keybase as a secure link from identity to keys, alternatives could be used instead.

Tools Used

Keybase

A service for linking identities and keys, used to allow trust of a website to be changed into trust of a key securely.

Kotlin

A JVM language, which allows for compact code that is null safe, with many other benefits. (Also allows tests to have names that fully describe their purpose.)

Gradle

Build management, and automated test running.

Tools Used (cont.)

CodeBeat

Static code analysis, looking for code smells and security issues.

TravisCI

Automatic test running in the cloud, to guarantee reproducible test results, and prevent code only working on 1 machine.

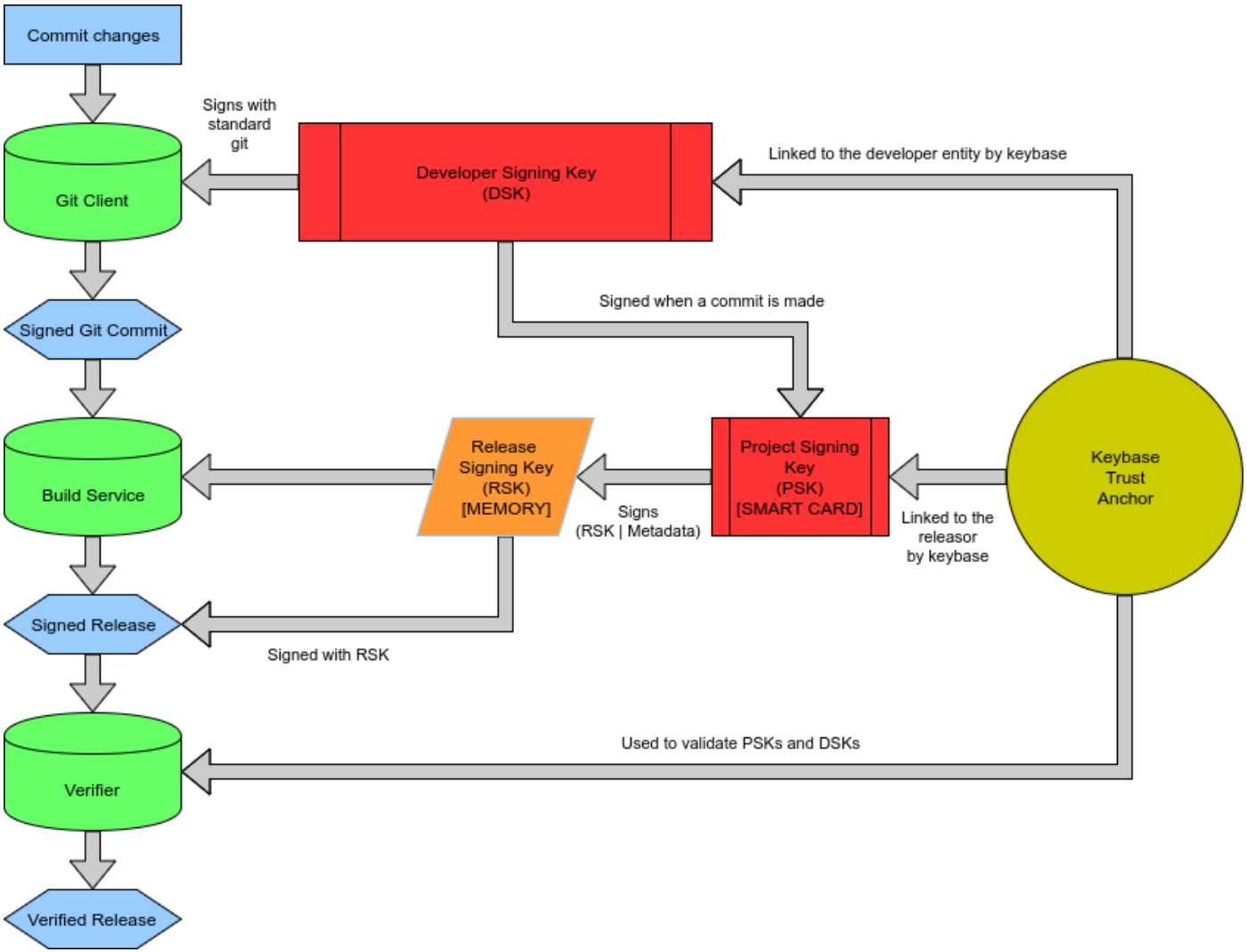
IntelliJ IDEA

IDE, built by JetBrains (who created Kotlin), with many features to support the language.

Other

Other standard tools (git, gpg, jarsigner etc.) were also used, but are omitted here due to space constraints.

Architecture Diagram



Signature Generation

- First the signing server generates a unique RSK
- The public half is stored in the jar
- The signatures of the PSK with the developer's DSKs are stored, so that uninvolved developers cannot be spoofed.
- The public half of the PSK is stored in the jar, with a reference to its entity's username.
- The PSK signs the RSK, along with some metadata, this signature is added to the jar
- Optionally, developers can sign the RSK (along with metadata) with their DSKs, which allows them to assure a given build.
- Finally, the JAR is signed with the RSK, which is then wiped from memory.

Signature Verification

- The public keys and signatures are extracted from the JAR.
- The JAR is validated against the public key that is contained inside of it (the RSK).
- The signature of the RSK with the PSK is then checked, which proves that the release was made on behalf of a given project.
- For each of the DSKs listed in the project the keys are verified, if the DSKs have signed the PSK then they can be treated as contributors to the project.
- The set of DSKs that signed the RSK is checked, using this it can be checked who assured a build.

Completed Items

Line Count

1161 Kotlin + 78 Gradle

demoJar

This is a simple project used to make a clean testing JAR

gitWrapper

Wraps git, and allows for getting repo status and commit information

gpgWrapper

Wraps gpg, and allows for signing and verifying messages, and also key management

jarInfo

Allows for reading of information and validation of JARs

Completed Items (Cont.)

jarInjector

Allows for injecting, signing and unsigning JAR files

keybaseConnector

Connects to Keybase and allows for getting details for a given identity

processTools

This helps with calling external processes, communicating with them, and checking exit codes

projectValidator

This checks the project is setup correctly, all tests are hooked correctly and that the packages are correct

Issues Faced

Keybase API issue — [GitHub issue]

The official Keybase API docs had issues getting signers for a given domain, which I have raised an issue about.

Runtime JAR signing

This is not possible using the JDK tools, so I use jarsigner.

Git inconsistencies

Although there is an unknown key return value, git returns unsigned for unknown keys, which breaks checking.

Git only allows 1 signature

So if the committer and author are different they cannot both sign.

Going Forward

Linking modules

The modules need to be linked together so that they can be used as one system together.

Web merging UI

The web merge tool that preserves signatures needs to be made