# What even is code signing in iOS?

APP DEVELOPMENT

iOS

Bruno Rocha

MARCH 28, 2024

If you have been developing for iOS for a while, chances are you had one of two issues involving **code signing**, the process where Apple/Xcode forces you to "sign" your app with a developer certificate in order to be able to archive it and submit it to the App Store.

I find code signing to be interesting not just because of what it does, but because it's one

under the hood. We just follow Apple's convoluted steps on how to make it work and move on with our lives.

In practice, code signing is an incredibly important safety feature of Apple's ecosystems. Knowing what it is and why it exists makes debugging issues related to it considerably easier, so I've written this article to help you understand it.

## Modern internet security: From sort-of safe to pretty safe

Code signing is not something that was invented by Apple. Rather, it's just a different name for the same security practices that power most of the modern internet. In order to understand code signing, we need to take a step back and explore what those broader security practices are.

One of the earliest ways of encrypting content on the internet came in the form of **symmetric cryptography,** which dictates algorithms that can encrypt and decrypt a message given a single special "secret key" or "password":

The word "symmetric" is used here because the same key is used for both encryption and decryption. This form of encryption works fine for things that happen in person, but not so much for things that happen entirely over the internet.

Imagine you're trying to communicate with someone over the internet, and you'd like to do so in an encrypted fashion. You need to agree on which secret key to use, but you don't have the option to meet in person. How can you communicate this secret key to each other?

It's easy to think that we could just send our secret keys to each other via the web, but that wouldn't be safe in the context of this example. If someone's secretly watching your network, they could intercept this message and get a hold of your secret key. That would be an absolute disaster as this person would not only be able to read what you're doing, but they would also have the power to **pretend** to be either of you by encrypting their own messages!

## Enter asymmetric cryptography

does both the encryption and decryption, the algorithms rely on a **pair of keys** that encrypt/decrypt each other's messages:

- Message + Key 1 = Encrypted message
- Encrypted message + Key 2 = Message
- (or, vice-versa:)
- Message + Key 2 = Encrypted message
- Encrypted message + Key 1 = Message

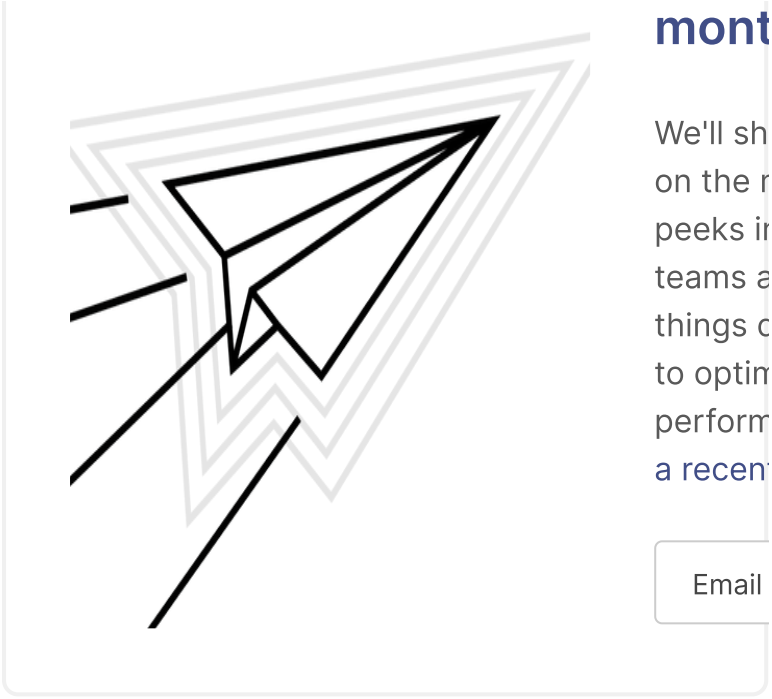
As you can see in the above example, one important feature of those algorithms is that the encryption also works in reverse. It's not only that messages encrypted by key one can only be decrypted by key two, you can also do the opposite.

Now, assume that we created one such pair of keys and that we sent one of the keys **(but not both!)** to the person with whom we're trying to communicate, via the web. Can you see why this is safer than the symmetric encryption example?

In this example, even if someone manages to intercept the key we sent over, we already have something that is safer than the first scenario. While the bad actor would theoretically still have the ability to decrypt what you're sending to the other person, **they**

In practice, the pair of keys we're talking about here is usually referred to as the **public/private key pair** precisely to denominate the use case we've described here. The "public" key is named as such because it's what you transmit across the web and don't care if a bad actor has access to it, while the "private" key is what you keep to yourself and protect at all costs. In iOS development, you very likely had to directly deal with these more than once not just because of code signing, but also because most modern SDKs rely on you setting up such keys in order to protect your account.

But as you might've noticed, this is not exactly 100% secure either. If the bad actor were to **intercept** the public key you sent over and replace it with a bogus one that they created themselves, they would not only be able to pretend to be you again, they would completely be able to lock you out as your original secret key wouldn't be a valid pair for their bogus one.




## Certificates and Certificate Authorities (CAs)

To prevent disasters like that, systems where security is critical have an additional layer of protection on top of those public keys.

Instead of blindly sending the raw public key over the web and hoping for the best, such systems bundle that public key within a file that aims to prove to the recipient that the public key they're looking at came from where it was supposed to come from and can be trusted that it wasn't modified or intercepted. Such files are called **certificates.**

The way the proof works is by having an entity (like a person or a company) encrypt a

entity's public key and confirm that what they described is in fact what you received, then you have proof that the certificate and its bundled public key did in fact come from that entity. If you can trust that the entity is not lying to you, then you can trust that the public key is legit. In the security field, these entities are called **Certificate Authorities (CAs),** and this process of encrypting small "proof messages" into files is what's referred to as **signing.**

This system is called **chain of trust**, and as the name implies, its security relies completely on you having trust in the CA and everyone else involved in that chain. Because of that, while technically anyone can be a certificate authority, you'll find that the internet mostly relies on a small group of large and heavily audited ones. Apple is one of them.

For additional security, certificates themselves usually come in the form of a chain. Instead of being signed by a single "main" private key that handles everything pertaining to that CA, your end-user certificate will instead usually be signed by the private key of *another* certificate that is more specific to your use case. Such certificates are called **intermediate**

certificates without losing the ability to validate the certificates at the end of the chain (the ones we developers deal with). Much like your end-user certificates, intermediate certificates are themselves signed by other certificates, in a chain that goes all the way up to a single **root certificate** lying at the top which effectively owns all other certificates in the chain.

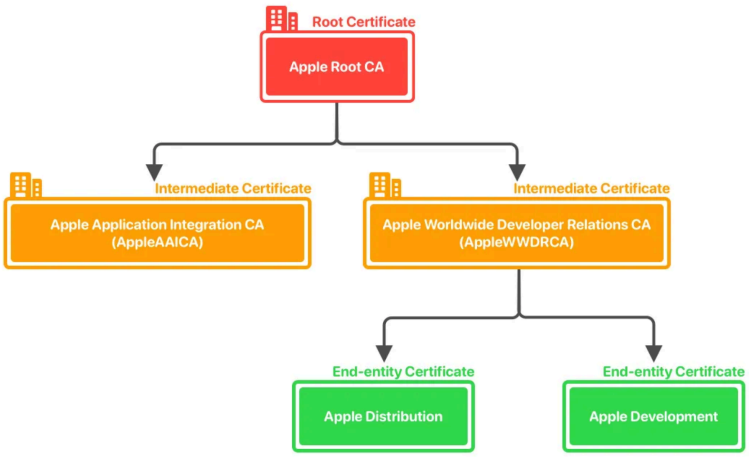



Diagram from Bing Kuo's Beginner's Guide to Code Signing

Other important security features of CAs and certificates worth mentioning include **expiration dates** and **the ability to revoke certificates** in cases where they cannot be trusted anymore (such as a leak).

In the case of Apple specifically, you can find information about all of the root and

"But wait, this doesn't solve the problem we talked about. What if a bad actor forges the public keys of a CA?", you might think. "Wouldn't they then be able to create their own chain of fake certificates and cause damage that way? "

If every single public key were to be transferred over the web, then theoretically yes. Luckily, the internet doesn't do that! To prevent forgery, products that rely on certificates as their security mechanism **generally hardcode the root certificates that they care about.** By shipping those critical public keys alongside their code, attacks involving intercepting keys via the network become impossible. As an example, Apple's root certificates are hardcoded in your iOS/macOS devices. Another example is your browser, which comes with root certificates from CAs related to HTTPS.

When put together, this system of public/private keys and chains of trusted certificates/CAs is called **Public Key Infrastructure (PKI)**. PKI is a critical component of modern secure communication and plays a central role in ensuring the confidentiality, integrity, and authenticity of digital transactions and data exchanges.

**prove that some piece of code is legitimate.** In other words, this is a security feature to prevent unauthorized or malicious code from running on the user's devices.

When you create your developer account, the process of uploading something to the App Store involves having you ask Apple to create a new certificate for you. This results in Apple giving you a private key that you're meant to store securely in your Keychain, which is then used by Xcode to sign your app bundles before finally sending them to Apple. Since Apple has access to your certificate, they have the power to prove that those bundles actually came from you, and not someone pretending to you.

In macOS, the process of having Xcode sign your app bundles is deep-down handled by the codesign CLI tool, and is a tool that I recommend learning how to use as it can be very handy when attempting to debug code signing problems in general.

## iOS continued: Device safety with Provisioning Profiles

platforms and have otherwise nothing to do with PKI.

While on the one hand the purpose of code signing is to make sure a particular piece of code can be trusted, the purpose of provisioning profiles is to make sure a particular device is allowed to run that same particular piece of code.

While platforms like Android allow you to download binaries from third-party app stores and install them on your phone without any restrictions, iOS forbids you from doing so unless the developer explicitly acknowledges during the submission process that your specific device is allowed to install that particular binary. This is because iOS is designed to be a closed ecosystem, and is the reason why you cannot for example install apps from a source that is not the official App Store. This is a critical component of the overall safety of Apple devices, although most iOS users would agree that a little bit more freedom would be appreciated. (In fact, the EU has long been attempting to force Apple to relax those restrictions, and it seems likely that they will succeed.)

You already know what these acknowledgments are: that's the provisioning

inspected by the device trying to install your app to confirm that it's actually allowed to do so.

Developers are required to provide provisioning profiles when targeting physical devices, and they contain information such as the bundle identifier of the app, the certificates tied to it, and the capabilities of the app (e.g camera access).

All builds targeting physical devices require provisioning profiles, but only debug and ad-hoc builds require an explicit list of devices. For Enterprise and App Store builds, profiles do not need to include device data since they refer to builds where the users are not known in advance.

## How code signing is handled in release processes

While Xcode can automatically manage code signing for you, one of the first things you learn in iOS development is that you should turn it off and never EVER press that darned "Fix Issues" button.

teammates, automatic code signing is guaranteed to cause some serious damage to your release process if you don't turn it off.

The reason is that you need to have access to your developer certificate's private key when submitting your app to the store or compiling a build that targets a physical device. If Xcode complains that you don't have a copy of the private key on your machine and you click the "Fix Issues" button, Xcode will not magically fetch the key from your teammate's computer (it couldn't possibly do so); it will instead **revoke** their certificates and create a brand new one for you. This means that the next time your teammates try to compile the app, they will be greeted by a message telling them that their certificate is not valid anymore. If everyone keeps on pressing that button, your team's developer account may end up with an ocean of dead certificates, and if one of those teammates happens to be a CI server, your entire release workflow will halt every time it happens.

To avoid that, teams disable Xcode's automatic signing features in favor of using external tools that were specially designed to solve this problem. Perhaps the most popular one is match, which is a plugin for fastlane that stores and manages all of your team's

their machine and that there are no duplicate/outdated certificates or profiles in your team's developer account.

In essence, iOS code signing is not just a formality; it's a crucial security feature used to verify the legitimacy of an app and protect users from malicious software. While you certainly don't need to master the details of how code signing works to be able to develop your apps, knowing a little bit of PKI and how it's used in practice in iOS can come in handy not just for debugging iOS-related signing issues, but also for helping you understand the modern internet in general.

## RELATED POSTS

### WWDC 2021 highlights

Alessandro Martin

JUNE 10, 2021

### How to manage the mobile release process

Gabriel Savit

JANUARY 12, 2024

### The hidden and not-so-hidden costs of inefficient mobile releases

Richard Huffaker

MARCH 22, 2024

Email