

Product

Integrations

Automations

Customers

Pricing

LOG

The 'what' and 'why' of iOS signing certificates and provisioning profiles — and how to manage them as your team grows

APP DEVELOPMENT

fastlane iOS



Bruno Rocha

MARCH 28, 2024

"just work". Apple has done a good job of abstracting away the inner workings of **code signing** so that, today, you don't need to understand the ins and outs of cryptographic theory to create a fully working iOS app and make it available to the world. But as your team grows, you'll notice that Apple's abstraction only goes so far, and some understanding of signing and how to handle it is needed. In this post, we'll take a look at why that happens and how large-scale teams successfully navigate signing.

What are certificates and provisioning profiles anyway?

While we don't need to dive into the full theory of software cryptography to successfully manage certificates and provisioning profiles, a condensed background might make future debugging sessions easier. After all, it's pretty likely that, as an iOS engineer, you've had to deal with mysterious code signing errors at some point in the past. Some knowledge of the fundamentals can give you a leg up in understanding and diagnosing issues!

an app on a specific device. When you first create a certificate, you'll also receive a **private key**. This key, which is unique to you, is used to encrypt a signature within your build artifacts, and as long as you don't somehow leak your private key to the external world, it serves as undeniable proof that it was *you* who verified and uploaded that build and not a bad actor pretending to be you. In an old school login/password setup, these credentials would usually be sent over the web, which would expose them to all kinds of man-in-the-middle attacks. With modern cryptography however, your private key never ever leaves your computer.

While certificates are designed to provide security around the communication between Apple and developers, **provisioning profiles** are designed to *provide security to the end-user*. To understand why provisioning profiles are needed, consider this challenge: *how can Apple give developers deeper access to the OS and the hardware without compromising the security of the average user by exposing them to cavalier (or nefarious) app makers?*

The answer? Only code that has been *authorized by Apple* to run on a specific device, can be executed. So how do you get authorization from Apple to distribute your

following questions:

- Who can sign a build of the app?
- What app can be signed?
- Where can the app be installed?
- Until when can the app be installed?
- How can the app make use of the capabilities of the OS and device?

When you create a provisioning profile on the Apple Developer portal (via the website, Xcode, or using the App Store Connect API), Apple *cryptographically signs* the profile.

When a user attempts to install an app on a device, the device first checks the provisioning profile's signature (the "authorization"). If the profile's signature is valid, it then checks the contents of the profile and verifies that the app being installed meets the profile's criteria.

Developers are required to provide provisioning profiles when targeting physical devices, and the profiles contain information such as:

- The allowed signing certificates for the app (the *who*)
- The allowed app and bundle identifiers of the app (the *what*)

[Product](#)[Integrations](#)[Automations](#)[Customers](#)[Pricing](#)[LOG](#)

- The entitlements (allowed capabilities) of the app (*e.g.*, push notifications) (the *how*)

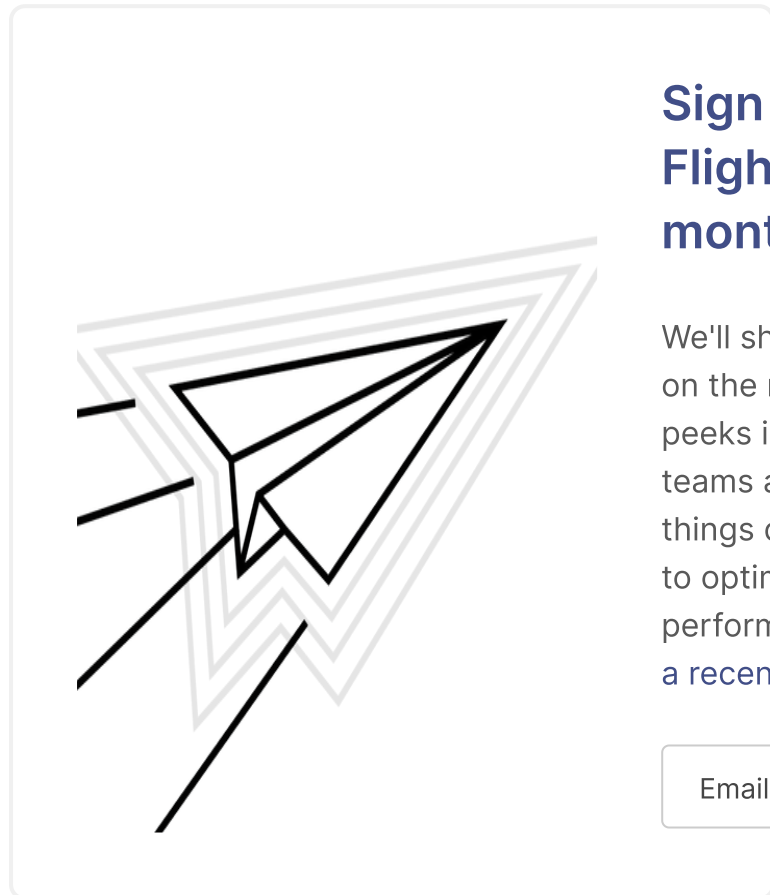
During the code signing process, this provisioning info is bundled into the binary and is then inspected by devices upon attempted installation. This allows devices to confirm that the binary is legitimate and that the current device is actually allowed to install it.

All builds targeting physical devices require provisioning profiles, but only some types of distribution methods require an explicit list of devices. For example, builds bundled with Enterprise or App Store distribution provisioning profiles do not need to include device data because such builds are designed to be installed by an undefined pool of users.

By forcing all apps to include valid provisioning profiles, Apple maintains a more secure app ecosystem which protects users against unwittingly installing untrusted apps on their devices. Unlike Android, which allows you to download binaries from the web and install them on your phone without restrictions, only apps that have been authorized by Apple (and meet the requirements of the provisioning profile they

[Product](#)[Integrations](#)[Automations](#)[Customers](#)[Pricing](#)[LOG](#)

behind how certificates work, [check this article out.](#))



What are certificates and provisioning profiles anyway?

How to handle certificates and provisioning profiles on large teams

Xcode Cloud and automatic code signing

Understanding code signing can help you scale your team successfully

How to handle certificates and provisioning profiles on large teams

Apple has put a lot of work into simplifying the management of profiles and signing certificates for developers by providing a simple 'automatic signing' option. As you

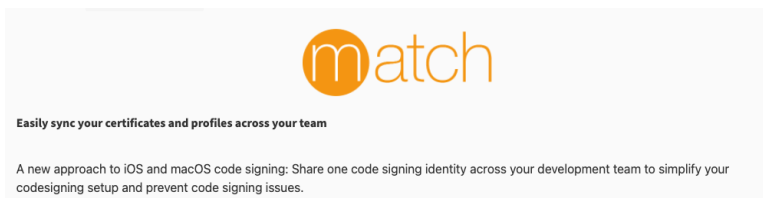
fine. So what's the problem?

The issue faced by large teams is not that automatic signing doesn't do what it claims to do, but rather that the way it does it isn't optimized for the team dynamics you'd experience in a larger project. Automatic signing is very destructive in nature: it revokes any previous certificates (those which are theoretically more likely to be outdated) in favor of creating new ones. So if you have a large project and multiple people are attempting to do build-related things on their machines at the same time, you'll likely start experiencing situations where one developer's local builds cause your own local builds to suddenly fail due to revoked certificates or invalid provisioning profiles. Your next build would then trigger Xcode to create a new certificate and/or profile for *you*, which would then result in *other* developers' builds failing...leading your entire team into an eternal loop of annoying signing issues.

For solo projects, it should be feasible to simply turn on the automatic signing feature and not worry about any of this signing stuff, but for large teams, the rule is always the same: *for the love of God, don't ever, **ever**, press the automatic signing button.*

source of truth for everyone who needs them. This might sound like a complicated setup, but most of the time it simply means that your team is storing the correct files in a repo somewhere, and then everyone always makes sure to update the files on their machine to match those in the repo before attempting to build anything locally.

Extremely large teams will likely develop an in-house solution for this, but there are open source tools available for teams that need a plug-and-play solution. One of them is [fastlane's *match* action](#):



In contrast to Xcode's "just start creating a bunch of certificates" approach to automatic signing, match makes sure the entire team is using the same single certificate. In the eventual case where you might need to update provisioning profiles to include new devices and/or capabilities, match will make sure these changes are replicated to everyone else working on the project. Behind the scenes, the match action works by simply dedicating a GitHub repo to store the

certificates to some external, third-party service completely negates the security benefit the private keys are meant to provide. As such, top-tier teams at scale will often roll their own internal solution for this problem. These internal solutions usually work similarly to match, with the main difference being that the files themselves are stored on-prem and gated behind the company's VPN/intranet, avoiding them ever being exposed to the internet. (However certificates are stored, there's always some risk factor in moving private keys out of a local environment – [codesigning.guide](#) does a good job of summarizing and quantifying the risks.)

Xcode Cloud and automatic code signing

While Apple appears to be putting effort into further automating the signing process with deeper integration of automatic signing in Xcode Cloud, the feature is still geared towards simple projects and will not work well for large teams. Unfortunately, I personally wouldn't expect Apple to evolve this further for teams at scale given their precedent of only supporting simpler use cases in many other Xcode features (DerivedData not being shared across multiple projects using the

Understanding code signing can help you scale your team successfully

Most iOS developers today recognize that signing certificates and provisioning profiles are an esoteric but important aspect of iOS development. As we've seen, they are critical for maintaining a secure development ecosystem, protecting both developers and end-users from bad actors. And while you don't need to understand the intricate workings of certificates and profiles to benefit from their usage, high-level knowledge of what they are and what purpose they serve can often be helpful both when dealing with day-to-day signing headaches and when planning and iterating on how you handle them as your team grows. Although Apple has done a good job of streamlining signing with their automatic signing feature, mature teams are often best served by manually managing certificates and profiles, either through open source tools like fastlane match, or by investing in a secure internal solution.


mobile app yet. struggling with a hairy one?

Try Runway Quickstart CI/CD to quickly autogenerate an end-to-end workflow for major CI/CD providers.

GET STARTED


RELATED POSTS

How to scale your iOS org: sharing tooling across multiple iOS projects




Jared Sorge
AUGUST 18, 2022

The Runway year in review



Richard Huffaker
DECEMBER 21, 2023

How to manage the mobile release process



Gabriel Savit
JANUARY 12, 2024

SIGN UP FOR THE FLIGHT DECK,
OUR MONTHLY NEWSLETTER

Email

SIGN UP



Product	Integrations	Automations	Customers	Pricing	LOG
Mobile insights		Version control	App review times		
Build Distro		CI/CD	App Store Connect status		
Automations		App stores	page		
Security		<ul style="list-style-type: none">• App Store Connect	App hotfix leaderboard		
Pricing		<ul style="list-style-type: none">• Google Play Console			
What's new		Slack	Company		
Explore sandbox		Monitoring	About us		
		All integrations	Contact		
Use cases			Terms of service		
fastlane			Privacy policy		
Release trains			Careers		
Mobile DevOps			Status		
Cross-platform					
<ul style="list-style-type: none">• React Native					
<ul style="list-style-type: none">• Flutter					