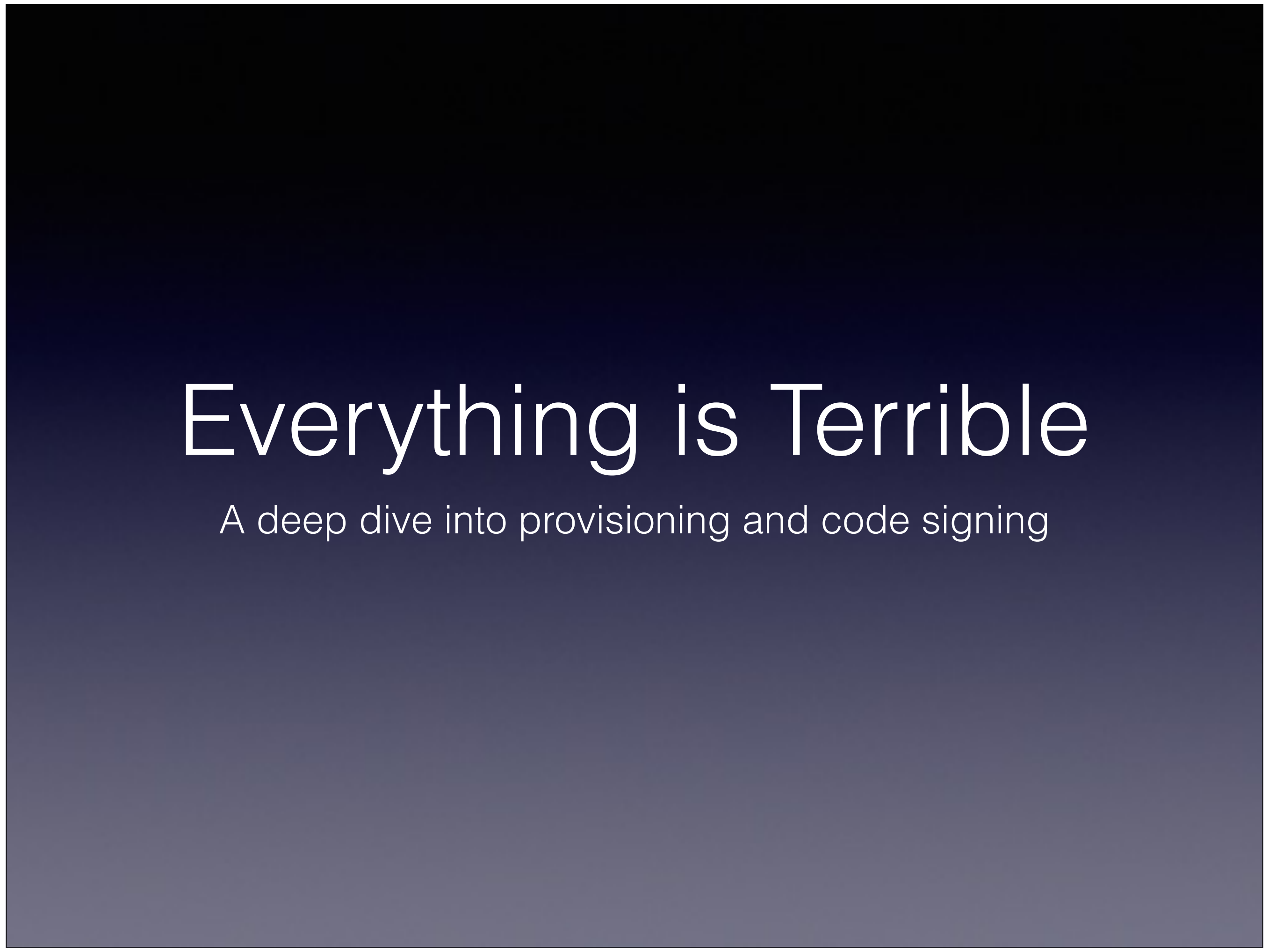
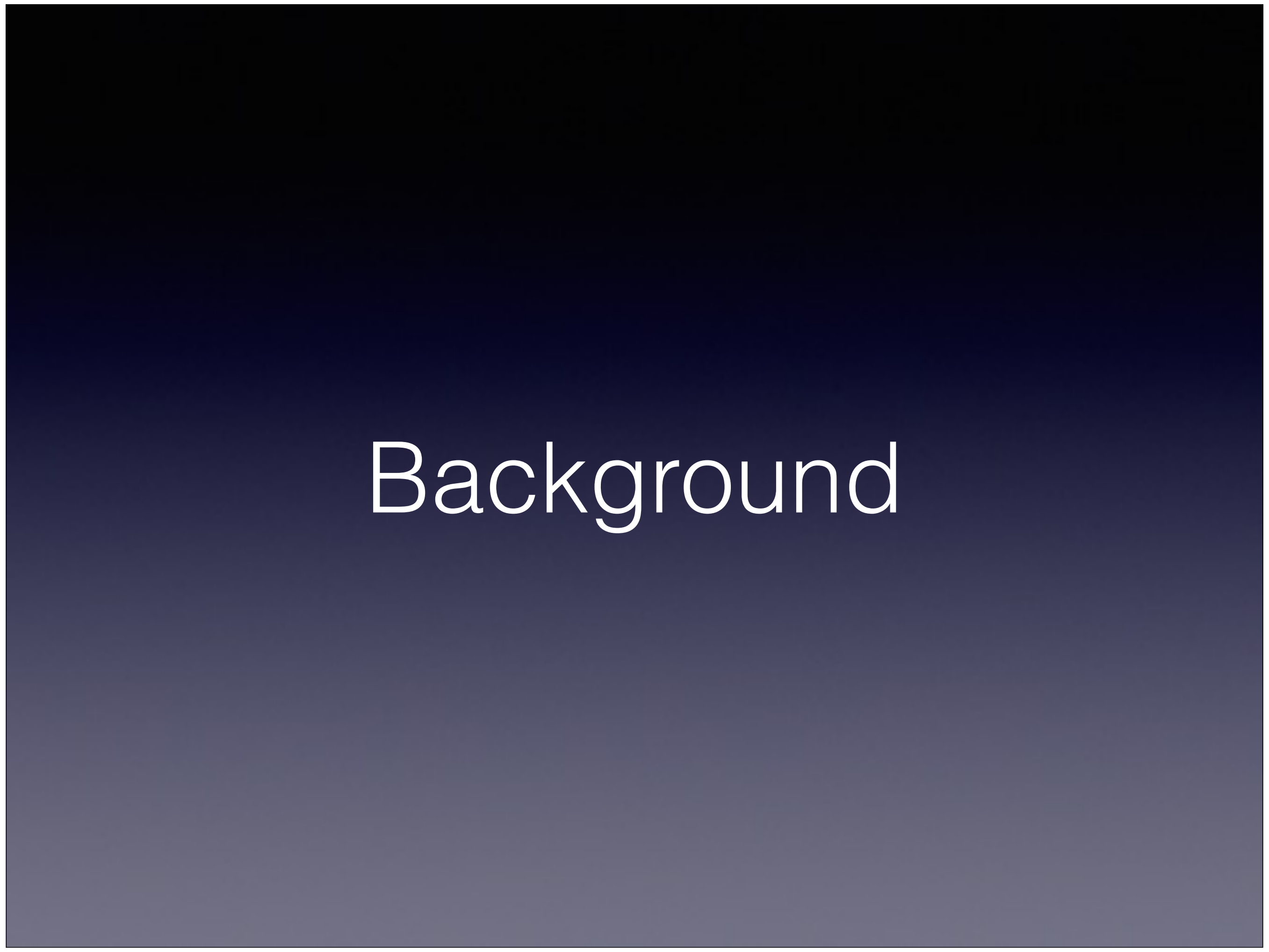


Everything is Terrible

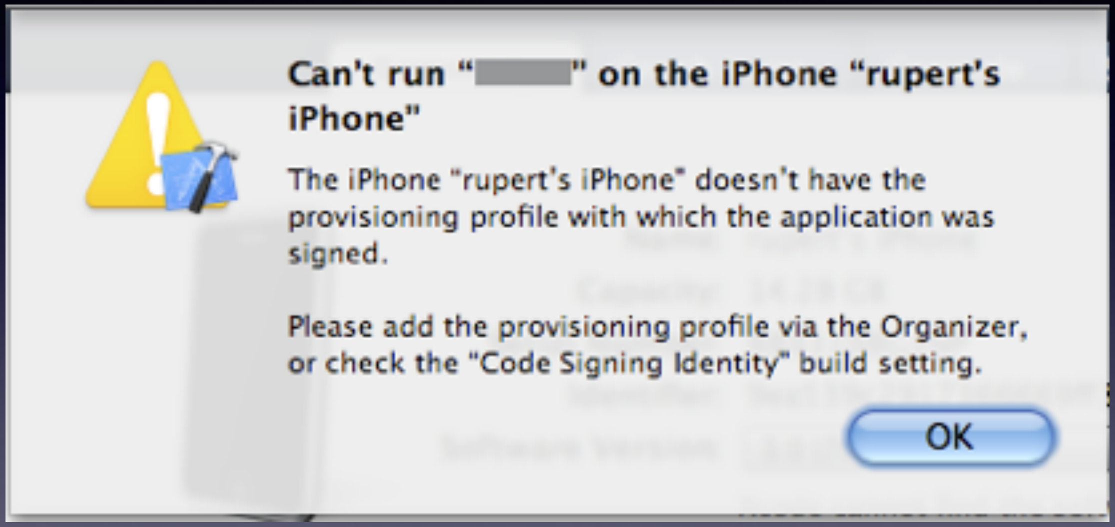
A deep dive into provisioning and code signing

Hello and welcome to “Everything is Terrible”. This is a deep dive talk into the processes behind provisioning and code signing on Apple platforms.



Background

Those that are lucky enough to not know why "Everything is Terrible", these next few slides should get you caught up to speed. For the rest of you, this will seem like a typical day in Xcode.

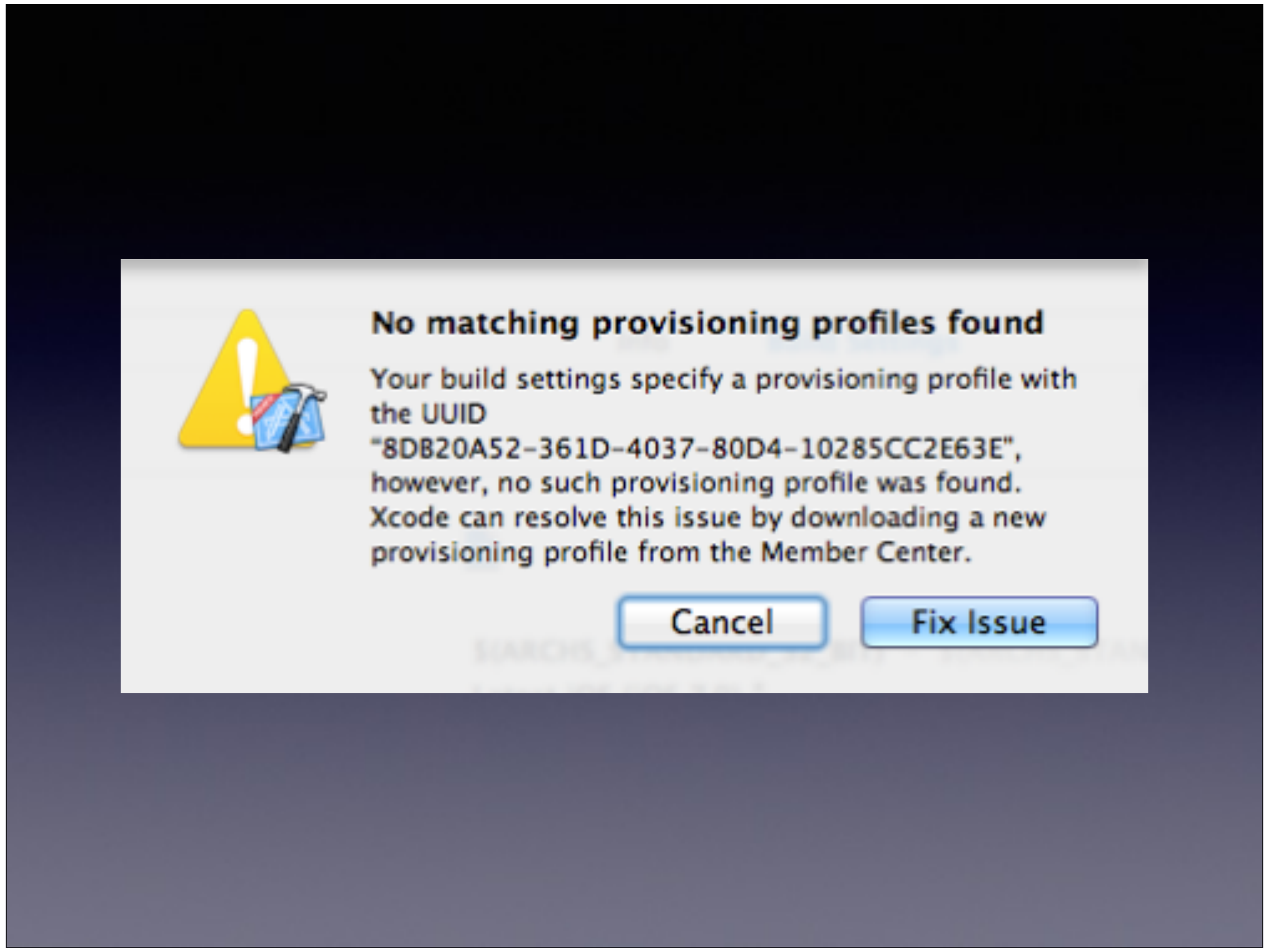


Can't run "[redacted]" on the iPhone "rupert's iPhone"

The iPhone "rupert's iPhone" doesn't have the provisioning profile with which the application was signed.

Please add the provisioning profile via the Organizer, or check the "Code Signing Identity" build setting.

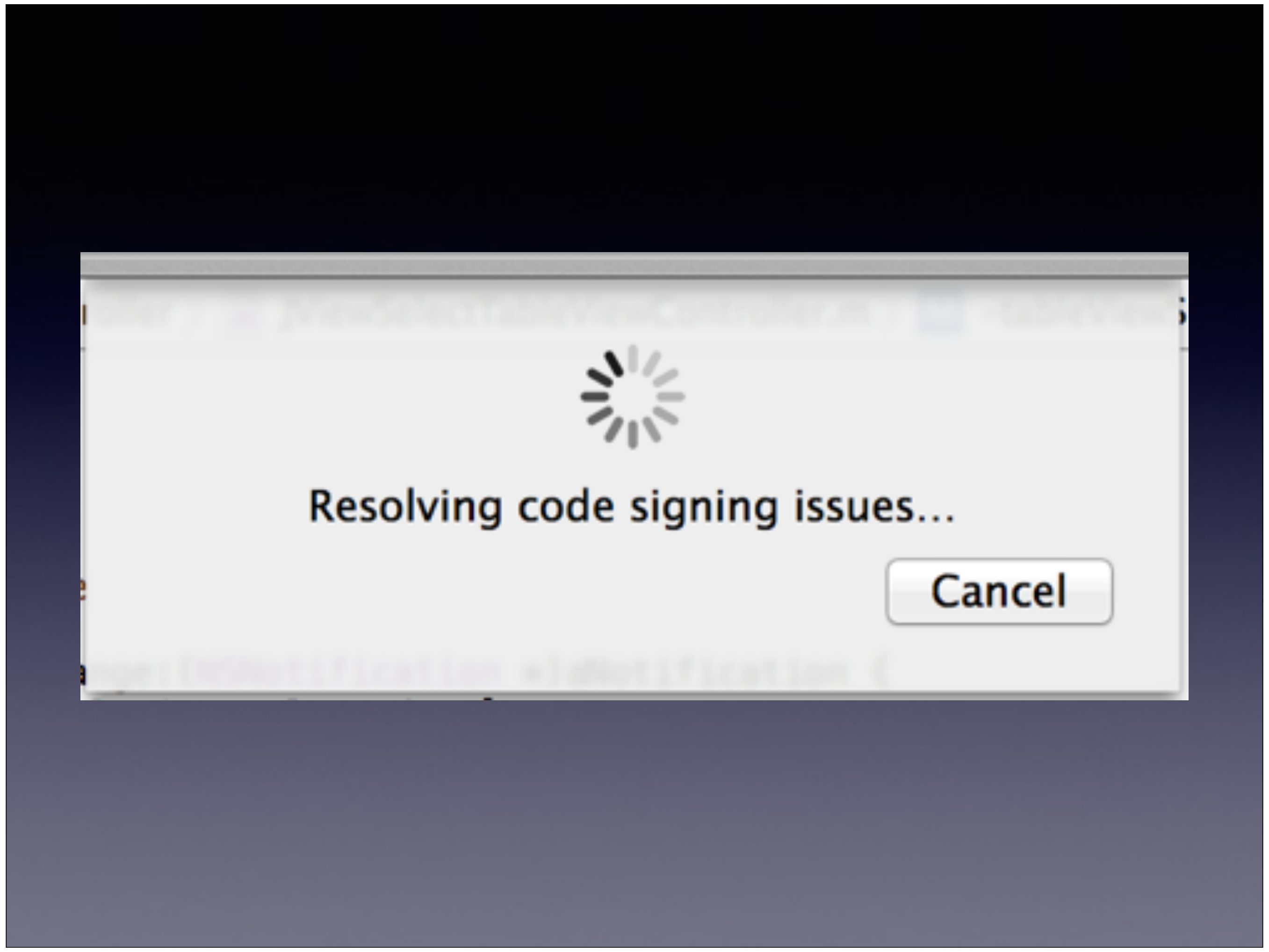
Can’t run app on the iPhone because the iPhone doesn’t have a matching provisioning profile, please add the correct one.



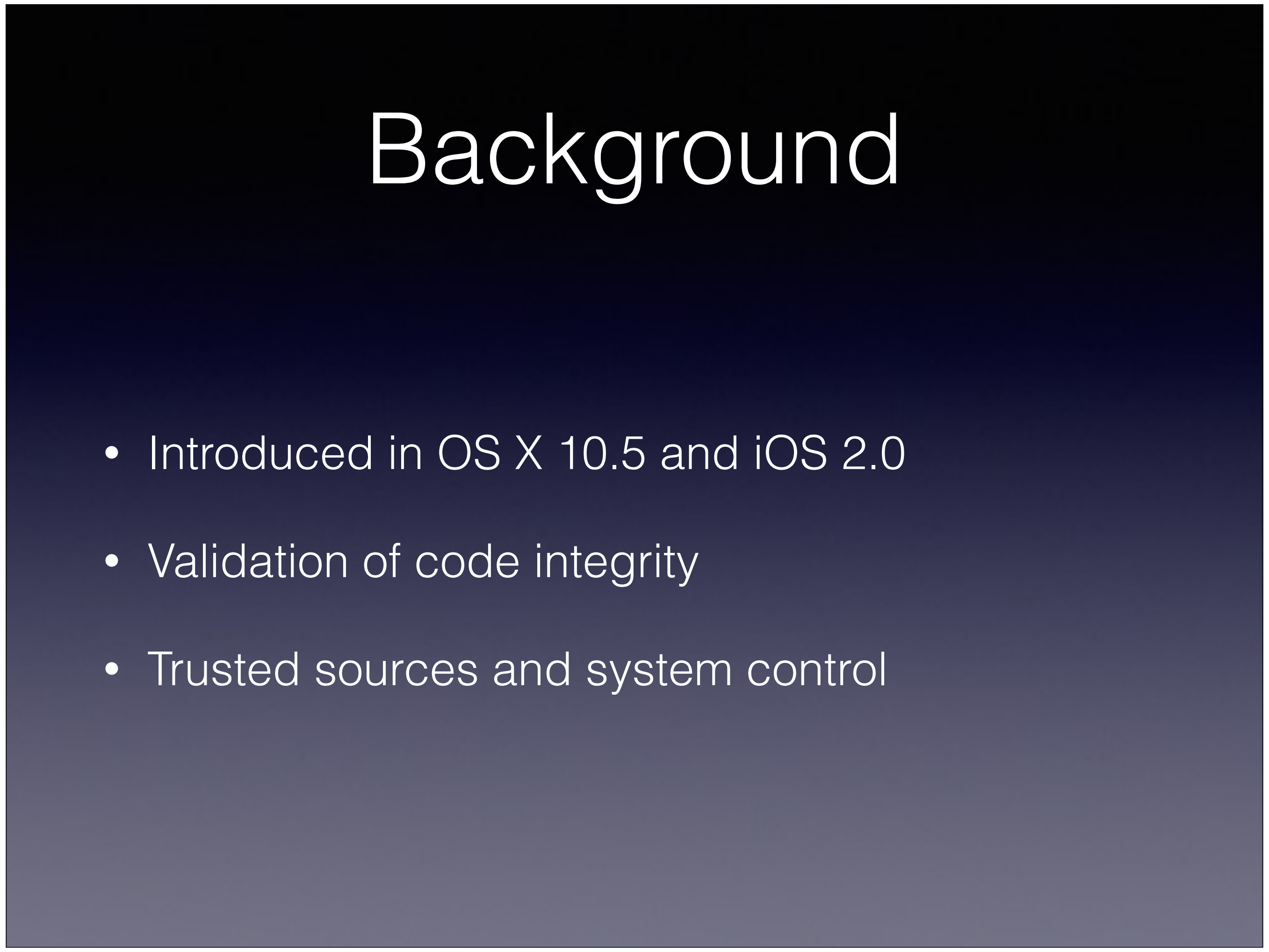
No matching provisioning profiles found

Your build settings specify a provisioning profile with the UUID "8DB20A52-361D-4037-80D4-10285CC2E63E", however, no such provisioning profile was found. Xcode can resolve this issue by downloading a new provisioning profile from the Member Center.

You specified a provisioning profile, but it doesn't exist, so go download a new one and try again.


Resolving code signing issues...

“Resolving code signing issues”

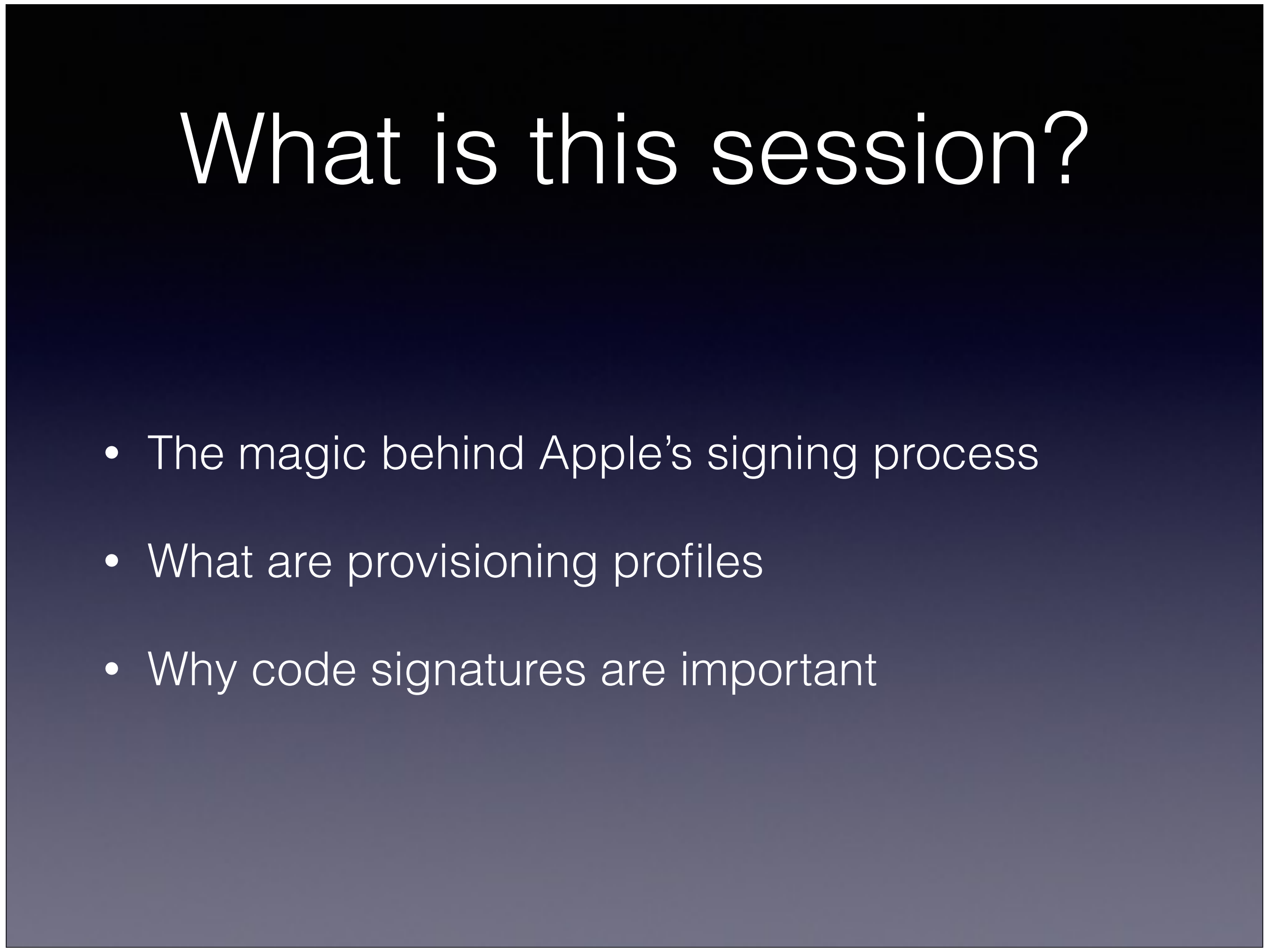


Background

- Introduced in OS X 10.5 and iOS 2.0
- Validation of code integrity
- Trusted sources and system control

Apple introduced code signing on OS X with the release of OS X 10.5 (Leopard) in 2007. Early the following year the iOS 2.0 SDK was released to developers that required all software to be signed. Since then Apple has continued to tighten down on requiring all software to be signed by third party developers. The purpose of code signatures is to ensure that no modified code is run by the operating system. This also allows for the control of what a trusted source of software is, who has permission to sign and run code.

https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html



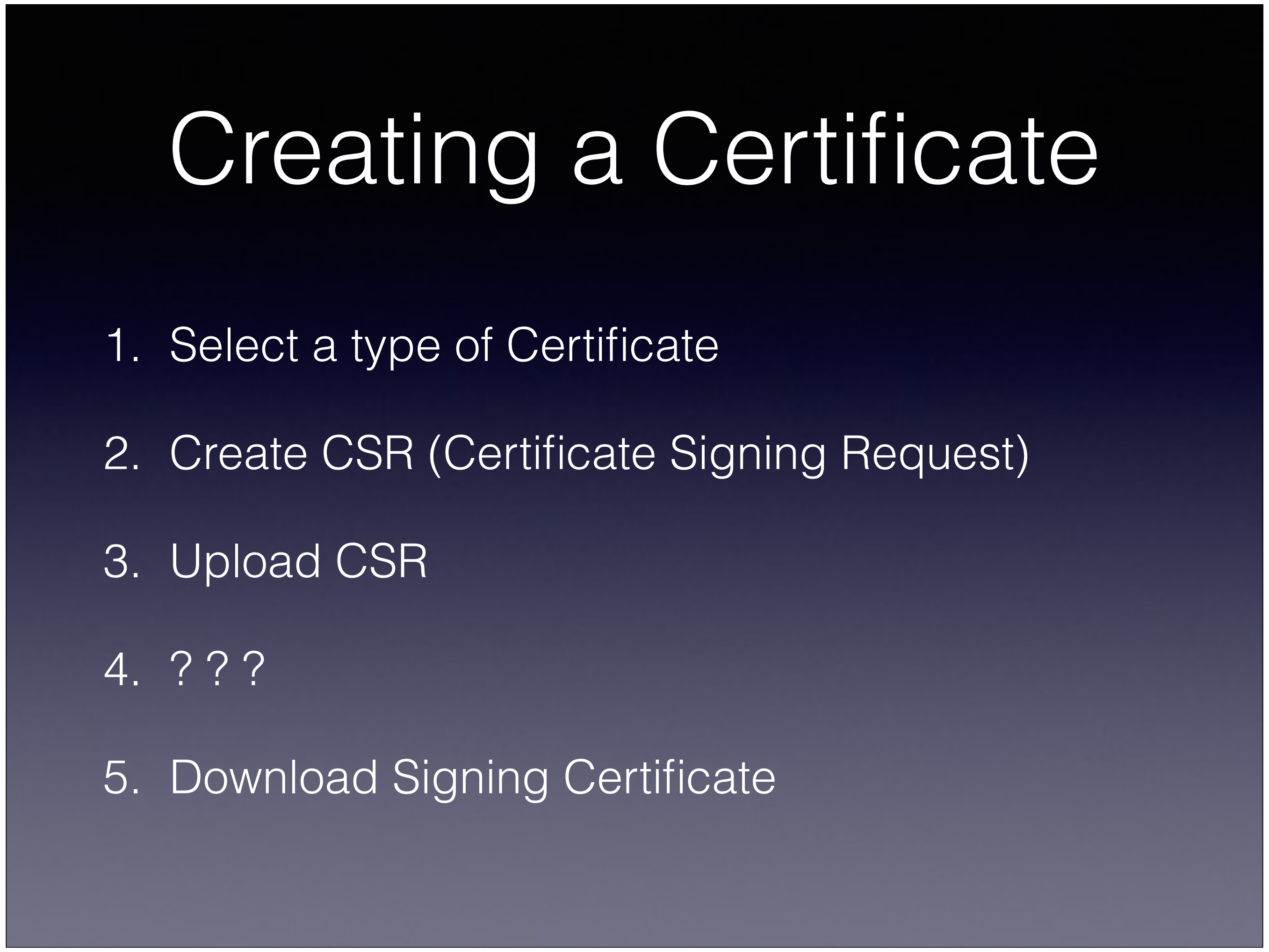
What is this session?

- The magic behind Apple's signing process
- What are provisioning profiles
- Why code signatures are important

This talk is going to walk through three major components of everything that is terrible. First, I will be walking through the process to setup a code signing identity and detailing how that process works. Second, the application of code signing and the deployment process through provisioning profiles, that we all know and love. Finally, why are code signatures significant; what do they mean for developers and users.

Signing Certificates

Signing Certificates, what are they?



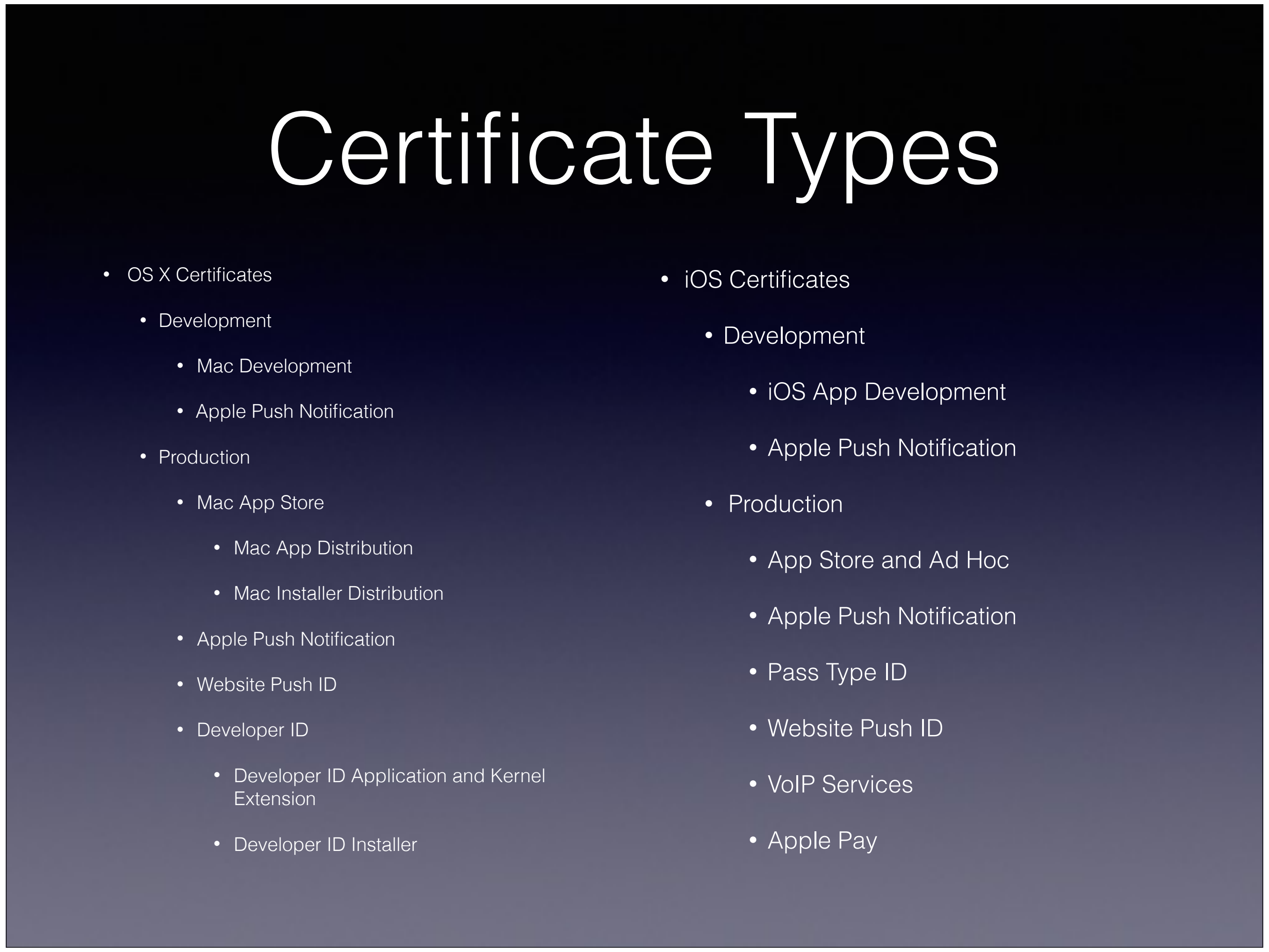
Creating a Certificate

1. Select a type of Certificate
2. Create CSR (Certificate Signing Request)
3. Upload CSR
4. ? ? ?
5. Download Signing Certificate

At this point there are countless blog posts and tutorials to guide developers through the process of creating a signing certificate. To summarize the process, the steps look like this:

1. Select the type of signing certificate you want from the Apple Developer Portal
2. Create the Certificate Signing Request through the Keychain app
3. Upload the signing request
4. MAGIC!
5. Download the signing certificate

What this doesn't cover is how complicated this process can be, and we are doing with each step. As a result, most of us fix codesign and provisioning issues by deleting everything and starting from scratch.

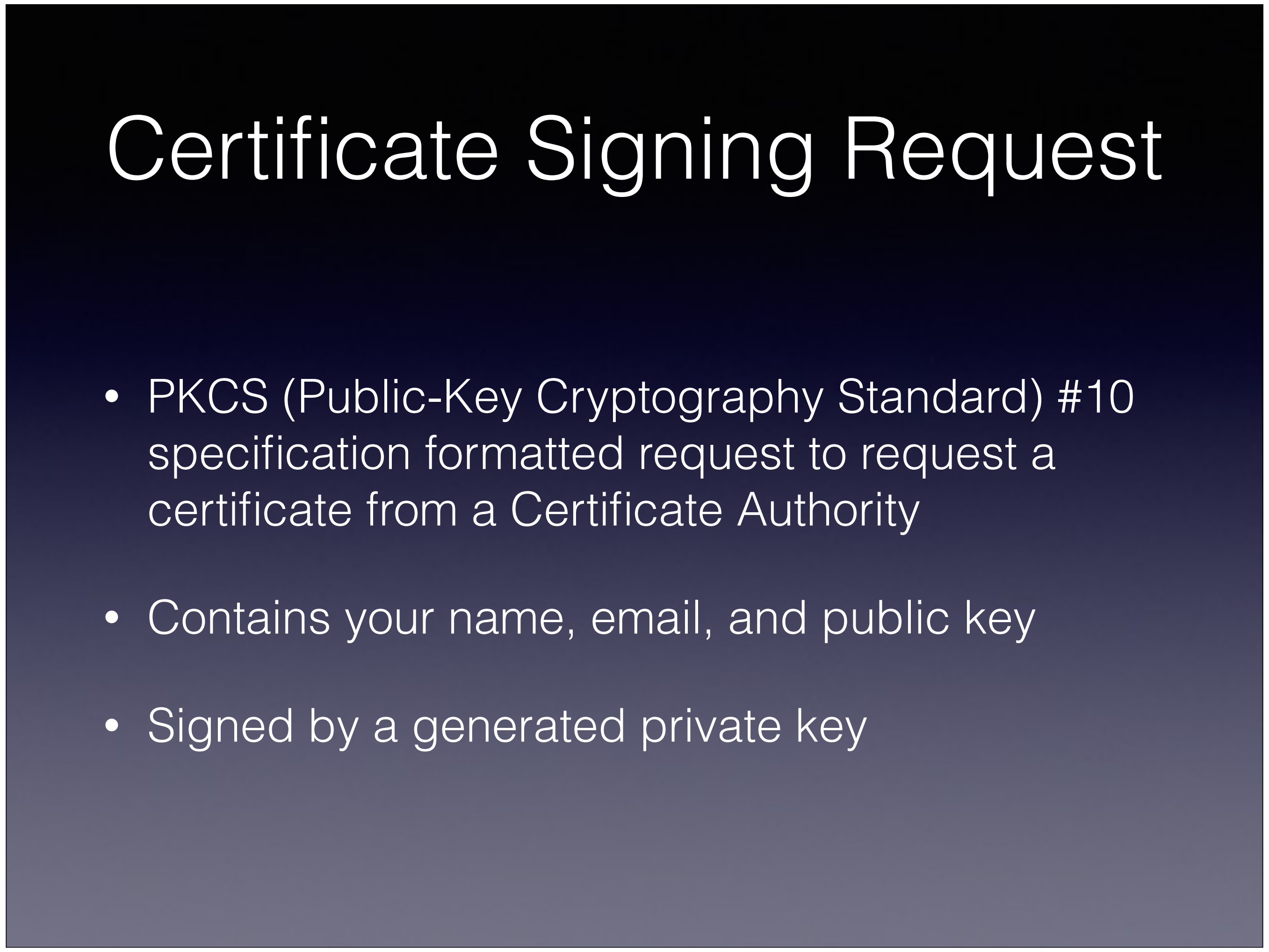

Certificate Types

- OS X Certificates
  - Development
    - Mac Development
    - Apple Push Notification
  - Production
    - Mac App Store
      - Mac App Distribution
      - Mac Installer Distribution
    - Apple Push Notification
    - Website Push ID
    - Developer ID
      - Developer ID Application and Kernel Extension
      - Developer ID Installer
- iOS Certificates
  - Development
    - iOS App Development
    - Apple Push Notification
  - Production
    - App Store and Ad Hoc
    - Apple Push Notification
    - Pass Type ID
    - Website Push ID
    - VoIP Services
    - Apple Pay

Step 1, pick the type of certificate you need. This sounds fairly simple, right?

Sure.

The document that use to explain the process of selecting the certificate type was retired and never replaced with adequate explanations when selecting which type of certificate to use. The current certificate creation “wizard” only gives 1-2 sentence explanation as to what each of these types of certificates are and why we might need multiple.



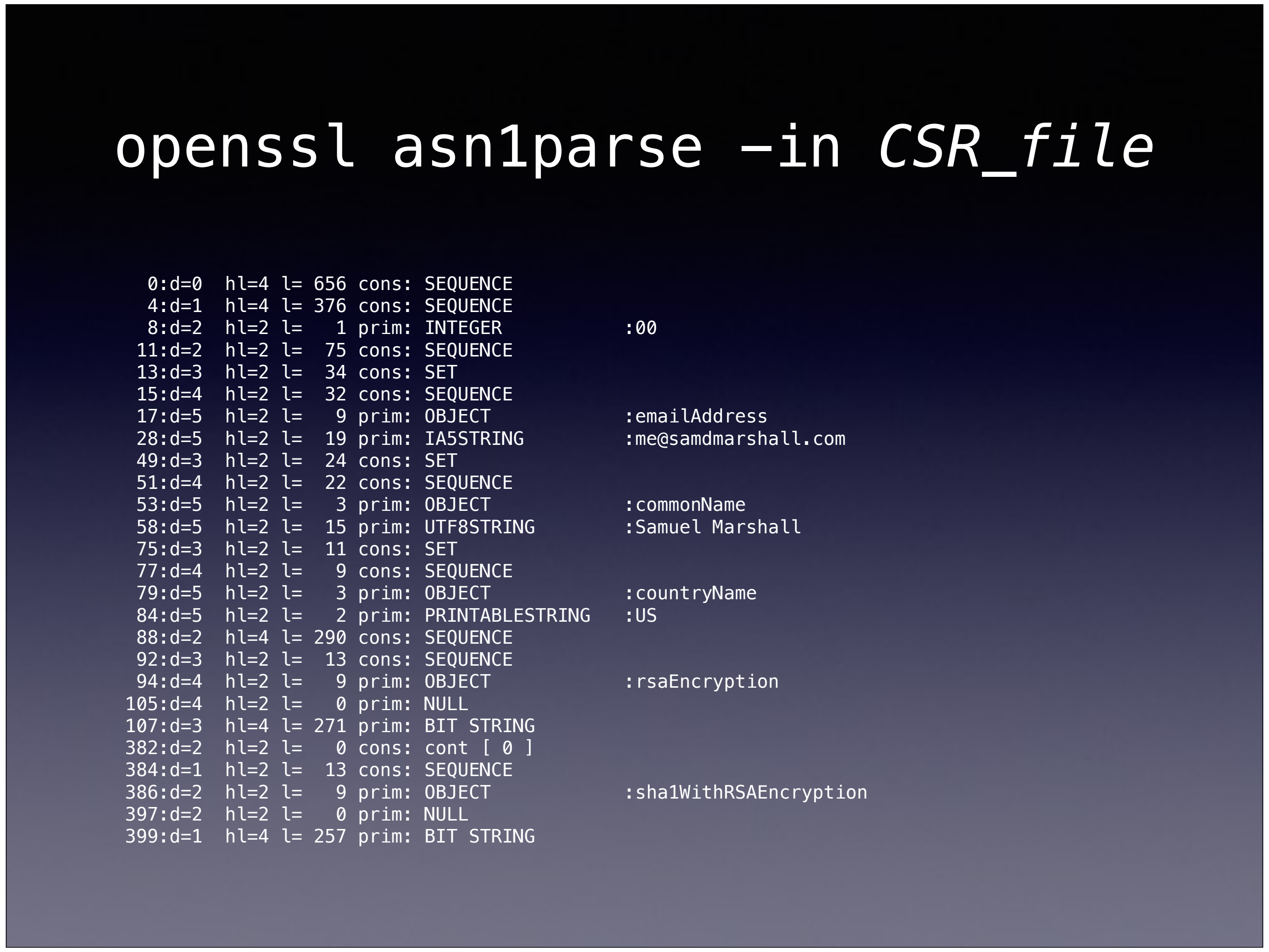
Certificate Signing Request

- PKCS (Public-Key Cryptography Standard) #10 specification formatted request to request a certificate from a Certificate Authority
- Contains your name, email, and public key
- Signed by a generated private key

Step 2, Creating the Certificate Signing Request. This is typically done using the Keychain app on OS X, however the format of this request adheres to the PKCS 10 format and can be generated by a number of tools on almost any platform. It contains very minimal amounts of information, specifically: name, email, and the public key of the signer. This is signed using a newly generated private key. This is what identifies you as the owner of the signing certificate.

http://en.wikipedia.org/wiki/Certificate_signing_request

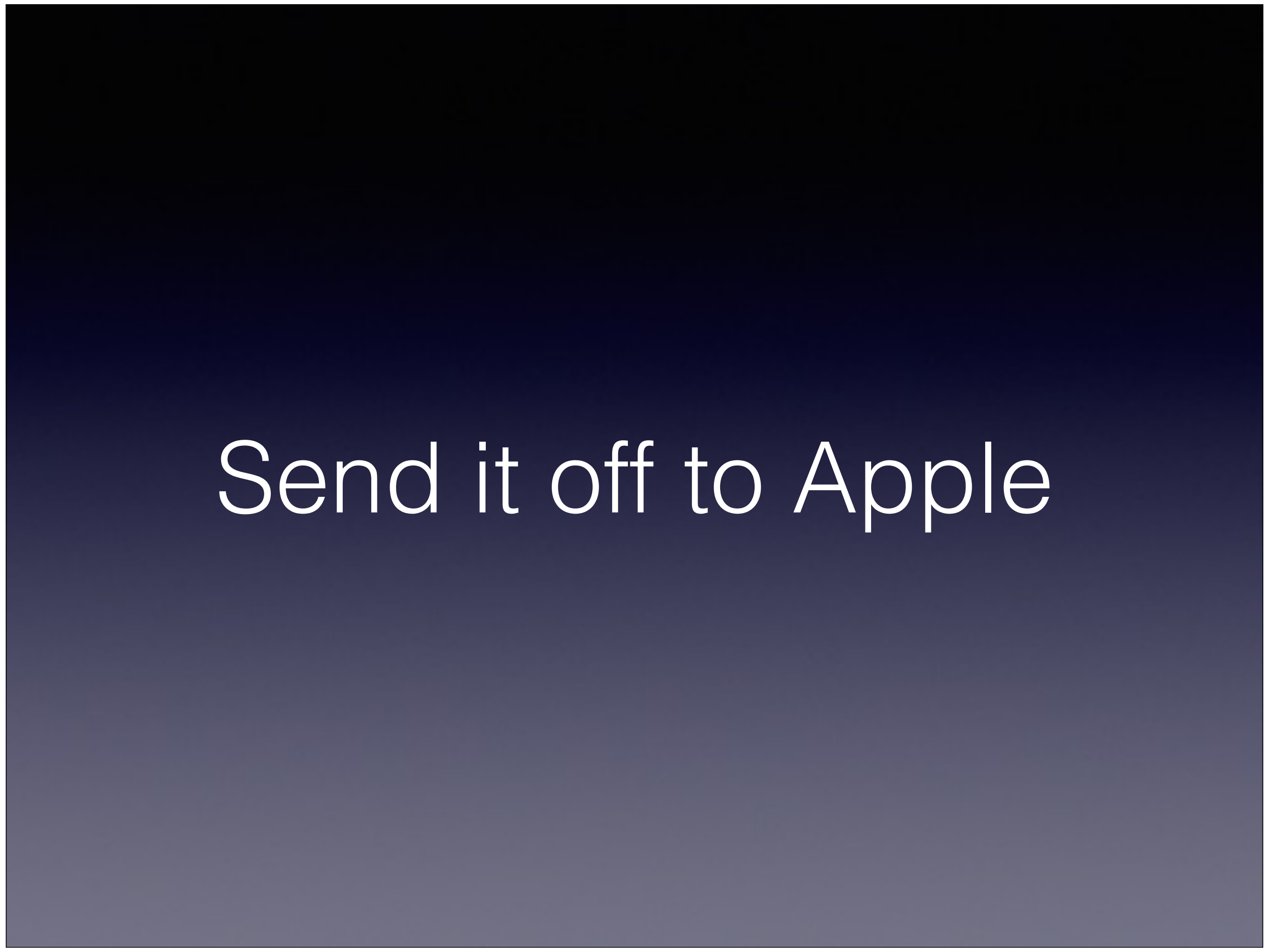
http://tools.ietf.org/html/rfc2986



openssl asn1parse -in *CSR_file*

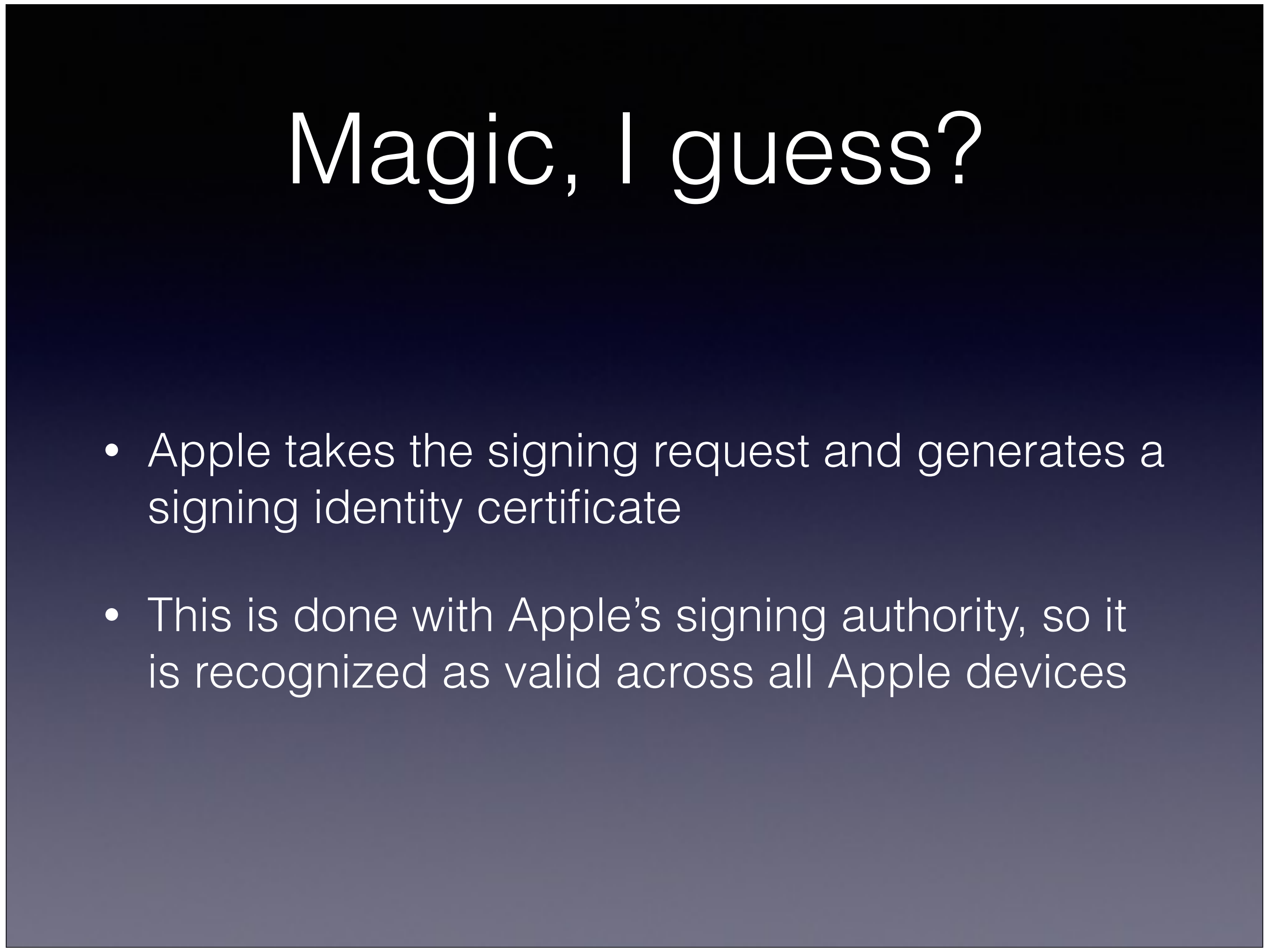
```
    0:d=0  hl=4 l= 656 cons: SEQUENCE
    4:d=1  hl=4 l= 376 cons: SEQUENCE
    8:d=2  hl=2 l=   1 prim: INTEGER           :00
   11:d=2  hl=2 l=  75 cons: SEQUENCE
   13:d=3  hl=2 l=  34 cons: SET
   15:d=4  hl=2 l=  32 cons: SEQUENCE
   17:d=5  hl=2 l=   9 prim: OBJECT            :emailAddress
   28:d=5  hl=2 l=  19 prim: IA5STRING         :me@samdmarshall.com
   49:d=3  hl=2 l=  24 cons: SET
   51:d=4  hl=2 l=  22 cons: SEQUENCE
   53:d=5  hl=2 l=   3 prim: OBJECT            :commonName
   58:d=5  hl=2 l=  15 prim: UTF8STRING        :Samuel Marshall
   75:d=3  hl=2 l=  11 cons: SET
   77:d=4  hl=2 l=   9 cons: SEQUENCE
   79:d=5  hl=2 l=   3 prim: OBJECT            :countryName
   84:d=5  hl=2 l=   2 prim: PRINTABLESTRING   :US
   88:d=2  hl=4 l= 290 cons: SEQUENCE
   92:d=3  hl=2 l=  13 cons: SEQUENCE
   94:d=4  hl=2 l=   9 prim: OBJECT            :rsaEncryption
  105:d=4  hl=2 l=   0 prim: NULL
  107:d=3  hl=4 l= 271 prim: BIT STRING
  382:d=2  hl=2 l=   0 cons: cont [ 0 ]
  384:d=1  hl=2 l=  13 cons: SEQUENCE
  386:d=2  hl=2 l=   9 prim: OBJECT            :sha1WithRSAEncryption
  397:d=2  hl=2 l=   0 prim: NULL
  399:d=1  hl=4 l= 257 prim: BIT STRING
```

If you want to have a peek into what the CSR contains, you can use the openssl tool to dump out the contents of the signing request. This is the identifying information that gets sent to apple that should represent you as a signer.



Send it off to Apple

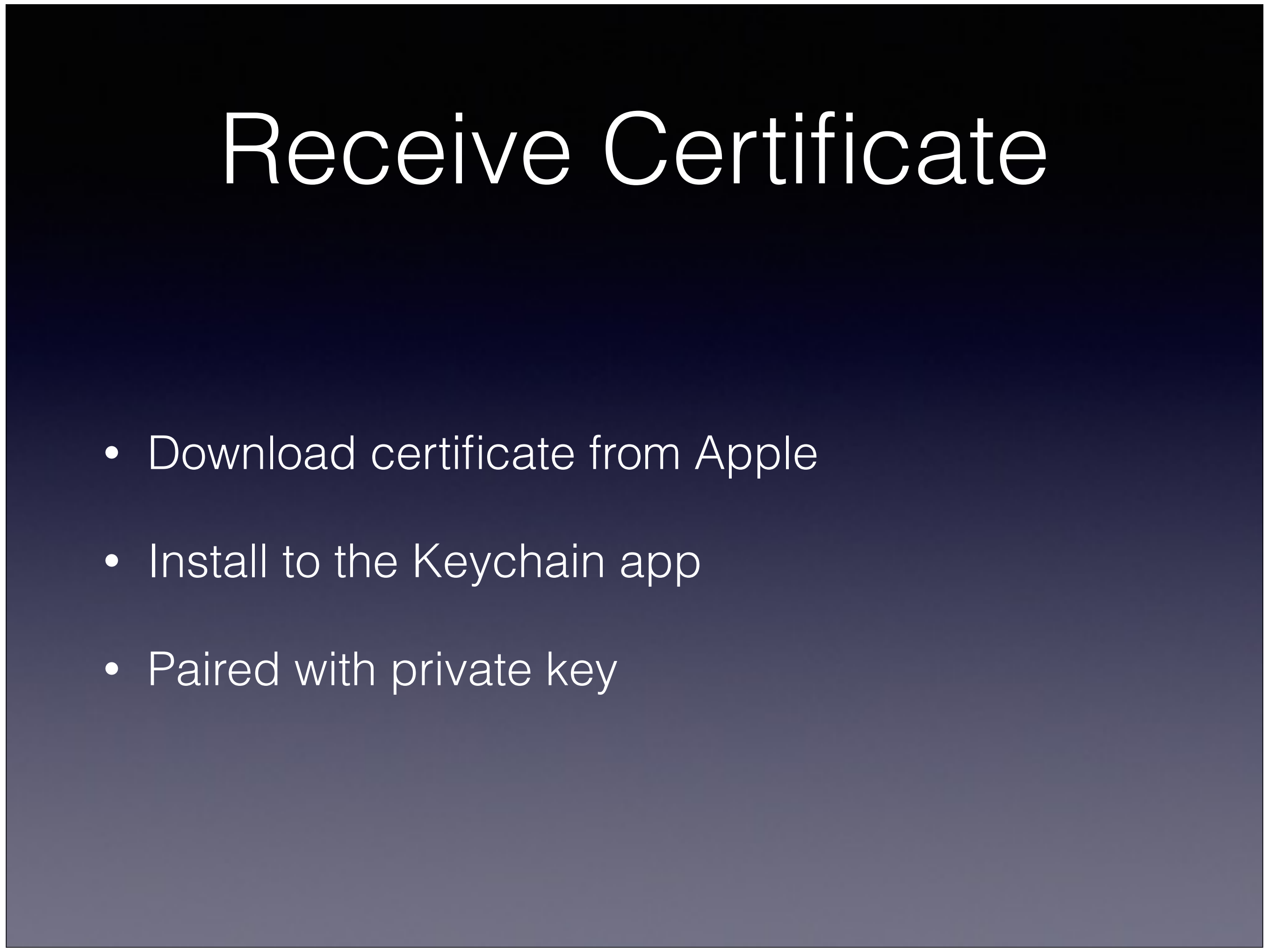
Once the signing request is created, you upload it to Apple. Then what?



Magic, I guess?

- Apple takes the signing request and generates a signing identity certificate
- This is done with Apple's signing authority, so it is recognized as valid across all Apple devices

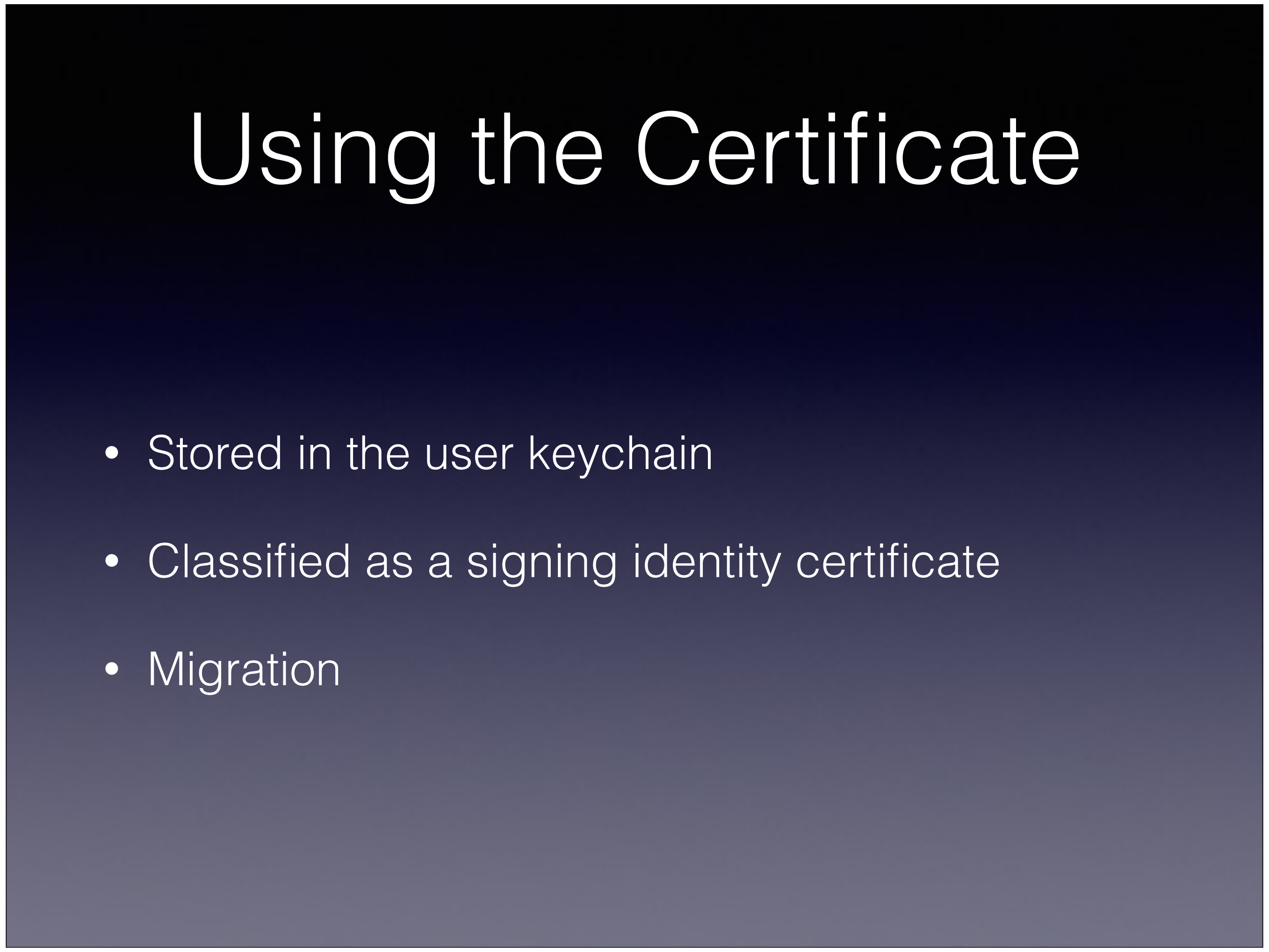
Magic, or that is what it seems like anyway. Apple takes your signing request and creates a signing identity certificate. This is signed by Apple's Root Certificate Authorities so it will validate across all devices running Apple's operating systems. It is a standard process for cryptographic signatures, for all intents and purposes this could be called "magic" for most.



Receive Certificate

- Download certificate from Apple
- Install to the Keychain app
- Paired with private key

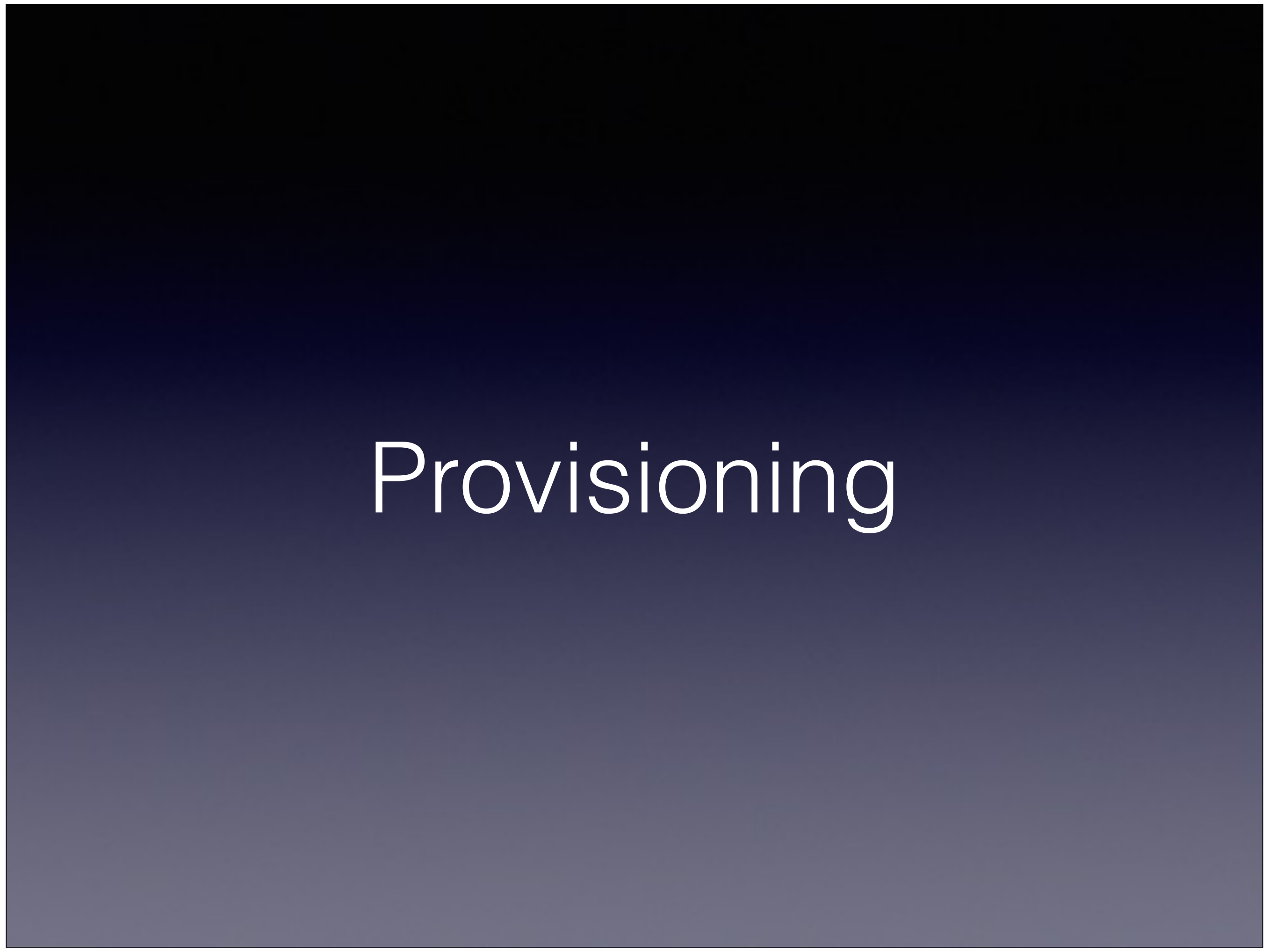
Receiving the certificate, this is probably only step that cannot be messed up. Apple gives you the signing certificate to download and you are expected to install this into a keychain on your computer. When this is installed it is paired with the private key used for signing. While this defaults to be installed into your user based keychain, I would recommend installing these to a separate keychain just used for signing. This mitigates a lot of work resolving broken signing identities.



Using the Certificate

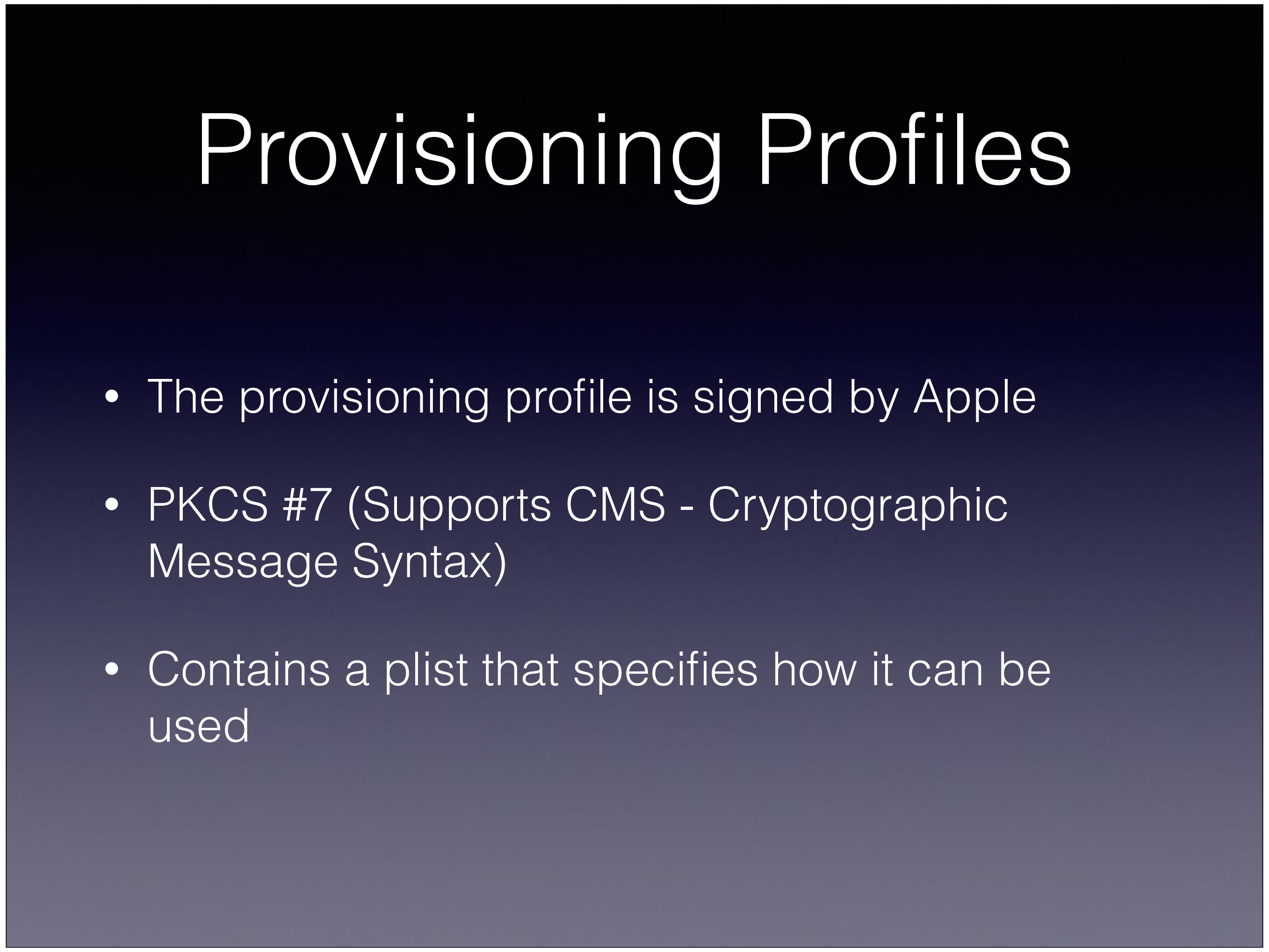
- Stored in the user keychain
- Classified as a signing identity certificate
- Migration

Once the signing certificate has been downloaded and installed, they are stored in the user keychain. They can be found by launching the Keychain app and selecting “My Certificates” from the sidebar. This view lists any user created certificate, signing certificates are classified as “Signing Identity Certificates” and can be queried as such from the Keychain API. These are identifiable by properties on the certificate; such as name, and issuer. They also are associated with the private key generated when the CSR was created. This is how code signatures can validate authenticity of the signing certificate. Many issues arise when trying to migrate these certificates between computers. Both the certificate and the private key it is associated with must be imported before another computer can sign with your identity.



Provisioning

Provisioning … Yeah, that.

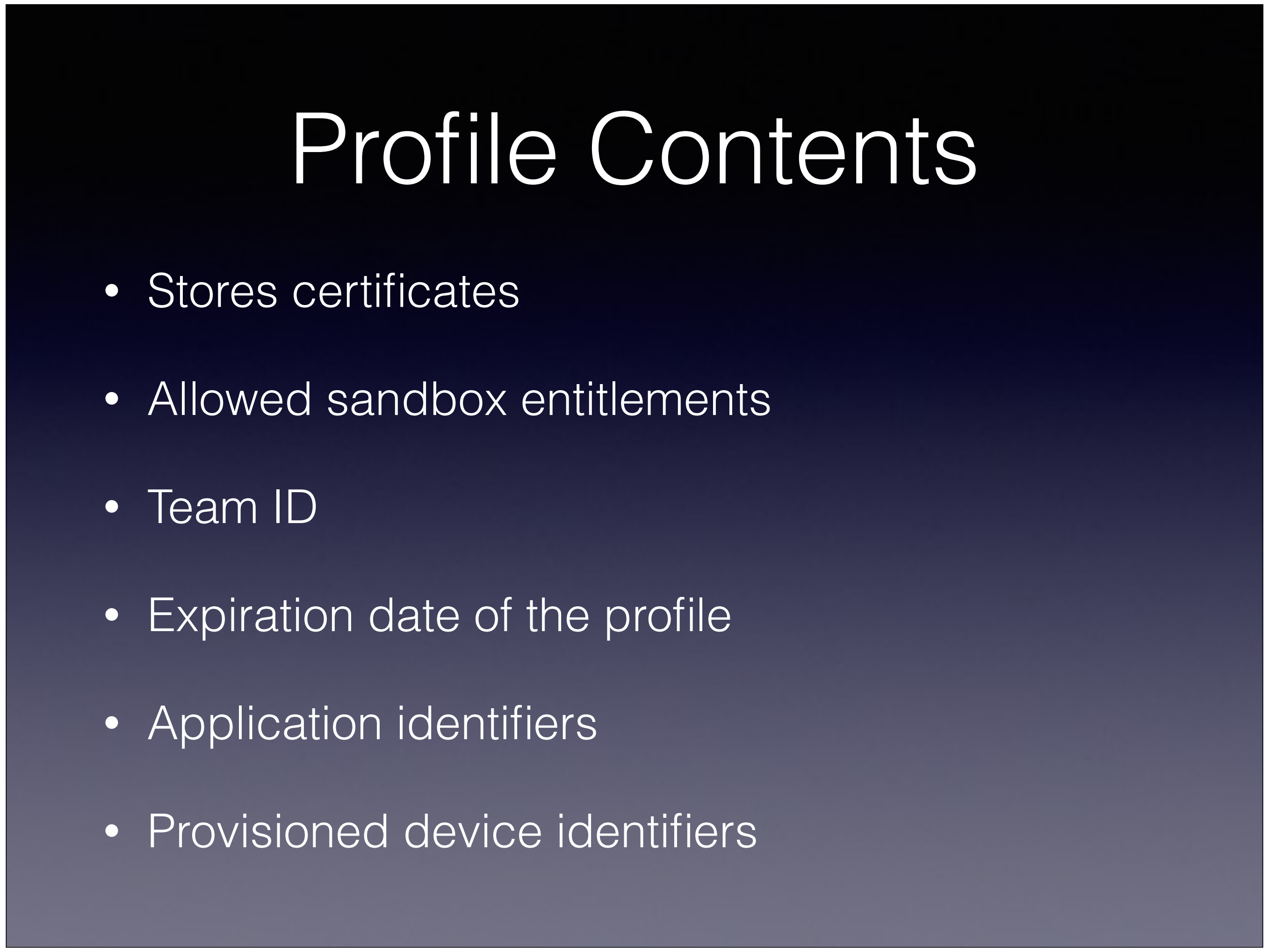


Provisioning Profiles

- The provisioning profile is signed by Apple
- PKCS #7 (Supports CMS - Cryptographic Message Syntax)
- Contains a plist that specifies how it can be used

What are provisioning profiles? Provisioning profiles are used to define where code is allowed to be run. They are signed by Apple and are PKCS 7 formatted, which is used for certificate distribution and management. This is an encrypted plist that contains data that defines what code is allowed to run where.

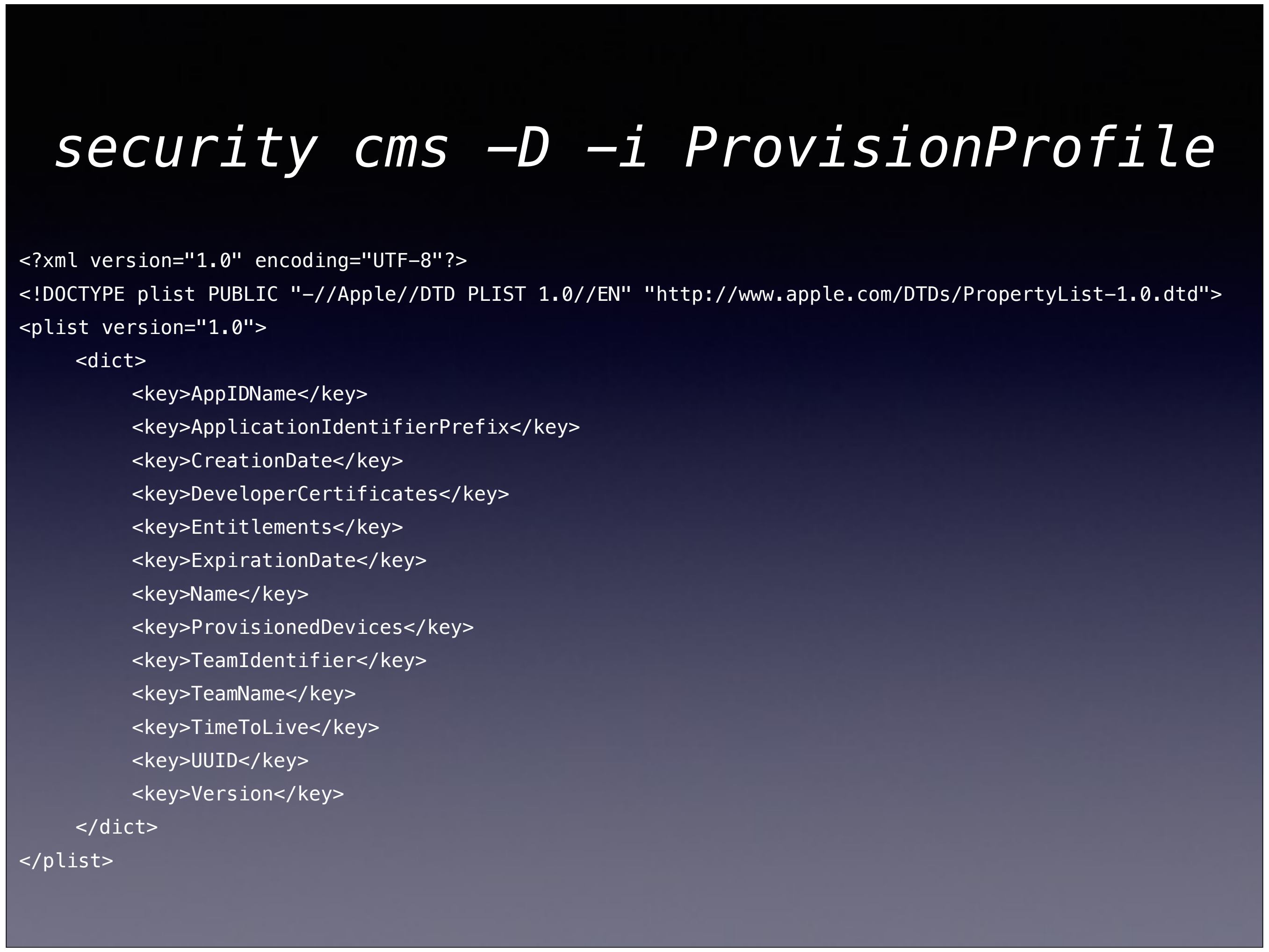
http://tools.ietf.org/html/rfc2315


Profile Contents

- Stores certificates
- Allowed sandbox entitlements
- Team ID
- Expiration date of the profile
- Application identifiers
- Provisioned device identifiers

Provisioning profiles contain a lot of information dictates where it can be run and installed on.

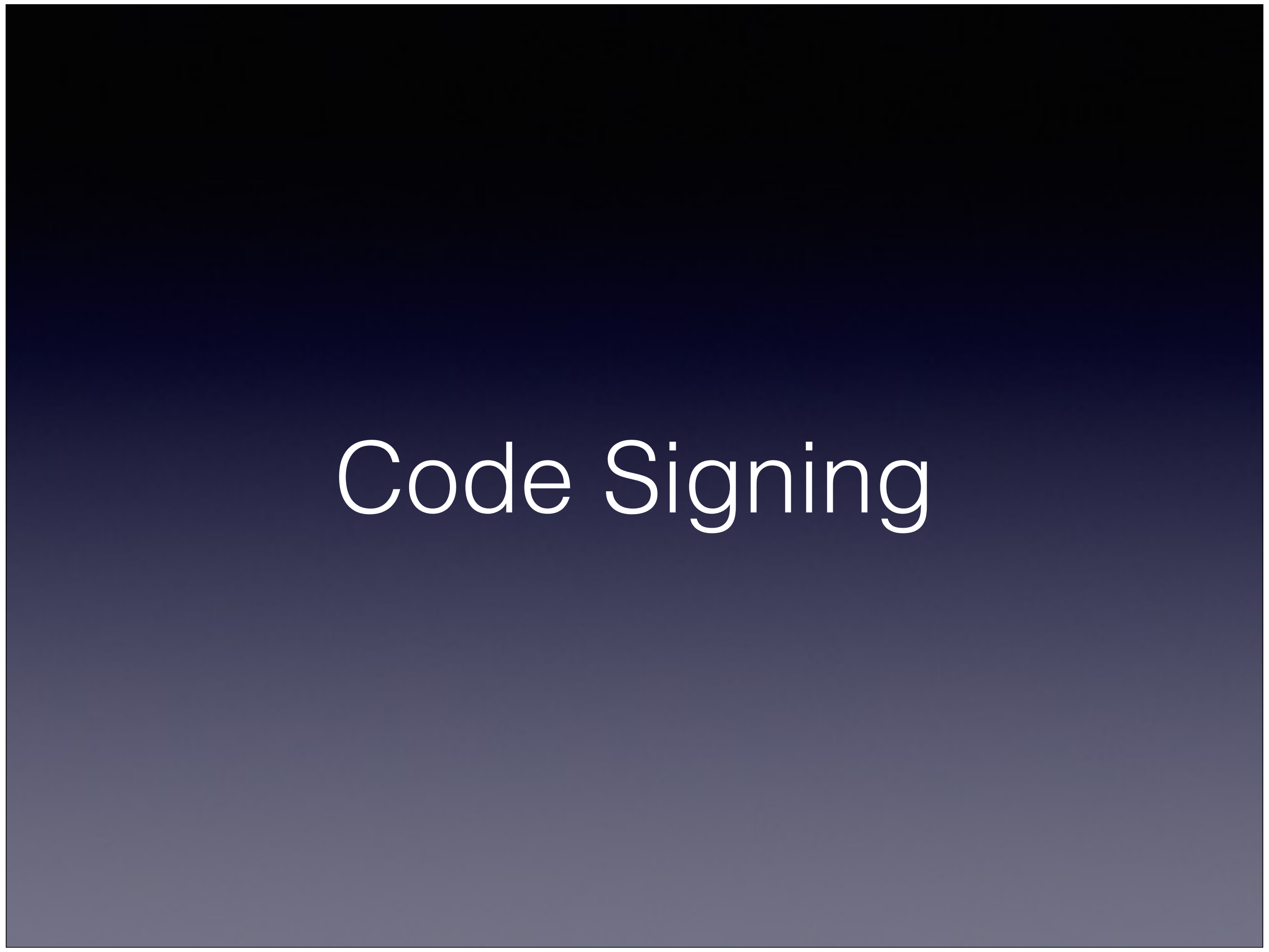
* It contains copies of relevant certificates
* sandbox entitlements that need to be approved to run.
* Team Identifiers, these must match the certificates you are signing the binary with
* expiration date of the profile, how long the software can be run before it cannot run anymore
* allow application identifiers, this is what permits your bundle identifiers to run, but not others.
* device identifiers, these are the unique divide identifiers that identify the hardware device you plan to run this software on



security cms -D -i ProvisionProfile

```
<?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>AppIDName</key>
        <key>ApplicationIdentifierPrefix</key>
        <key>CreationDate</key>
        <key>DeveloperCertificates</key>
        <key>Entitlements</key>
        <key>ExpirationDate</key>
        <key>Name</key>
        <key>ProvisionedDevices</key>
        <key>TeamIdentifier</key>
        <key>TeamName</key>
        <key>TimeToLive</key>
        <key>UUID</key>
        <key>Version</key>
    </dict>
</plist>
```

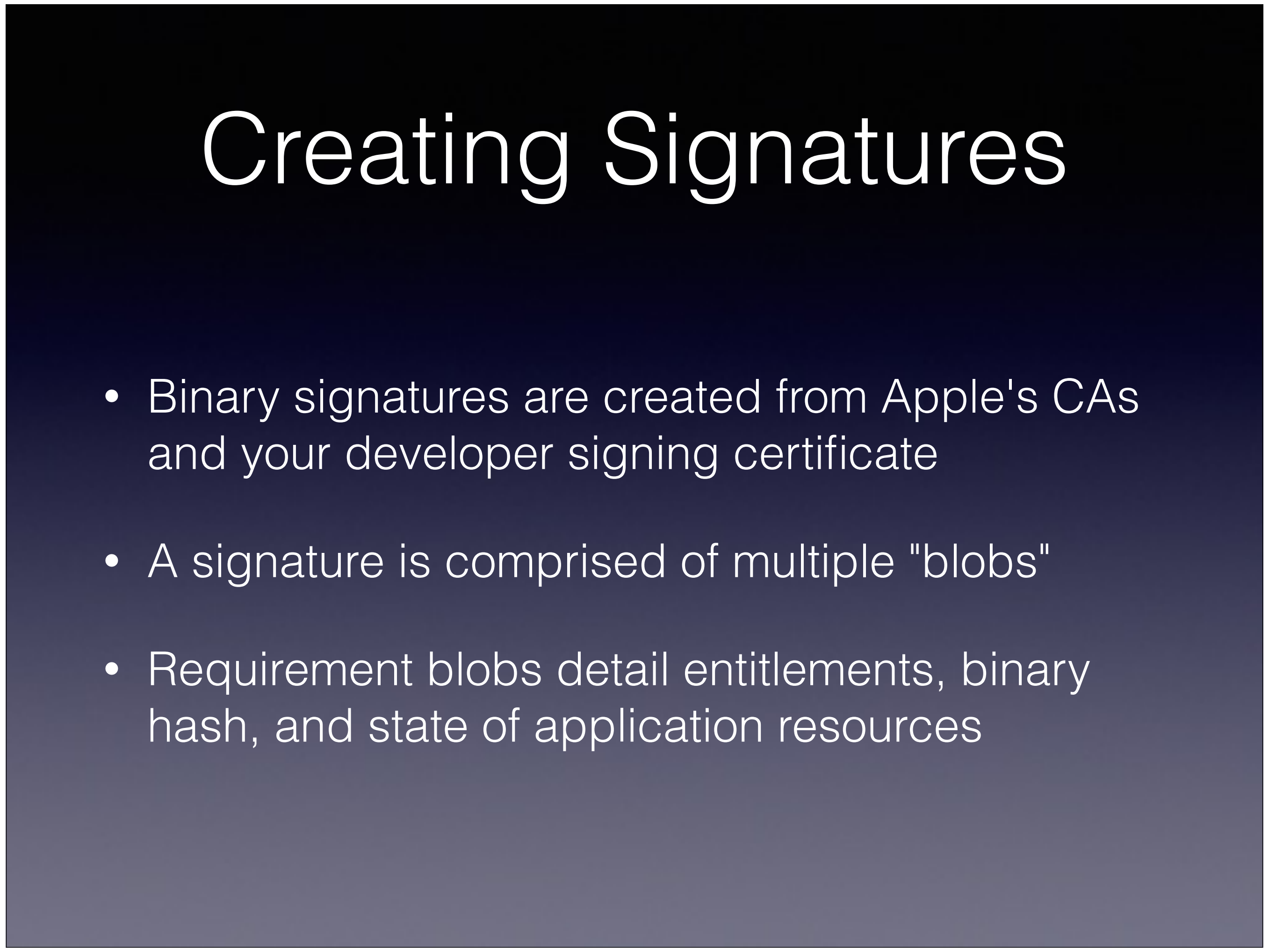
To verify the contents of a provisioning profile, you can pass it through the command listed on this slide. What will be output is listed below, which is the human-readable plist file that describes what and where can be run. This can be used to check against what devices are authorized to install on and the supported signing identities it has.



Code Signing

https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/AboutCS/AboutCS.html

https://developer.apple.com/library/mac/technotes/tn2206/_index.html

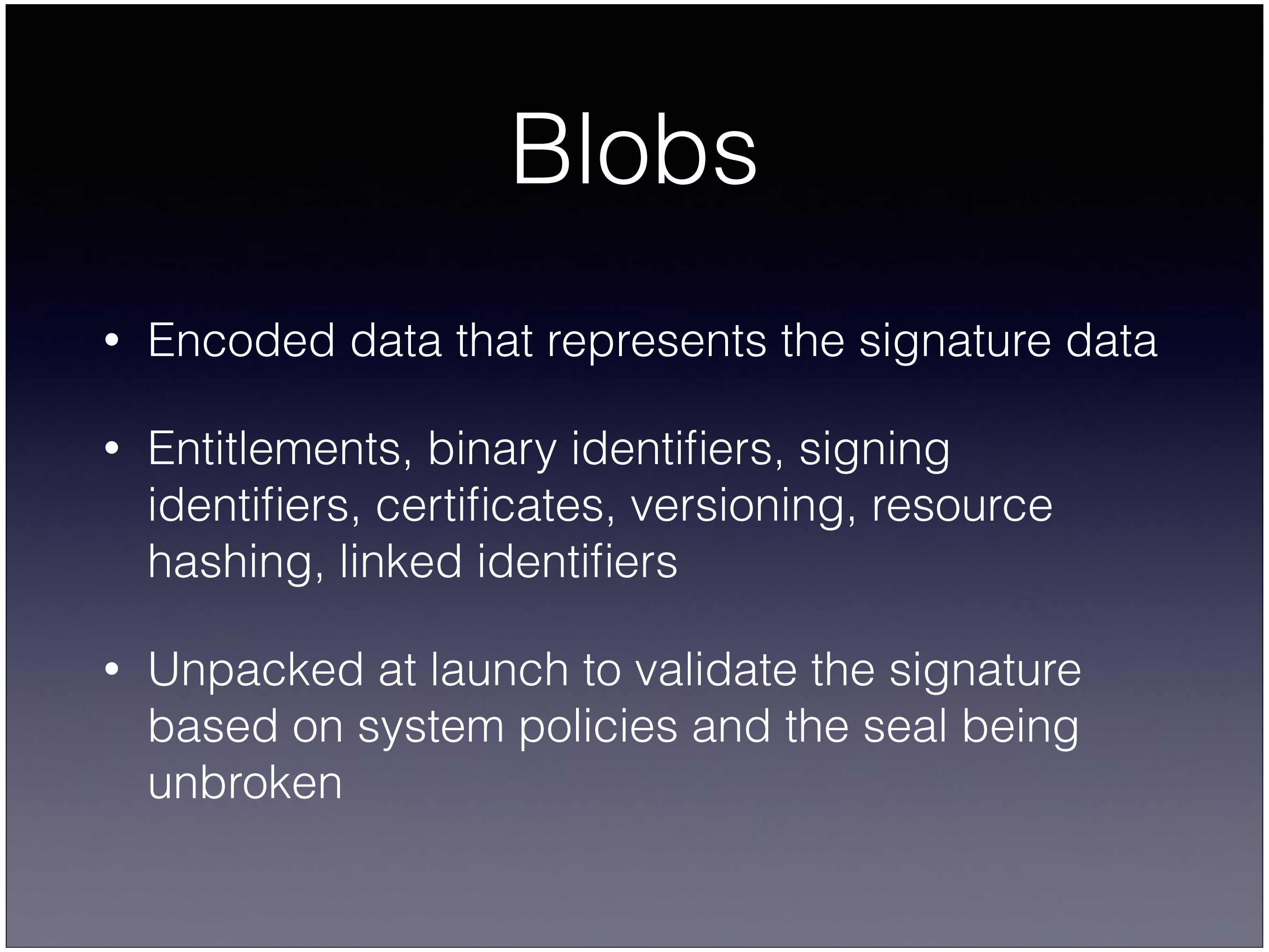


Creating Signatures

- Binary signatures are created from Apple's CAs and your developer signing certificate
- A signature is comprised of multiple "blobs"
- Requirement blobs detail entitlements, binary hash, and state of application resources

Signing code. We use a tool called `codesign` to create signatures. These signatures are created using your developer signing certificate and apple's own certificate authorities. To embed all this data, it is encoded into a set of rules using "Requirement Language" and stored as blobs inside the binary file. These blobs define a set of rules which describe the requested entitlements, code hashes, and state of application resources.

https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/RequirementLang/RequirementLang.html



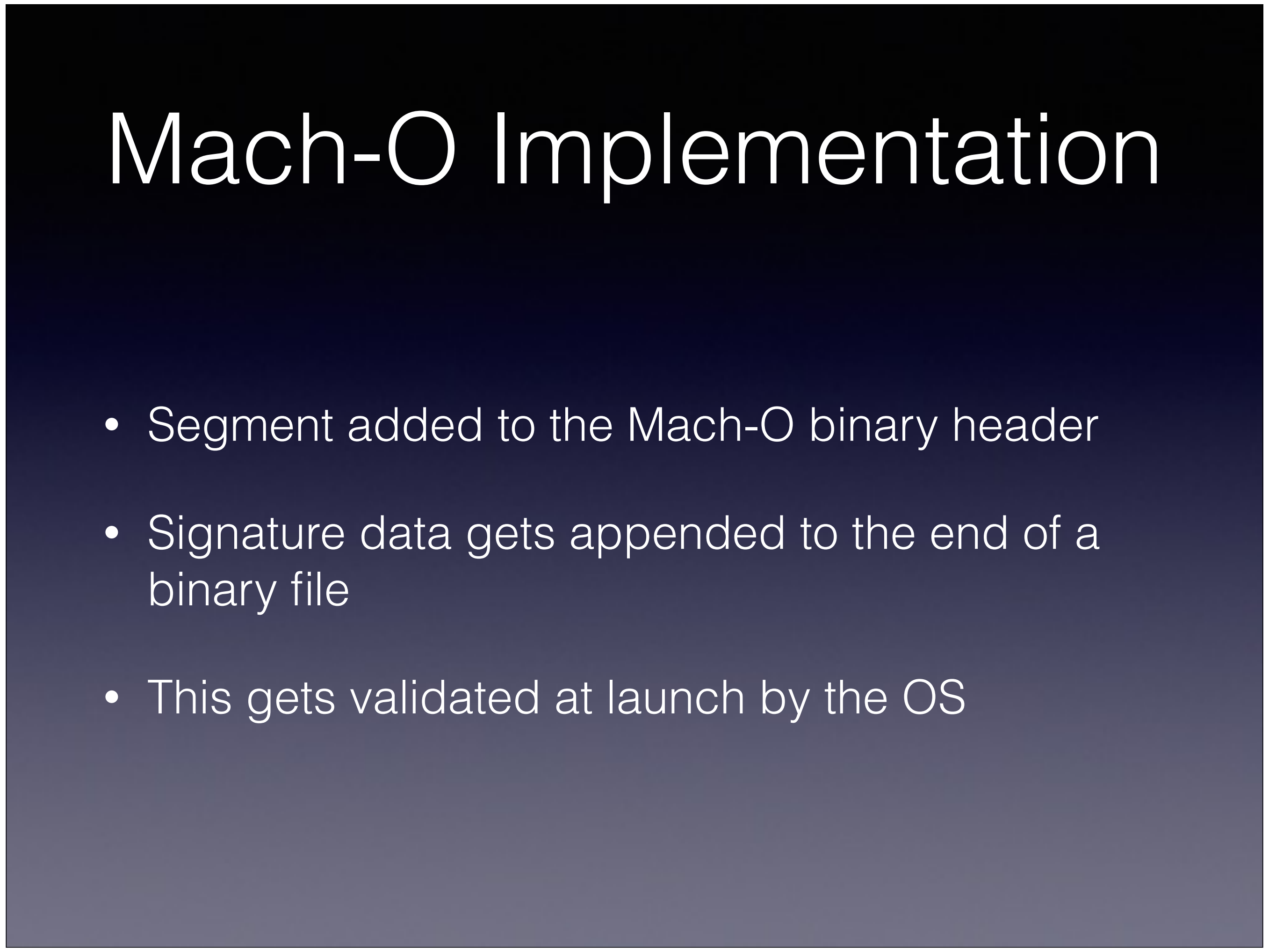
Blobs

- Encoded data that represents the signature data
- Entitlements, binary identifiers, signing identifiers, certificates, versioning, resource hashing, linked identifiers
- Unpacked at launch to validate the signature based on system policies and the seal being unbroken

"Blobs". This is the technical name of the encoded signature data. Blobs are a generic way to encode various types of data. Sandbox entitlements, app identifiers, signing identities, certificates, versioning, resource hashes, and identifiers of linked entities are all encoded into blobs. These are unpacked and evaluated when the binary is loaded on launched.

http://www.opensource.apple.com/source/libsecurity_codesigning/libsecurity_codesigning-33803/lib/sigblob.h

http://www.opensource.apple.com/source/libsecurity_codesigning/libsecurity_codesigning-33803/lib/sigblob.cpp



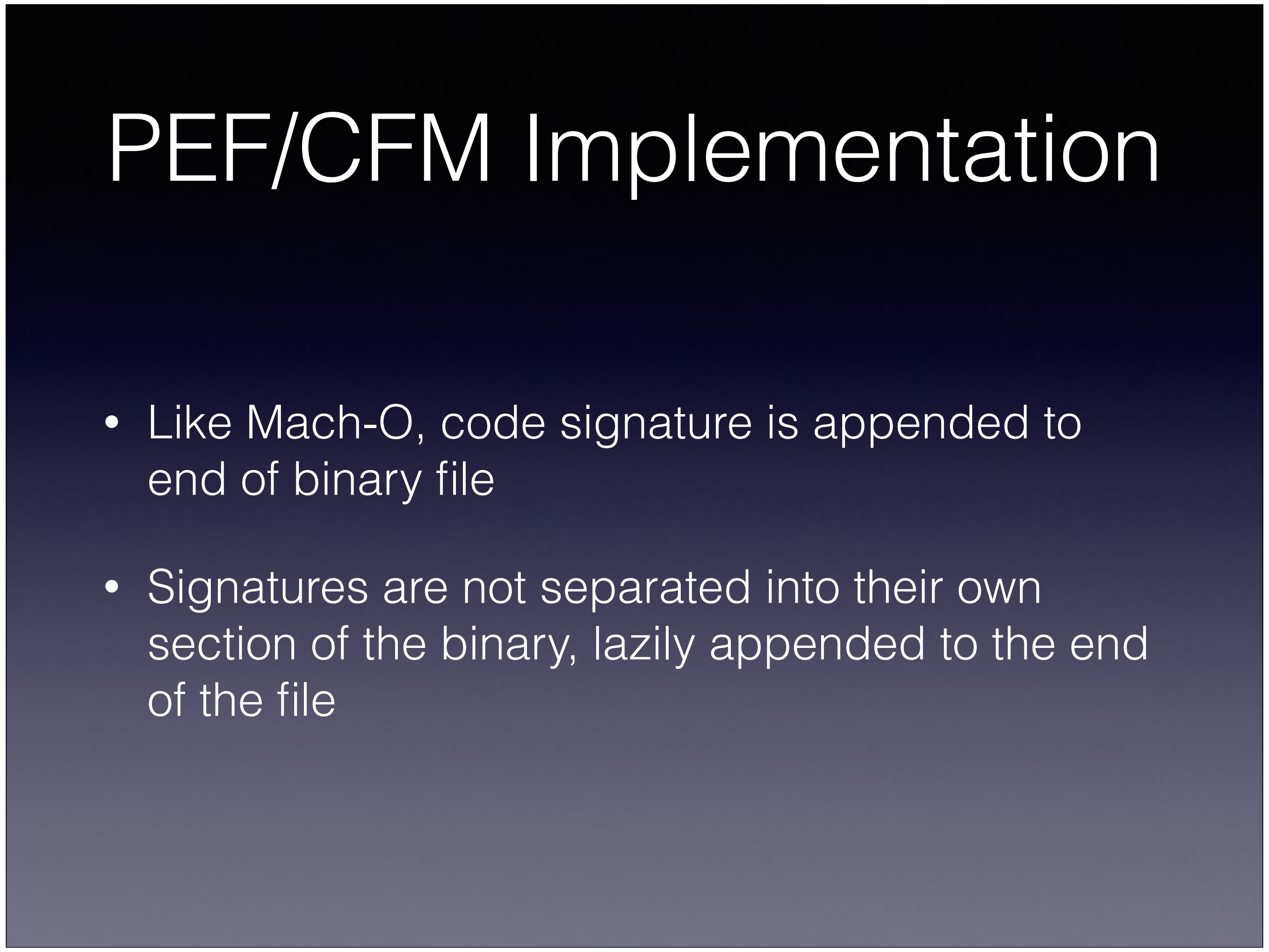
Mach-O Implementation

- Segment added to the Mach-O binary header
- Signature data gets appended to the end of a binary file
- This gets validated at launch by the OS

Mach-O binaries, this is the format of the executable binaries on modern OS X and all iOS systems. This is where all the compiled code lives. Without making this a presentation about how binaries are formatted: Binary files have a header that contains segments that dictate: type of binary, architecture supports, linked libraries, and runtime information. When a binary is code signed, a new segment is added to the header. This identifies where the signature blobs are stored. This is appended to the end of the binary file. This is a special segment that gets validated by the OS when the binary is loaded by the dynamic linker.

http://www.opensource.apple.com/source/libsecurity_codesigning/libsecurity_codesigning-33803/lib/machorep.h

http://www.opensource.apple.com/source/libsecurity_codesigning/libsecurity_codesigning-33803/lib/machorep.cpp



PEF/CFM Implementation

- Like Mach-O, code signature is appended to end of binary file
- Signatures are not separated into their own section of the binary, lazily appended to the end of the file

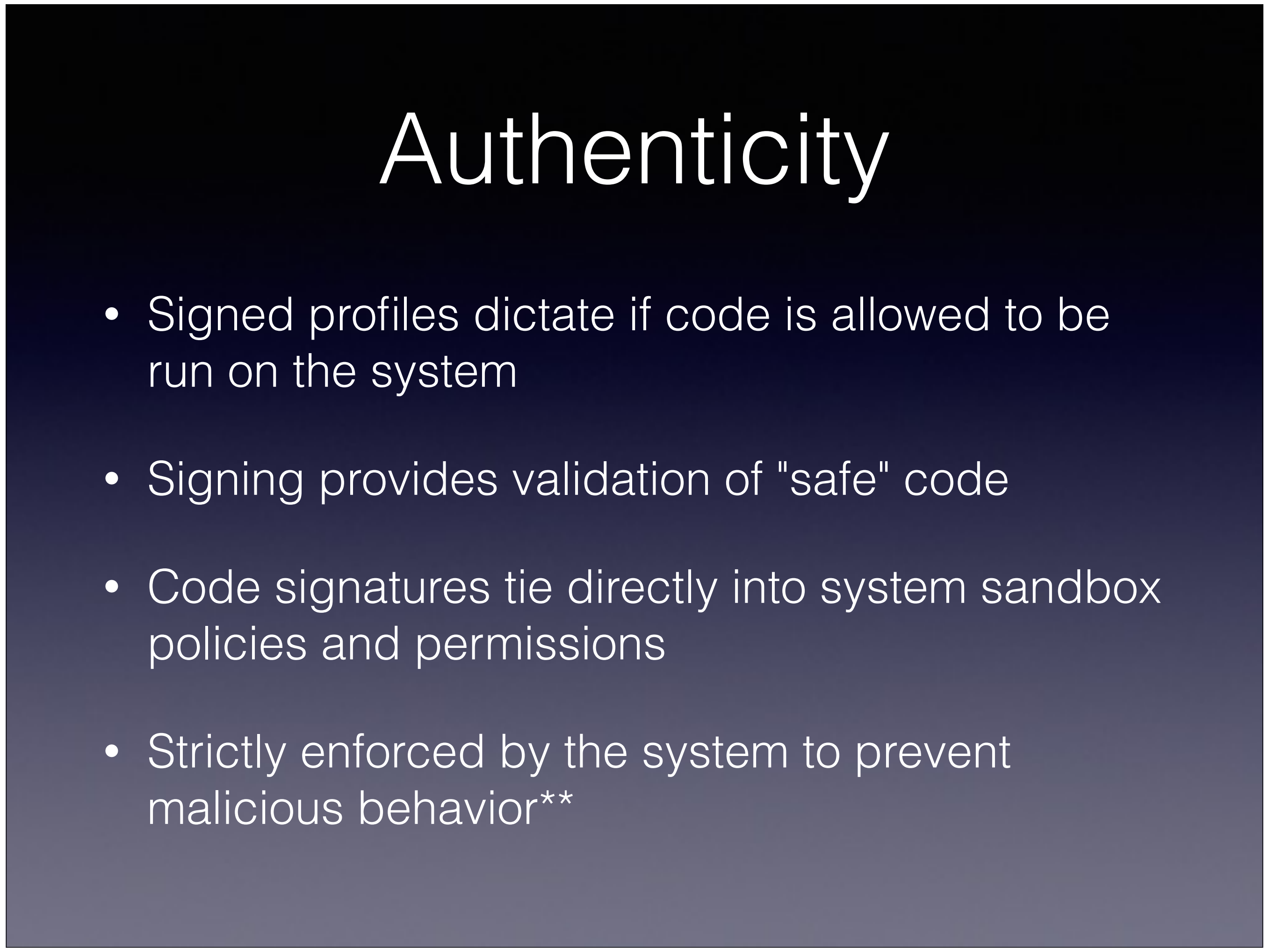
Portable Executable Format or Code Fragment Manager binaries are for PPC architectures. They are now defunct since 10.7 Lion, but there is still support for signing these. Like Mach-O binaries, the code signature is appended to the end of the binary file. However, this differs slightly because it blindly appends to the end of the file.

http://www.opensource.apple.com/source/libsecurity_codesigning/libsecurity_codesigning-33803/lib/cfmdiskrep.h

http://www.opensource.apple.com/source/libsecurity_codesigning/libsecurity_codesigning-33803/lib/cfmdiskrep.cpp

Why is this important?

So, why is all of this important?

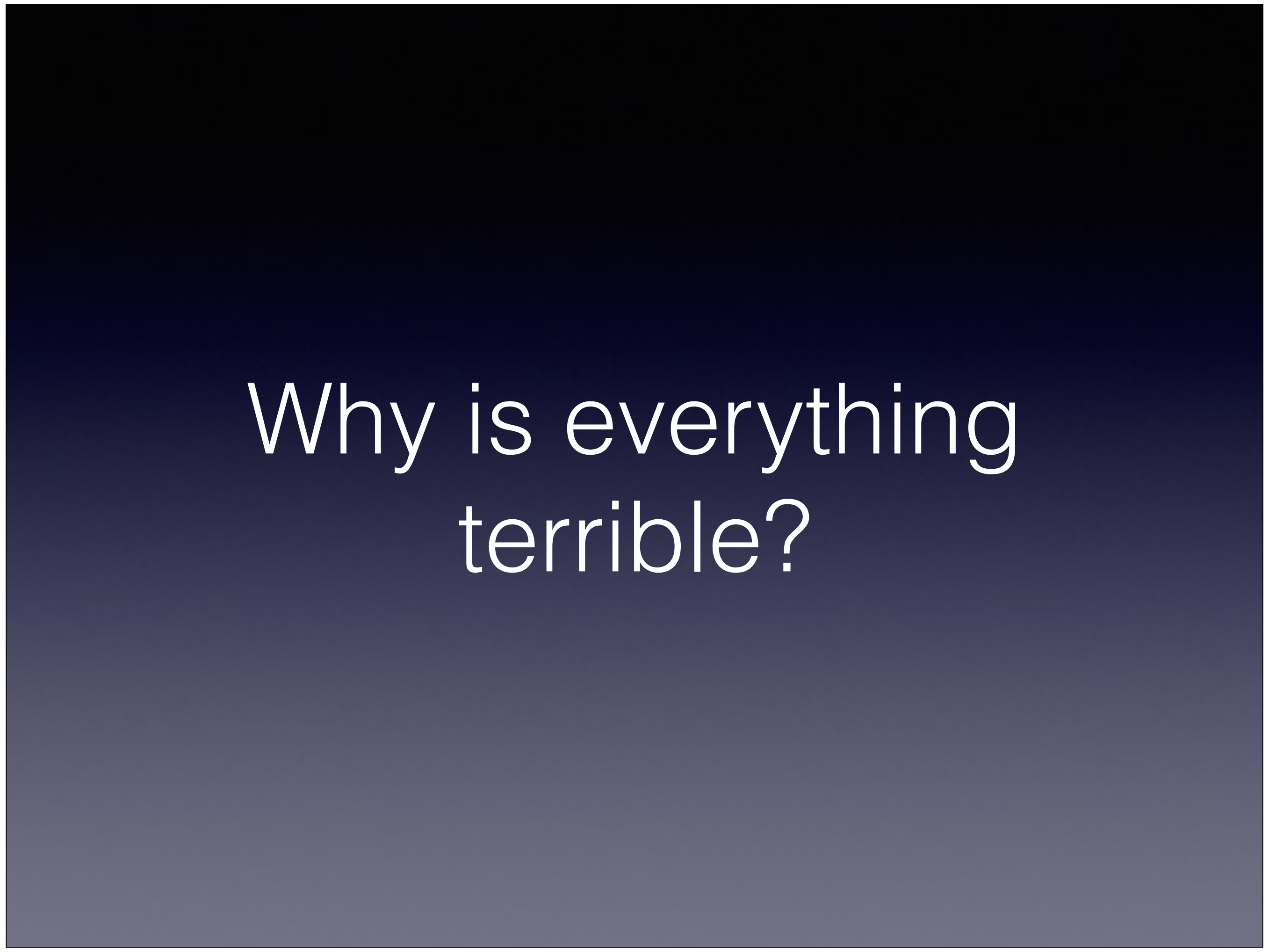


Authenticity

- Signed profiles dictate if code is allowed to be run on the system
- Signing provides validation of "safe" code
- Code signatures tie directly into system sandbox policies and permissions
- Strictly enforced by the system to prevent malicious behavior**

Basically, why is this important beyond the fact that Apple says we have to? Signed profiles enforce a system of where we can run code. The target device must be known. Signing adds a validation to code that it is from the original developer, and that it is unmodified. This allows software distributors to become trusted by consumers. Additionally, code signatures are directly used with the enforcement of application sandboxes and system permission policies. Your signed application can only ever access the services you have allowed it to, which means consumers can be better informed about what apps are doing on their systems. The “walled garden” approach that Apple takes gives some measure of defense against malicious behavior from developers.

**Note: Signing code doesn’t magically make code non-malicious, likewise unsigned code isn’t inherently malicious either. However unsigned code becomes a very easy target of third-party modifications.



Why is everything terrible?

So, why is everything terrible? The process behind the creation and validation of code signatures is complex. Since this process is presented as completely opaque there isn’t much recourse on how to fix problems that occur anywhere along the path. The majority of the issues come from the fragile state of cryptography and unique identification.