Migrating Code Signing Configurations to Xcode 8

This is a guide to code signing software for iOS development and deployment. The information contained here may be useful to better understand how the process of code signing works and is implemented, but is specifically for people that build applications and software that must be signed by multiple teams (e.g.: Development and Enterprise) and how to migrate your existing signing configurations to work in Xcode 8.

Table of Contents

Introduction to Code Signing

To understand the role that code signing plays in the overall ecosystem, you must first understand how it works. This section is a guide to how signing certificates are created and the role that code signatures play in the iOS ecosystem. To do this, we will walk through each step in the process of signing an application.

The act of signing something means to have it be validated by a known trusted authority. This is very important to the ecosystem that Apple has created for the iOS platform. All software that gets run on an iOS device must be signed by a source trusted by Apple. To enforce this, Apple has all developers of iOS software gain authorization to install and run any code onto an iOS device apply to do so through the Apple Developer Program. Once you are a member of this program, you will have to make a request for a signing certificate so you can sign software to be installed and run on an iOS device.

Certificate Signing Request

To get a signing certificate you must first start by creating a CSR (Certificate Signing Request). To create a CSR, you will first have to generate a new private key to use for signing the request. To do this, you can run the following command:

```
$ openssl genrsa -out MyPrivateSigningKey.key 2048
```

This command will generate a new private key and write it to disk at the path given with the `-out` parameter. The parameter `2048` tells `openssl` to use a 2048 bit long modulus for generating the RSA key pair. This is the standard for creating signing certificates on OS X.

Once the private key has been created, it will be used to sign the CSR. To generate a new CSR, you will need to use the following command:

```
$ openssl req -new \
        -key MyPrivateSigningKey.key \
        -out MyCertificateSigningRequest.csr \
        -subj "/emailAddress=[Your email address]/commonName=[Your Name
```

There are three fields that should be specified as part of this request:

- emailAddress

- commonName
- countryName

The contents of these fields will be used to add metadata to your signing request so that the certificate you are requesting is identifiable as “yours”. So, if I was going to create a new CSR, the command would perform would look like this:

```
$ openssl req -new \
        -key MyPrivateSigningKey.key \
        -out MyCSR.csr \
        -subj "/emailAddress=hello@pewpewthespells.com/commonName=Saman
```

This will create a CSR that contains the information I passed as the subject of the request and my public key, then that data will be signed by my private key to validate it originally came from me. With the Certificate Signing Request created, I can now go to the Apple Developer Portal and submit it to request a Development signing certificate.

Signing Certificate

Once the CSR has been submitted to the Certificate Authority, an identity/digital ceritificate is created using the information in the CSR. This certificate that is generated is signed by the Certificate Authority (Apple) and issued to a developer. This allows the developer to use this certificate to sign software that they want to install onto an iOS device without having to have the iOS device explicitly know about each individual developer. This is because the iOS device will know about (and trust) the Certificate Authority that the developer’s certificate was generated by. Thus, the iOS device knows it can trust a certificate that was signed by Apple.

To Recap:

```
[1. Private Key] -----> [2. CSR] -----> [3. Apple Developer Certificate
                                                                |
                                                                |
                                                                |
                                                                V
        [5. Signed Application] <----------- [4. Developer Identity Cer
                    |
                    |
                    |
                    V
            [6. iOS Device] ----------------------------------------
```

1. The developer’s private key is used to sign the CSR

2. The CSR contains the developer's information and public key
3. Apple's Certificate Authority recieves the CSR and will generate and sign an identity certificate for the developer
4. The developer recieves the identity certificate and uses it to sign applications using this certificate and their private key
5. Software that is signed by the identity certificate is said to be trusted by the signer.
6. Software installed on iOS devices have their certificates validated against the CA cert that signs developer's certificates. This validates the chain of:
    - Software being trusted by the developer
    - The developer is trusted by Apple
    - The device trusts Apple

    Therefore, the device trusts software from the developer.

Signing a Binary

Part of the requirement of building iOS software is that you must sign all of the software that you are going to be deploying to an iOS device. This is due to the strict security policies that are enforced by the operating system. Xcode helpfully integrates this step into building applications for us by invoking `codesign` on the executable binary. This will generate a signature of the contents of the executable code in the binary using the private key associated with the identity certificate that was created by Apple. The generated signature will then be embedded into the executable binary to allow for it to be validated. This will ensure that the code that the application has cannot be modified without causing a validation error against the embedded signature. Since iOS applications are comprised of more than the executable binary, this single embedded signature is not enough to ensure integrity of the entire application.

On OS X, applications are seen as files with the `.app` extension. This is called a "bundle". Bundles are directories that contain a structured format and layout of contents. By giving directories file extensions, it allows them to be registered with the system to be loaded by another program. In the case of `.app` bundles, these get loaded by LaunchServices, which is a system responsible for running software initiated by the user. LaunchServices is responsible for running software that gets started by other software. So, typically when you launch Xcode, you are either double clicking on it in Finder, or clicking on it in the Dock, or asking Spotlight to launch it. All of these are separate pieces of software that are running on your system that ask for a new application to be launched based on the `.app` bundle you are trying to run. On OS X, application bundles follow this structure:

```
+-o Foo.app                 | user visible application
  +-o Contents              | directory that holds the contents of the
    +-o Info.plist          | file that has information about the bundl
    +-o PkgInfo             | metadata file produced when creating the
    +-o MacOS               | directory containing the executable binar
    | +-o Foo               | executable binary
    +-o Resources           | directory of resources availale to the ap
    | +-o Foo.icns          | icon that is shown for the user visible a
    +-o Frameworks          | any frameworks or dynamic libraries used
      +-o Sparkle.framework | example of a common framework in OS X sof
```

This is what is called a “deep application bundle”, as there are additional directories between the top level `Foo.app` and where everything is stored.

On iOS, applications follow a slightly different structure as they are “shallow application bundles”

```
+-o Foo.app                     | user visible application
  +-o Info.plist                | file that has information about the b
  +-o Foo                       | executable binary
  +-o Foo.icns                  | icon that is shown for the user visib
  +-o Frameworks                | any frameworks or dynamic libraries u
    +-o AFNetworking.framework  | example of a common framework in iOS
```

As you can see, instead of having separate directories like the application bundles on OS X does, iOS uses a more shallow approach to storing the necessary assets and data used by an application.

So to ensure that not only the executable binary remains unmodified, but also the data and assets that are used while it is being run, a new directory is added to the application bundle. This new directory is named `_CodeSignature`, and contains a `CodeResources` file. This file is a plist that lists all of the files that are included as part of the bundle and gives each file a hash. This hash is used to validate the contents of the bundle remain unchanged. Some resources can be configured to be updated and omitted as part of the resource check, to prevent an application from invalidating its own signature.

This process allows developers to perform a build and sign it immediately and know that the binary that they plan to distribute is the same as the one that was originally built.

Provisioning Profiles

While the certificates are used to validate the authenticity of the signer of an application, there is an additional component that is used to validate that the application is allowed to be installed on a specific device. This component is

called the provisioning profile. A provisioning profile is a plist file that is cryptographically signed by Apple's CA to ensure it cannot be modified after being created. This allows Apple to have complete control over the deployment mechanism that is used on iOS.

A provisioning profile contains some specific information for enabling deployment of an application:

- certificates that can be used to sign the application
- bundle identifier to be matched against the application
- method of deployment (Enterprise-style or based on device UDID)
- team identifier
- sandbox entitlements
- expiration date of the installable

All of these attributes are used in determining of the device is allowed to install the application that the provisioning profile was bundled in. This restricts who can install a development-signed application and allows for the source of any signed application to be traced back to the account that the certificate was created from on the Apple Developer Portal.

The purpose of the provisioning profile is to allow for specific configuration of an installable to be secure, while also not making it prohibitive to update that configuration at any time. Within the Apple Developer Portal, you are allowed to edit and regenerate provisioning profiles at any time. Doing this doesn't invalidate the previous profile, it only generates a new one with different contents based on what you modified.

Signature Validation

As mentioned earlier, the iOS operating system strictly enforces that all executable code is signed against one of Apple's Certificate Authorities. This doesn't only apply to running code, the validation process starts before an application even installed.

When developing software for iOS, you will build and sign the application on your computer, then Xcode (or some other process if you use a different IDE) will talk to an iOS device that is connected via USB. It will perform a authentication handshake with the iOS device, then start communicating with it over a local secured socket connection via the USB interface. At this point the iOS device is informed that we want to install a software package, and begin sending the data over to the device. The iOS device will receive this data and reconstruct the application in a temporary sandboxed environment. Once the application has been completely recieved, the operating system will validate that the application is signed and is capable of being installed onto this device.

Once the application is determined to be from a trusted source, it is validated against the provisioning profile to ensure that it can be installed and run on this device. Each iOS device has a unique identifier, the UDID (Unique Device IDentifier). This identifier is composed by taking the SHA1 of a string composed of the following components:

- serial number
- ECID (aka UniqueChipID, this is unique to every device)
- MAC address of the wifi card
- MAC address of the bluetooth card

This identifier contains enough information to uniquely identify all iOS devices. The identifier is required to be used by developers when registering specific devices with their account on the Apple Developer Portal. Each account is restricted to a specific number of devices that are allowed to be registered. A provisioning profile can include any number of identifiers of the devices registered to this pool. This restricts developers to only being able to install devices that are register to their pool of devices. If the device's UDID doesn't match any of the identifiers that are registered with the provisioning profile, then the app will not be installed as it is not approved. If the application's signature and provisioning profile both pass the validation step, then a true sandbox container is created for that application to reside in. A new directory is created on the iOS system where the application bundle will be installed to. From there it will have access to a limited scope of the file system and resources.

Additionally, whenever and application is launched on iOS, the system performs additional validation and code signature verification steps to ensure that since the installation was performed, nothing has modified the executable's code. The process in charge of this is called `amfid` which stands for "Apple Mobile File Integrity Daemon". This is a service that runs in the background and ensures that everything that is being run on the system is allowed to run on the system. Recently there was an issue opened with Apple about the performance of this process in regards to launching applications that contain many dynamic libraries. One of the requirements of running any code on iOS is that all executable code must be verified as trusted before it can be loaded into memory by the dynamic linker. This process was causing a bottle-neck on the system which resulted in the behavior described in this issue, and led to Apple recommending that developers limit the number of dylibs that an app uses to a handful.

Types of Deployments

There are two ways of deploying an application to an iOS device, and this is based on the signing configuration.

1. Development – Deploying an application to your devices
2. Distribution – Deploying an application to other people's devices

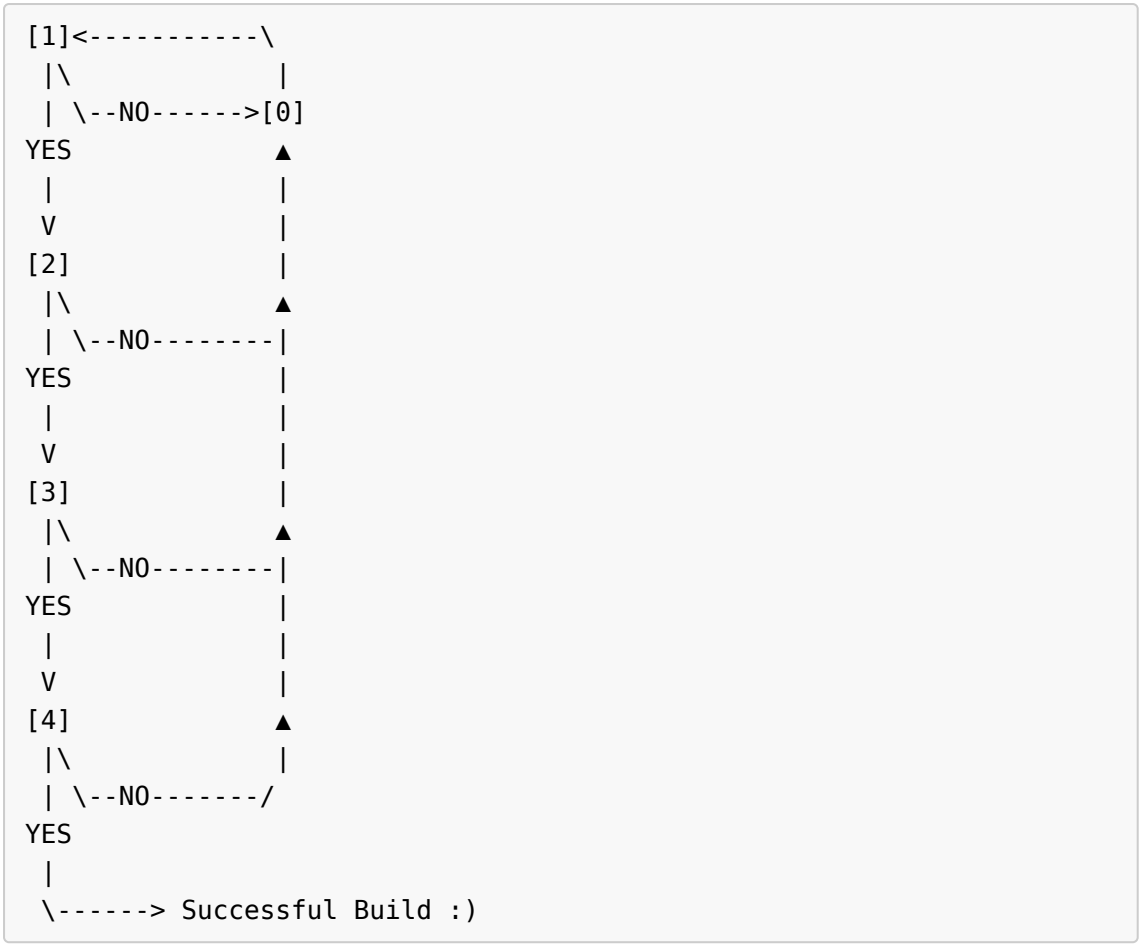
These two configurations are significant because they depend on the signing identity that gets used to performing the signing of the application.

As a software developer, to deploy an application to one of your iOS devices, you must have a private key that is paired with an identity certificate that is signed by Apple's Developer Certificate Authority. An identity certificate that is signed by that CA will have permission to install on a limited number of specific devices, and then attach a debugger to that specific application to do development with it. This is a relatively elevated level of permissions on the system, and for this reason Apple enforces the limit of the number of devices registered to a single developer account.

The second kind of deployment is for the purpose of software distribution. Once an application has been fully developed and ready for use, it needs to be signed for distribution. This enables a level of permissions that is more locked down than the development style of deployment, and more in-line with apps that are acquired from the App Store or from Enterprise vendors. Applications that are submitted to the App Store must be signed with a distribution identity certificate first. This requires an additional CSR be made and submitted to Apple. However, this time it gets signed by a different CA, the Apple Distribution Certificate Authority. Additionally, applications that are built for internal use are likewise signed with an Enterprise Distribution identity certificate.

Understanding "Fix Issue"

A new dialogue was introduced to Xcode that assists developers by resolving problems with their code-signing configurations. This was to help developers avoid the terrible process of working out what they were missing to get their application deployed. This was a shortcut of completing the following steps:



1. Determine what identity certificates are available
2. Check that the identity certificates had the corresponding private keys
3. Determine what provisioning profiles were available
4. Check that there is a provisioning profile that can be used to install on the target device with the signing identity that was found and bundle identifier of the app

If the answer to any of these was “NO” then you would end up going to step 0 in the flow-chart, which represents the Apple Developer Portal. Here the “Fix Issue” dialogue would go and create a new CSR and request a new certificate if there isn’t one already, and automatically create a new provisioning profile that would be able to be used to deploy the application. This is extremely helpful behavior, the downside of it is that it is approaching a complicated problem with the outlook of a hammer. Truth be told, it is an approach that is is suitable for developers that don’t have a complete understanding of how the code signing system works and wouldn’t be able to resolve for themselves. Also, this feature of Xcode plays a very important role in the setup of automatic code signing configurations.

> Note: When working with frequent changes to the provisioning profile (eg: changing entitlements), it is best to do all of the debugging work with a

> development profile, as Xcode will happily regenterate those when necesssary. Once the configuration has been finalized in the development environment, you should be manually updating and regenerating the distribution provisioning profile to accomodate any new changes.

↑ Table of Contents

> Now we are getting to the part about how to configure Xcode to sign applications based on build configurations and how that has worked previously and going forward. For this part I am going to be using **xcodebuild** and xcconfig files for most of my examples, as that is what I am working with. This is primarily targetted towards engineers that are working on an app that gets built for both Enterprise and App Store distribution, and how to modify your existing approach to work in Xcode 8. This also contains a number of tips about how to generally configure the signing information for performing a build across various versions of Xcode.
>
> I am going to start off by outlining some basic expectations around handling building software for iOS and the management of signing identities:
>
> 1. No developer should have access to the private key and signing certificate of the distribution signing identity.
> 2. Builds of the application for distribution (either App Store or Enterprise) are performed on a CI server and made available to those that need it.
> 3. Build settings should be specified in xcconfig files. This is done to prevent destructive edits or over-rides on the part of Xcode.
> 4. The CI server has access to:
>   - Provisioning Profiles - We self-host our own Jenkins instances, which makes managing this aspect of the build extremely easy as we have direct access to the file system to add any new provisioning profiles. If you are using another system, you will have to either include the profiles in the repo or use some other means to ensure they are available to perform the build.
>   - Developer Account - For Xcode to perform an Archive and Export actions, it may need access to a development account so that it gets resigned correctly for uploading.
>
> These points are the premise around the remainer of this guide; to perform builds in a uniform manner that limits access to the signing credentials of the development account.

Signing In Xcode 7 and Prior

In Xcode 7 and prior, there was an established method of signing applications that was governed by specifying values for the `CODE_SIGN_IDENTITY` and `PROVISIONING_PROFILE` build settings. These were used to determine the signing configuration that should be used for creating the embedded signature in an application.

Signing Methods

While both the `CODE_SIGN_IDENTITY` and `PROVISIONING_PROFILE` build settings required values to be able to build an iOS application, there were two approaches that could be taken with this; automatic signing and manual signing configuration. Both follow the same core logic path but can be leaveraged for different purposes in practice. The behavior behind code signing in Xcode has followed this set of rules:

1. Determine the bundle identifier of the compiled application
2. Determine the UDID of the device that the application is being deployed on
3. Determine the provisioning profile that should be used based on the value of `PROVISIONING_PROFILE`
4. Determine the signing identity that should be used based on the supported certificates in the provisioning profile and the value of `CODE_SIGN_IDENTITY`

Steps 1 and 2 of this process will not fail, as they are core to building and deploying an application to an iOS device. Steps 3 and 4 can fail, and this is where the “Fix Issue” dialogue will appear. Step 3 will fail if the developer has a signing identity, but lacks a provisioning profile that supports the bundle identifier and target iOS device. Xcode will go ahead and create a new provisioning profile that supports the existing signing identity and the target device and download that for use. Step 4 will fail if the developer has no valid signing identity. At that point Xcode will perform all of the steps to create a new signing identity for the developer and download that along with a new provisioning profile. This is to save developers the hassle around determining which of the two potential problems are the real root cause.

Automatic Signing

In Xcode 7 and prior there is an approach of “automatic signing” which utilizes the ability of the “Fix Issue” dialogue to work around issues of deploying an app to a device for development purposes. To employ this behavior in your own signing configurations, you should specify the following values:

```
CODE_SIGN_IDENTITY = iPhone Developer
PROVISIONING_PROFILE =
```

To be clear, what we are doing here is saying that we want Xcode to sign the application using a signing identity that contains the phrase *iPhone Developer* and to not use a specific provisioning profile. This means that as long as each individual developer on a team has a valid development signing identity (identity certificate and private key), they can safely deploy to any device without fear of breaking the signing configuration. As mentioned, this will cause Xcode to prompt to “Fix Issue” whenever attempting to deploy to a device that the individual developer doesn’t have a provisioning profile for. This is ok as Xcode will download that profile and you will be able to deploy to the target iOS device. This may cause Xcode to dirty the .xcodeproj file by adding in values to the build settings. Once the newly created provisioning profile has been downloaded by Xcode, that value can be safely removed as it will be found automatically by Xcode when the individual developer performs a build.

This behavior is extremely desirable for build configurations that only members of the app development team would use. This sets no requirements on siging identity beyond being a member of the Apple Developer Program. This is ideal for working on teams of any size, because it allows all developers to manager and maintain their own signing identities and provisioning profiles for their devices, without forcing anyone to maintain a profile that contains everyone’s certificates and devices.

Manual Signing

While automatic signing is ideal for developing an app, it doesn’t work as well when dealing with specific requirements of development or distribution. This is when a manual signing configuration must be used. Typically the requirement is around using a specific provisioning profile for a build. In this case, you will have to configure the build settings as such:

```
CODE_SIGN_IDENTITY = iPhone Distribution
PROVISIONING_PROFILE = 2249294d-440a-427c-bbef-432326c6552b
```

This will tell the build system that to deploy this application with the provisioning profile that has an identifier of `2249294d-440a-427c-bbef-432326c6552b` and is signed by a *iPhone Distribution* identity certificate. This forces Xcode to use these settings or return an error when building if the requirements could not be met. This is the ideal configuration for performing App Store, Enterprise, or any sort of Ad-hoc

distribution to prevent your entire team having access to the private key or accidentally over-riding that with their own distribution signing certificate.

Building for Development

When building software for development, you want a process that will not cause issues that halts the process of writing and debugging code for any length of time. As mentioned previously, the best configuration to use for this type of build is the “automatic” signing method. This reduces the strain of maintenance and lowers the potential of breaking changes being applied to the Xcode project file.

To configure this, you should start by creating a new build configuration for your development build. You can reuse the existing “Debug” configuration that Xcode automatically creates for this. Additionally, you should create a new xcconfig file to house the settings to want to make specific to this build configuration. Once you have done both of those things, in the Xcode project editor, you will want to assign a the xcconfig file to the application’s target for the “Debug” configuration.

Now, go to the xcconfig file and make sure it has the following items in it:

```
CODE_SIGN_IDENTITY = iPhone Developer
PROVISIONING_PROFILE =
```

This will tell Xcode to resolve the provisioning profile for us. It is important to note that the empty-value that is given to `PROVISIONING_PROFILE` will appear as the value *Automatic* in the Xcode build settings editor. The same can be applied to the `CODE_SIGN_IDENTITY` value, where, instead of *iPhone Developer*, you would see the value *Automatic* as well. This will tell Xcode to use any type of signing identity that satisfies the requirement of the provisioning profile that was found that matches the UDID and bundle identifier of the application. You should note that the *Automatic* value for `CODE_SIGN_IDENTITY` is distinctly different from the *None* option. To understand how this works, the value that the build setting is give is used as a filter against all the known results. So the value of `CODE_SIGN_IDENTITY` can be represented by the following command:

```
$ security find-identity -p codesigning -v | grep "$CODE_SIGN_IDENTITY"
```

So when the value of `CODE_SIGN_IDENTITY` is `iPhone Developer` you will only be using identities that match that, whereas when using an empty string, you will see all the results.

After that is setup, you will want to go into the Scheme Editor in Xcode and select the “Debug” build configuration for the actions associated with the various types of builds you can perform. These options give you control over what settings are passed to the build system when those scheme actions (Run, Test, Analyze, Profile, and Archive) are invoked. This allows you to create signing configurations that are directly tied to which scheme is built.

Building for Distribution

Configuring apps to be built for Distribution works almost identically to the configuration steps required to get apps to be built for Development. A new build configuration should be set-up as well as a scheme to be able to perform builds with that configuration. The difference here is that instead of using the *Automatic* value for the `PROVISIONING_PROFILE` build setting; the identifier of a specific provisioning profile is used.

This alters the behavior of the signing process slightly, as now the bundle identifier that is used for the app target must correspond with the identifier that is named in the specified provisioning profile. As a result, the Xcode build system will be responsible for locating a signing identity on disk that matches one of the identities that is listed within the provisioning profile. If the certificate or private key are not found, then an error is raised to the developer to notify them that the app target could not be signed to be deployed to the target iOS device.

These changes are desired because when building and deploying an app for distribution, the specific distribution provisioning profile and corresponding signing identity should be used. Builds created for distribution should ideally be performed on a computer that nobody has access to modifying, such as a continuous integration server. This restricts access to the signing identities and allows all builds to be performed in a uniform fashion.

Follow the same steps as you performed previously for setting up the scheme; by configuring the scheme to use the same build configuration for each type of action. This allows all types of builds of the scheme to produce a binary that is signed and configured the same way.

↑ Table of Contents

Signing in Xcode 8

With the introduction of Xcode 8, new methods of managing the signing configuration of a target was introduced. These new methods conflict with the

established behaviors that existed in Xcode 7 and prior. If your signing configurations followed the patterns described in the previous section, then you may already be familiar with the way that this does not work in Xcode 8.

Additionally, there are two new build settings that are introduced to help manage the new signing methods: `DEVELOPMENT_TEAM` and `PROVISIONING_PROFILE_SPECIFIER`.

- `DEVELOPMENT_TEAM` setting is used to provide greater control over the signing identity used in signing a binary (especially for people that are part of multiple teams).
- `PROVISIONING_PROFILE_SPECIFIER` setting is used to indicate the type of signing method that should be used for a given target. Targets that want to employ the *manual* method of code signing will not use this setting, and will instead use the deprecated `PROVISIONING_PROFILE` build setting. If the setting is set then, the new automatic code signing method will take over.

Signing Methods

Xcode 8 introduces a new method called “Automatic Signing”. This is a replacement to the existing semi-automatic method of managing signing identities. This change may cause some initial hiccups due to some changes in how the “manual” and “automatic” signing methods have been separate. I **strongly** recommend that everyone migrates over to the new Automatic Signing method as soon as possible as it results in fewer potential disasters in the management of the signing identities and information.

Automatic Signing

This is 100% automatic and takes cues off the values of the `DEVELOPMENT_TEAM` and `CODE_SIGN_IDENTITY` build settings. There are some key differences to this approach to what you may have previously understood as an “Automatic” signing configuration.

- The `DEVELOPMENT_TEAM` setting must be set to a valid team identifier. This cannot be empty, which would intuitively resolve to use signing identities of any team that are on disk. This is important to understand for situations where all the members of your development team are not part of the same “Development Team” as defined by the Apple Developer Account. This situation often comes up when working with contractors or small companies where the developers may use their own developer accounts instead of a company account for day-to-day development. To avoid causing repeated edits to the Xcode project file or the xcconfig files, the company should be adding all the developers to their account to provide access to the same

development team. (NOTE: This doesn't require that all developers have access to the private keys used for signing non-development builds.)
- The specified `CODE_SIGN_IDENTITY` that is used should be a generic entry; such as "iPhone Developer" (without a name specifier to prevent conflicts with other team members).
- Neither `PROVISIONING_PROFILE` nor `PROVISIONING_PROFILE_SPECIFIER` need to be set for the application target to be built and deployed to an iOS device.

When Automatic Signing is enabled, there are a couple of metadata values that get written into the xcodeproj files that specify what development team should be displayed as in-use by each target. Unintuitively, these values are not set as part of the build settings that for each target, but instead part of the TargetAttributes dictionary defined on the root PBXProject object in the pbproj file within the xcodeproj bundle. In addition to the team identifier, another key-value pair is set in the TargetAttributes; this key-value pair indicates to Xcode (in versions that support it) that the UI of the Xcode Target Editor should reflect that a given target should be using Automatic Signing. These values are automatically updated by Xcode when necessary, in my experience, this shouldn't cause churn in the xcodeproj file as long developers are not changing these settings.

Manual Signing

The new manual approach to managing the signing configurations is truly 100% manual. When using this method, you will have to specify the values for the `CODE_SIGN_IDENTITY` and `PROVISIONING_PROFILE` build settings. While this sounds similar to the existing method of signing in Xcode 7, there are some important differences here.

- Provisioning profiles can now be specified by name instead of UUID that they are given upon creation. This makes it easier to track which profile is used for what and allows for new profiles with the same name to be created and used without having to update any existing configurations or project files.
- Provisioning profiles that were created by Xcode automatically by the "Fix Issue" feature cannot be used in a manual signing configuration. (If you were hoping to get away with telling Xcode to match against the name pattern that Xcode uses to generate new profiles you are out of luck).
- The "Automatic" setting for the `PROVISIONING_PROFILE` build setting no longer works. To have Xcode automatically resovle which provisioning profile to use for deployment you will have to use the Automatic Signing method.

To elaborate more on a potential issue/friction point and why you should migrate over to using the Automatic Signing approach: manual configuration means everything is manual; this include the provisioning profile creation. Since the automtically created profiles from Xcode do not work in this mode, the someone with access to an Admin or Agent level of the Apple Developer Account will have to log in and add the UDIDs of the devices to the Developer Portal. After doing that, a new provisioning profile must be created that grants the ability to run the application on all of the devices and include the signing identities of all the team memebrs that were added to the Developer Portal for the account. This provisioning profile and list of devices will need to be constantly mantained to be updated when new team members and new development/testing devices are added.

Maintaining such a profile is going to be a full-time job for someone and depending on the size of your team will quickly exhaust the ability to add new devices. Simply put, outside of a handful of exceptional cases there is no reason to continue to use the manual approach to signing if it can be avoided as it will be a non-trivial burden to any developer. This model of management fundamentally doesn't work at any level of development and should only be used to build existing legacy configurations that are already configured as such.

Building for Development

> This section is going to be talking about the way to configure the Xcode project in the case of using the new Automatic Signing method. This is what I am using for all the projects I maintain and has been recommended to me as the method that should be used by all developers going forward.
>
> Additionally, if you are using the Manual Signing method, then your builds are already configured and working without issue. If you want to migrate to the new Automatic Signing method then you should read the rest of this blog post.

As with the previous methods of configurating code signing, setting up building for development is going to start with a build configuration and a scheme. To ensure that development builds are created, you should be configuring the scheme actions (Run, Test, Analyze, Profile, and Archive) to all point at the same build configuration you plan to use for development. In addition to this, you will want to mark the `CODE_SIGN_IDENTITY` build setting for **all** of the build configurions to be set to "iPhone Developer". This may seem unintuitive, but is the correct approach that should be taken to ensure nobody will accidentally

modify the intended settings for the scheme. This has a couple of benefits over the previous approach to assigning a signing identity per build configuration:

- All developers can build the application in all of the modes that would be produced, but cannot distribute it. This is extremely useful when attempting to debug a crash or failure-case in the code that is triggered outside of a normal development build.
- Much harder to override or break the desired signing identities that are configured.

In addition to that change, you will want to set the `DEVELOPMENT_TEAM` setting to correspond with the identifier that your team has. If you don't know what this is, you can look it up by logging into the Developer Portal using the Apple ID for your Apple Developer Account and looking under the "Membership" section.

Once you have this information, setting up the automatic configuration requires checking the box to enable it in the "Signing" section of the "General" tab of the Xcode Target Editor.

Building for Distribution

In order to create builds of your application for distribution while using the new Automatic Signing method in Xcode 8, you have to utilize the "Archive" scheme action. This is assigned a build configuration in the scheme editor, and while that build configuration is setup to use the `CODE_SIGN_IDENTITY` value of "iPhone Developer"; the "Archive" action will over-ride that to always use a distribution signing identity and provisioning profile. So instead of performing a regular "Build" action, you will have to invoke the "Archive" action. This will generate a build and bring up the Xcode Organizer window that lists all of the archives created by Xcode as an ".xcarchive" in the UI. To do this same process on a headless CI computer, you will have to invoke two **xcodebuild** commands:

```
$ xcodebuild archive \
             -workspace MyApp.xcworkspace \
             -scheme MyApp-Enterprise \
             -configuration Enterprise \
             -derivedDataPath ./build \
             -archivePath ./build/Products/MyApp.xcarchive

$ xcodebuild -exportArchive \
             -archivePath ./build/Products/MyApp.xcarchive \
             -exportOptionsPlist ./export/exportOptions-Enterprise.plis
             -exportPath ./build/Products/IPA
```

These commands will do the following:

1. Create the xcarchive file
    1. Specify the “Archive” action to be performing.
    2. Specify the Xcode file to be working in, this is necessary for the scheme to be resolved correctly. If the `-workspace` flag is not specified, then **xcodebuild** will default to trying to use a xcodeproj file to resolve the scheme, even if the scheme itself is stored as part of the xcworkspace file. This can result in build failures from not building the implicit dependencies defined by the scheme.
    3. Specify the build configuration to use. This is necessary to do so that the scheme gets built with the correct configuration that is assigned to it. This is due to behavior in the Xcode build system that says when invoking **xcodebuild** from the command line that a default (Release) configuration should be used if none is specified.
    4. Specify the build location, for our builds we want to place them in a build directory that is always going to be the same so that the CI server knows where to look for the built products in.
    5. Specify the file path to create the xcarchive bundle.

2. Create the ipa file
    1. Flag to tell **xcodebuild** that we are not building a target, but exporting an existing xcarchive.
    2. Specify the path to the xcarchive bundle. This is the xcarchive that was just produced by the previous invocation of **xcodebuild**.
    3. Specify the values that should be used when producing the ipa file. These values are stored in a plist file on disk that can have all the information filled in already. An example of this is listed below along with documentation of what the key-value pairs for the plist are and what they mean.
    4. Specify the directory to export the ipa file to. This produces an installable that can be used for distribution.

> Note: This process is exactly the same for creating distribution builds for Enterprise or the App Store. The “method” key in the exportOptions.plist will have the value “app-store” instead of “enterprise. Additionally, the”teamID" key will have a different value for your Enterprise team identifier versus your Development team identifier.

Example `exportOptions.plist`:

```
<?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>compileBitcode</key>
    <false/>
    <key>method</key>
    <string>enterprise</string>
    <key>teamID</key>
    <string>G00DC0NF1G</string>
    <key>uploadBitcode</key>
    <true/>
    <key>uploadSymbols</key>
    <true/>
    <key>manifest</key>
    <dict>
        <key>appURL</key>
        <string>foo.bar/app</string>
        <key>displayImageURL</key>
        <string>foo.bar/display-image</string>
        <ke>fullSizeImageURL</key>
        <string>foo.bar/full-sized-image</string>
    </dict>
</dict>
</plist>
```

Documentation for `-exportOptionsPlist`:

```
compileBitcode : Bool

    For non-App Store exports, should Xcode re-compile the app from bit

embedOnDemandResourcesAssetPacksInBundle : Bool

    For non-App Store exports, if the app uses On Demand Resources and
    are embedded in the app bundle so that the app can be tested withou
    packs. Defaults to YES unless "onDemandResourcesAssetPacksBaseURL"

iCloudContainerEnvironment

    For non-App Store exports, if the app is using CloudKit, this confi
    "com.apple.developer.icloud-container-environment" entitlement.
    Available options:
        * Development
        * Production
    Defaults to Development.

manifest : Dictionary

    For non-App Store exports, users can download your app over the web
    distribution manifest file in a web browser. To generate a distribu
    value of this key should be a dictionary with three sub-keys:
        * appURL
        * displayImageURL
        * fullSizeImageURL
    The additional sub-key "assetPackManifestURL" is required when usin

method : String

    Describes how Xcode should export the archive. Available options:
        * app-store
        * ad-hoc
        * package
        * enterprise
        * development
        * developer-id
    The list of options varies based on the type of archive. Defaults t

onDemandResourcesAssetPacksBaseURL : String

    For non-App Store exports, if the app uses "On Demand Resources" an
    "embedOnDemandResourcesAssetPacksInBundle" isn't YES, this should b
    URL specifying where asset packs are going to be hosted. This confi
    app to download asset packs from the specified URL.

teamID : String

    The Developer Portal team to use for this export. Defaults to the t
```

```
thinning : String

    For non-App Store exports, should Xcode thin the package for one or
    Available options:
        * none (Xcode produces a non-thinned universal app)
        * thin-for-all-variants (Xcode produces a universal app and all
        * (a model identifier for a specific device (e.g. "iPhone7,1"))
    Defaults to <none>.

uploadBitcode : Bool

    For App Store exports, should the package include bitcode? Defaults

uploadSymbols : Bool

    For App Store exports, should the package include symbols? Defaults
```

↑ Table of Contents

Working in Both Worlds

Getting these two different systems to work together is possible so that developers and the CI servers can update to Xcode 8 when the time is right without holding back anyone else. For this, we are going to use an xcconfig file to setup the configuration for doing this as it is non-trivial and unnecessarily confusing to configure in the Xcode Target Editor interface.

First, we want to define what are the variables at play that can change the signing configurations:

- Build Configuration, this is the `CONFIGURATION` build setting.
- Target type, this will be determined by the `WRAPPER_EXTENSION` build setting. (application targets will have this set to `app`)
- Xcode version, this can be determined by a couple undocumented build settings that are set by Xcode. We are going to use `XCODE_VERSION_MAJOR`.

From this we are going to create two variables that will be able to interpret the different values that these three settings will have:

```
CONFIGURATION_AND_VERSION = $(CONFIGURATION)_$(XCODE_VERSION_MAJOR)
WRAPPER_EXTENSION_AND_CONFIGURATION_AND_VERSION = $(WRAPPER_EXTENSION)_
```

With these two variables defined in an xcconfig file, we can use the variable substitution behavior in xcconfig files to allow for conditional assignment of the

other build settings we need to modify to alter the signing configuration that Xcode will use in a build:

- `CODE_SIGN_IDENTITY`
- `PROVISIONING_PROFILE`
- `DEVELOPMENT_TEAM`

CODE_SIGN_IDENTITY

The first variable that will take on different properties is the `CODE_SIGN_IDENTITY` build setting. In the style of configuration that was described in the “Xcode 7 and Prior” section of this post, the value of this build setting varied based on the build configuration used. This behavior changes in Xcode 8, so we need to utilize a variable that will only differ between the major versions of Xcode, `XCODE_VERSION_MAJOR`.

```
CODE_SIGN_IDENTITY = $(CODE_SIGN_IDENTITY_$(CONFIGURATION_AND_VERSION))
```

This assignment line says that at build-time, the valuye that gets assigned to `CODE_SIGN_IDENTITY` will be based on another variable that is constructed based on the values of the build configuration and the Xcode version. By doing this, you can customize each variation of the assignment:

```
CODE_SIGN_IDENTITY_Debug_0700 = iPhone Developer
CODE_SIGN_IDENTITY_Enterprise_0700 = iPhone Distribution
CODE_SIGN_IDENTITY_Production_0700 = iPhone Distribution

CODE_SIGN_IDENTITY_Debug_0800 = iPhone Developer
CODE_SIGN_IDENTITY_Enterprise_0800 = iPhone Developer
CODE_SIGN_IDENTITY_Production_0800 = iPhone Developer
```

In this scenario, we are configuring the variables such that the behavior designed for building in Xcode 7 remains the same, while updating the values to correspond with the desired state in Xcode 8. To show this in action:

```
CODE_SIGN_IDENTITY = $(CODE_SIGN_IDENTITY_$(CONFIGURATION_AND_VERSION))

// To resolve the value of CODE_SIGN_IDENTITY we must first resolve CON
CONFIGURATION_AND_VERSION = $(CONFIGURATION)_$(XCODE_VERSION_MAJOR)

// So, therefore...
CODE_SIGN_IDENTITY = $(CODE_SIGN_IDENTITY_$(CONFIGURATION)_$(XCODE_VERS

// Now that that variable has been resolved, we need to resolve CONFIGU
XCODE_VERSION_MAJOR = // This is defined by Xcode, it will be 0700 in X
CONFIGURATION = // This is defined by the Xcode build system, it is the

// Therefore, when we are building a Debug build in Xcode 8, it will re
CODE_SIGN_IDENTITY = $(CODE_SIGN_IDENTITY_Debug_0800)

// which means we have to look up the value of CODE_SIGN_IDENTITY_Debug
CODE_SIGN_IDENTITY_Debug_0800 = iPhone Developer

// which means the original assignment of CODE_SIGN_IDENTITY resolves a
CODE_SIGN_IDENTITY = iPhone Developer
```

This approach to resolving the values of build settings at runtime based on environmental conditions is behavior that is employed by the Xcode build system itself for many common build settings. This is a system you can rely on to work for transitioning the entire development team over to the new release of Xcode.

PROVISIONING_PROFILE

A similar approach is going to be taken to assign the value of the provisioning profile that should be used. However, this time we will need a bit more precision when performing the assignment. Since provisioning profiles only get used when deploying an executable binary, and not any executable code, we have to limit the assignment of the profile to only application targets. This can be done by creating multiple sets of xcconfigs and use different ones for each target, but it is easier to manage a single set of information rather than many.

```
PROVISIONING_PROFILE = $(PROVISIONING_PROFILE_$(WRAPPER_EXTENSION_AND_C
```

This will set up the value of `PROVISIONING_PROFILE` to be dependent on the values of the following build settings:

1. `WRAPPER_EXTENSION`
2. `CONFIGURATION`
3. `XCODE_VERSION_MAJOR`

Based on the setup that was described in the “Xcode 7 and Prior” section of the post, there are only two cases where a specific provisioning profile should be used in a build:

- Building the app for Enterprise Distribution in Xcode 7
- Building the app for Production (App Store) Distribution in Xcode 7

To get this behavior we are going to setup the `PROVISIONING_PROFILE` value to respect these two cases, but be set to “Automatic” in all others. This means that Automatic Signing in Xcode 8 will work as intented and builds can still be performed as they have been in Xcode 7.

```
PROVISIONING_PROFILE = $(PROVISIONING_PROFILE_$(WRAPPER_EXTENSION_AND_C

// First, expand the variable WRAPPER_EXTENSION_AND_CONFIGURATION_AND_V
WRAPPER_EXTENSION_AND_CONFIGURATION_AND_VERSION = $(WRAPPER_EXTENSION)_

// Therefore...
PROVISIONING_PROFILE = $(PROVISIONING_PROFILE_$(WRAPPER_EXTENSION)_$(CO

// Now do the same for CONFIGURATION_AND_VERSION
CONFIGURATION_AND_VERSION = $(CONFIGURATION)_$(XCODE_VERSION_MAJOR)

// Therefore...
PROVISIONING_PROFILE = $(PROVISIONING_PROFILE_$(WRAPPER_EXTENSION)_$(CO

// Now we know we are looking for variables with follow the pattern:
// PROVISIONING_PROFILE_$(WRAPPER_EXTENSION)_$(CONFIGURATION)_$(XCODE_V

// Based on the criteria already defined as needing the two provisionin
PROVISIONING_PROFILE_app_Enterprise_0700 = 0be1f9f5-2c59-4a11-b118-2e9d
PROVISIONING_PROFILE_app_Production_0700 = 21510e33-dbe3-4209-9506-e907
```

When this variable gets expanded at build-time, for any build that wouldn’t resolve to either of these two values, we are going to assign an empty value to the `PROVISIONING_PROFILE` build setting. This will cause it to be resolved automatically at build-time which is the expected behavior for all types of builds in Xcode 8 and the Debug build configuration in Xcode 7.

DEVELOPMENT_TEAM

As mentioned previously, the `DEVELOPMENT_TEAM` build setting is new as of Xcode 8; this means it doesn’t need to be conditionally set per version of Xcode, as only Xcode 8 will use it. The value that gets assigned to this build setting is going to depend on your situation. If you are only building for one development team, then you can do a direct assignment of this value as such:

```
DEVELOPMENT_TEAM = H3LL0W0RLD
```

However, if you are working with multiple development teams, such as a Developer and Enterprise accounts, then you may need to use the `CONFIGURATION` build setting to conditionally change it per build configuration:

```
DEVELOPMENT_TEAM_Debug = H3LL0W0RLD
DEVELOPMENT_TEAM_Enterprise = G00DC0NF1G
DEVELOPMENT_TEAM_Production = H3LL0W0RLD
```

This results in the ability to dicate to the Xcode build system that the signing identity of a different team should be used based on which build configuration is being built. This would take on the form of the assignment pattern:

```
DEVELOPMENT_TEAM = $(DEVELOPMENT_TEAM_$(CONFIGURATION))
```

The value of this build setting is important for Xcode 8 to resolve which team and signing identity should be used as part of the scheme actions for Building, Deploying, and Archiving.

PROVISIONING_PROFILE_SPECIFIER

The last component of the signing system is the new `PROVISIONING_PROFILE_SPECIFIER` build setting. For our purposes this doesn’t need to be set to anything since Automatic signing is going to take over and build for us. This means you can define it as such in the xcconfig file to prevent unintentional edits to the value:

```
PROVISIONING_PROFILE_SPECIFIER =
```

Wrapping Up

The resulting xcconfig file should look something like this:

```
CONFIGURATION_AND_VERSION = $(CONFIGURATION)_$(XCODE_VERSION_MAJOR)
WRAPPER_EXTENSION_AND_CONFIGURATION_AND_VERSION = $(WRAPPER_EXTENSION)_

CODE_SIGN_IDENTITY_Debug_0700 = iPhone Developer
CODE_SIGN_IDENTITY_Enterprise_0700 = iPhone Distribution
CODE_SIGN_IDENTITY_Production_0700 = iPhone Distribution
CODE_SIGN_IDENTITY_Debug_0800 = iPhone Developer
CODE_SIGN_IDENTITY_Enterprise_0800 = iPhone Developer
CODE_SIGN_IDENTITY_Production_0800 = iPhone Developer
CODE_SIGN_IDENTITY = $(CODE_SIGN_IDENTITY_$(CONFIGURATION_AND_VERSION))

PROVISIONING_PROFILE_app_Enterprise_0700 = 0be1f9f5-2c59-4a11-b118-2e9d
PROVISIONING_PROFILE_app_Production_0700 = 21510e33-dbe3-4209-9506-e907
PROVISIONING_PROFILE = $(PROVISIONING_PROFILE_$(WRAPPER_EXTENSION_AND_C

DEVELOPMENT_TEAM_Debug = H3LL0W0RLD
DEVELOPMENT_TEAM_Enterprise = G00DC0NF1G
DEVELOPMENT_TEAM_Production = H3LL0W0RLD
DEVELOPMENT_TEAM = $(DEVELOPMENT_TEAM_$(CONFIGURATION))

PROVISIONING_PROFILE_SPECIFIER =
```

This will yield a configration that will build without issues in Xcode 7 and 8 while taking advantage of the disparate signing systems in each. For more details on how to configure your xcconfig files, please check out this guide.

↑ Table of Contents
