

PRACTICAL SECURITY

Rob Napier

robnapier.net/cocoaconf

Security is a tough topic. The best security system in the world is the one that you never see and causes nothing to happen. That makes it hard to know whether you're doing it right. It's tough to know what the right amount of security overhead is for a given project. That's a whole profession. It used to be my job before I was a Cocoa developer.

But most developers on most projects don't want to have to study all the ins-and-outs of security theory and practice. You just want to know what you need to do to make your system reasonably secure. Secure enough that it's not going to be embarrassing.

TODAY'S TOPICS

- Encrypting Network Traffic
- Disk Encryption
- Protecting Secrets
- Handling Passwords
- Correct Encryption

So today I'm going to cover some of the major security tools you need for everyday applications from business apps to games. When we're done here, you should have a small toolkit of concrete things you can implement, along with a better understanding of how to evaluate security implementations you encounter.

<build> First and foremost we'll discuss network encryption. This is one of the simplest things you can do, and I'll show you how to do it right.

<build> Then we'll discuss data protection and disk encryption for iOS.

<build> From there, we'll talk about data's gatekeeper, the secret.

<build> And the most common kind of secret, the password.

<build> Then we'll wrap up with the most technical section, how to use AES correctly.

ENCRYPT YOUR TRAFFIC

I'm going to start with the security tool that gives the greatest bang for the buck.

HTTPS

- Payload Encryption
- URL Encryption
- Cookie Encryption
- Server Authentication
- Session Hijack Prevention
- Replay Attack Prevention

If you don't do anything else to improve the security of your app, turn on HTTPS. Apple all but forces you to do this these days, but still I see a lot of blog posts on how to turn off HTTPS.

So many other problems either go away or are greatly reduced by doing this one thing. Done correctly, SSL solves a ton of problems. **<builds>...<builds>**

Even done incorrectly, HTTPS is still better than HTTP. It's hard to imagine a case where it's worse security, and that's not true of all security tools. The only caveat is you shouldn't assume that just because you have SSL, you don't have to worry about any other security problems. SSL doesn't fix weak user authentication. And it doesn't magically fix an insecure REST protocol. But it's a great example of a best practice. In one move it makes a lot of problems go away.

HTTPS requires a little more setup on the server side and can increase the load. Factor that into your design. There's just no excuse for not using it. But let me touch on a few other possible complaints.

BUT CERTS COST \$\$\$!

- Several < \$100/year. Some < \$50/year
- You can spend this much in billable time just researching
- Even snake oil helps

Good snake oil is better than commercial!

Server certs cost money. Yeah, that's usually true, but here are some points on that.

<build>First, they've gotten really cheap.

<build>Second, they don't cost much money compared to the time required to design and implement another security solution. Even a really expensive cert like Verisign's can cost less than the time it would take to research and implement your own solution.

<build>Third, even if you configure your connections to trust all certs, it's still more secure than to not use HTTPS. Yeah, it's possible someone will spoof DNS, but giving up authentication doesn't mean you have to give up encryption, too.

<build>And anyway, you can actually be more secure with a self-signed cert than with a commercial cert.

A LOT OF TRUST

You Expect...

- Verisign
- Network Solutions
- Thawte
- RSA
- Digital Signature Trust

But Also...

- Cisco, Apple, ...
- US, Japan, Baltimore, Belgium, Germany, Hong Kong, Netherlands, Slovakia, Taiwan, Turkey, ...
- Autoridad de Certificacion Firmaprofesional, Buypass, Disig, Certigna, Certinomis, E-Tuğra, Echoworx, QuoVadis, XRamp

“What?” you say? Isn’t a self-signed cert the worst thing ever? No. Remember, the Verisign root certificate is a “self-signed cert.” If you only trust your own cert, that’s more secure than trusting the 188 signing authorities that iOS trusts, plus every certificate they’ve signed. When you get a commercial cert, you need to trust yourself to keep your private key secure, and you have to trust the provider to keep their certificate secure. When you use a self-signed cert, you only have to trust yourself.

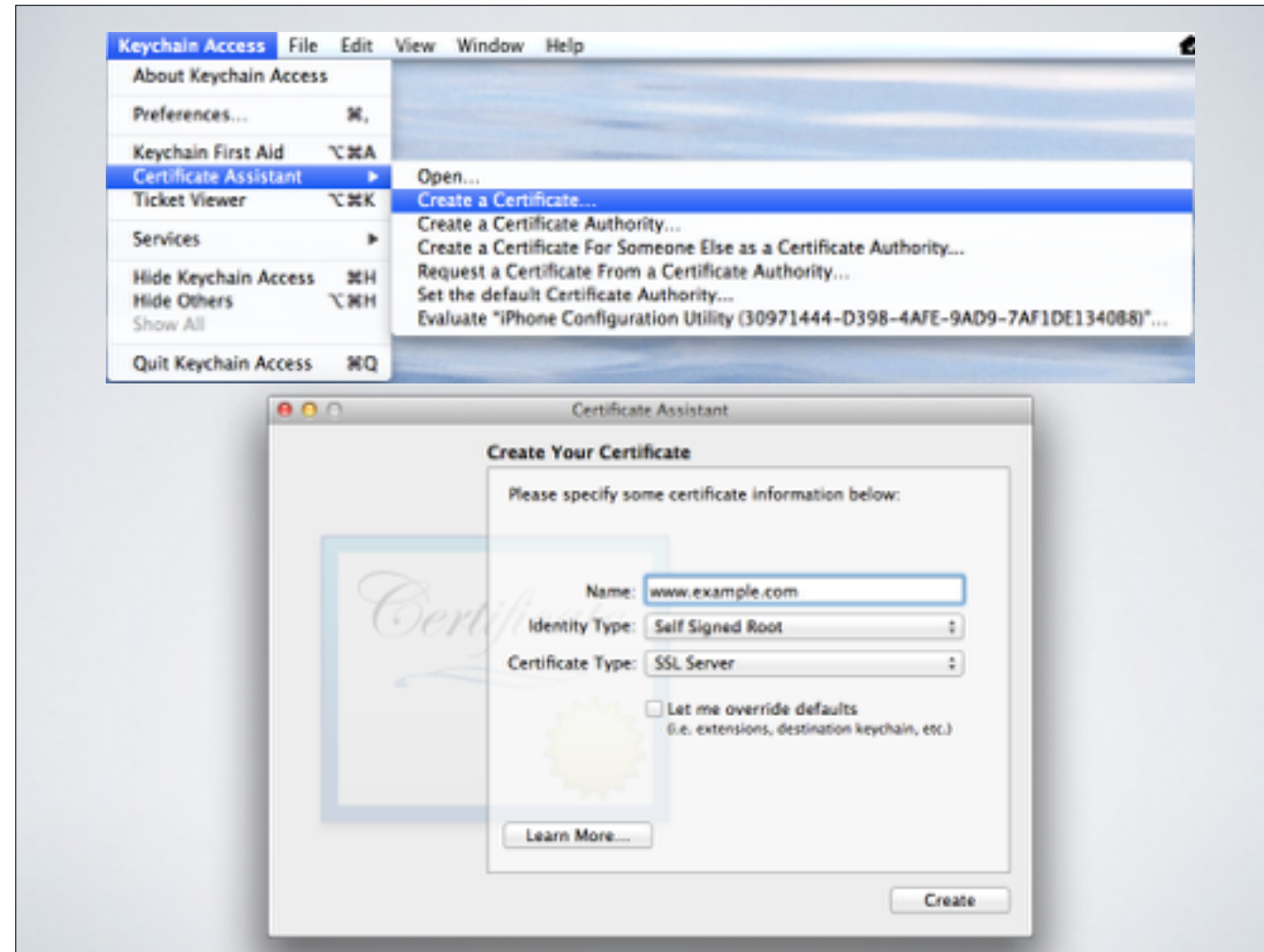
$T = \text{Trust required}$

$\forall T > 0 : T_{Self} + T_{Other} > T_{Self}$

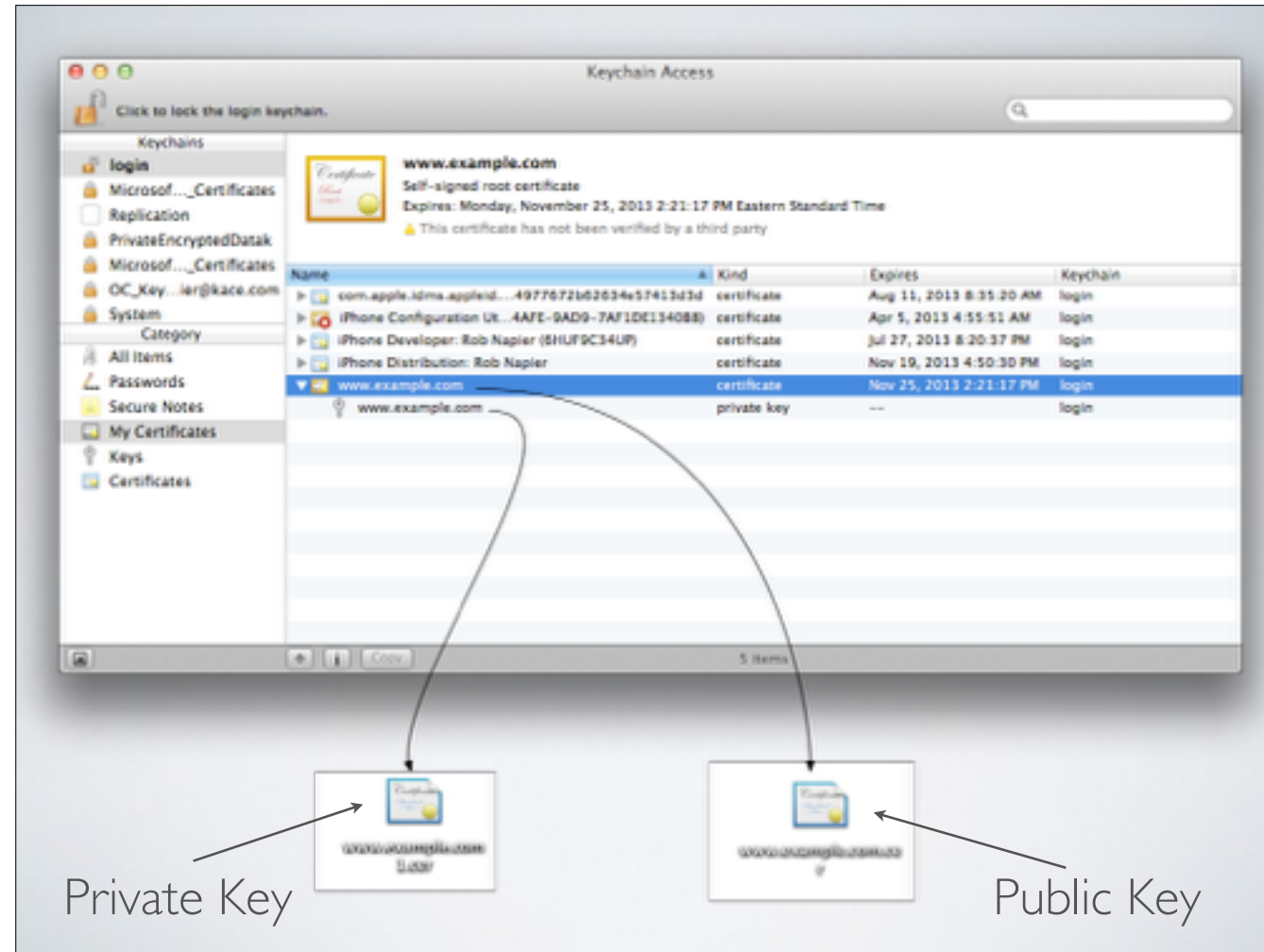
DON'T ARGUE WITH MATH

No matter how much you trust Verisign, it's always riskier to trust both yourself and Verisign than to just trust yourself.

I'm not saying you shouldn't get a commercial cert. You usually should. The few bucks is almost always worth the trouble. But if you want to only trust your own cert, here's how you do it with NSURLSession.



First, you create certificate. The name needs to exactly match your host name. So if you include “www” in your URL, the cert needs to include “www” and vice versa.



Export the public and private keys to separate .cer files. Store your private key on the server and keep it secret and secure. The only way you can trust this key is to know that you're the only one who has it. This is exactly what you would do if your certificate were signed by Verisign.

Your public key goes into your resource bundle. You can rename the files if you like.

```

final class Connection: NSObject, NSURLSessionDelegate {

    private lazy var anchors: [SecCertificate] = {
        let path = NSBundle.mainBundle().pathForResource("www.example.com", ofType: "cer")!
        let certData = NSData(contentsOfFile: path)!
        let certificate = SecCertificateCreateWithData(nil, certData)!
        return [certificate]
    }()

// ...

func URLSession(session: NSURLSession,
    didReceiveChallenge challenge: NSURLAuthenticationChallenge,
        completionHandler: (NSURLSessionAuthChallengeDisposition, NSURLCredential?) -> Void)

```

In your code, you'll create a certificate object using `SecCertificateCreateWithData` and put it in an array for use later.

<build>By implementing the delegate method `URLSession:didReceiveChallenge:completionHandler:`, you take control of whether a certificate is accepted. By the end of the method, you need to call the completion handler with one of various dispositions.

AUTHENTICATION RESPONSES

`.UseCredential` (*accept cert*)

`.CancelAuthenticationChallenge` (*reject cert*)

`.PerformDefaultHandling`

`.RejectProtectionSpace`

The simplest authentication responses are `useCredential`, which means the certificate is ok, and `cancelAuthenticationChallenge`, which means its not.

In our example, the question we want to answer is whether the certificate is signed by our trusted anchor certificate. The tool we use to answer that question is a trust object.

```

func URLSession(session: NSURLSession,
didReceiveChallenge challenge: NSURLAuthenticationChallenge,
completionHandler: (NSURLSessionAuthChallengeDisposition, NSURLCredential?) -> Void) {

    let protectionSpace = challenge.protectionSpace

    if (protectionSpace.authenticationMethod == NSURLAuthenticationMethodServerTrust) {

        if let trust = protectionSpace.serverTrust {

            SecTrustSetAnchorCertificates(trust, anchors)
            SecTrustSetAnchorCertificatesOnly(trust, true)

            var result = SecTrustResultType(kSecTrustResultInvalid)
            let status = SecTrustEvaluate(trust, &result)

            if status == errSecSuccess {
                switch Int(result) {
                    case kSecTrustResultProceed, kSecTrustResultUnspecified:
                        completionHandler(.UseCredential, NSURLCredential(trust: trust))
                        return

                    default:
                        print("Could not verify certificate: \(result)")
                }
            }
        }
    }

    // Something failed. Cancel
    completionHandler(.CancelAuthenticationChallenge, nil)
}

```

So let's go back to our delegate method. NSURLSession passes us a pre-configured trust object as part of the challenge.

<build> We add our anchors, which is just that one certificate we read from the bundle, and then configure the trust object to only use the anchors we gave it. By default it'll also use the built-in list of anchors.

<build> We evaluate the trust using SecTrustEvaluate.

<build> And if we get a good result, we call pass .UseCredential to the completion handler. Otherwise we pass .CancelAuthenticationChannel.

"Proceed" generally means that the user has explicitly accepted this cert. You usually won't get that, especially on iOS. Most of the time you'll get "Unspecified", which means that the cert is trusted according to default policy, but the user hasn't explicitly accepted it. All the other responses are varying levels of untrusted.

I still recommend getting a commercial cert most of the time, because it's usually less work. But either way, there's no excuse for not using HTTPS.

ENCRYPT YOUR TRAFFIC

- Use HTTPS for all sensitive traffic
- Validate that the cert is trusted
- Your cert can be as good as commercial
- But commercial is probably easier

And that's really all there is to it.

<build> Use HTTPS for all sensitive traffic. How do you know it's sensitive? If it would be ok to publish it on the front page of the New York Times, along with your customer's name, then it's not sensitive. Otherwise, it is.

<build> Validate that the cert is trusted. Don't try to skip the verification step.

<build> But it's fine to only trust your own certificate, using the techniques I just showed you.

<build> That said, it's usually easier to just get a commercial cert. They're not that expensive, and you don't have to hand-evaluate them.

Questions?

DISK ENCRYPTION

Now that you've downloaded this sensitive data, let's look at how you keep it safe on the device. iOS provides built-in data encryption, and in many cases you should be using it. I'm not going to go into a big analysis of how strong the data protection is. The answer is, it's good enough. Yes, a short PIN, which is the norm on iPhone, can be brute-forced in about 20 minutes. It doesn't matter. Like locking your front-door, device encryption is one of those easy things that you do because it's easy. And it gives control to the user. If they want long passwords, and on an iPad long passwords are pretty practical, then the user is free to improve their security.

Quick note on fingerprints.

HOW EASY IS IT?

```
try data.writeFile(name,  
                  options: [])
```

And it's so easy. How easy? As long as you don't need to read or write files while you're in the background, it's a single change to one line of code.

HOW EASY IS IT?

```
try data.writeFile(name,  
                  options: .DataWritingFileProtectionComplete)
```

If this is still too hard for you, wait for the end...

That's it. You're done. How can you seriously say to your customer that this was too hard? If you have an existing file, or if you need to write with something other than NSData, you can use NSFileManager like this.

USING FILEMANAGER

```
extension NSFileManager {  
    func protectFileAtPath(path: String) throws {  
        try setAttributes([NSFileProtectionKey: NSFileProtectionComplete],  
                           ofItemAtPath: path)  
    }  
}
```

NSDataWritingFileProtectionComplete → NSFileProtectionComplete

NSFileManager setAttributes:ofItemAtPath.

Note that NSData and NSFileManager use different constants,

<build>but they mean the same things. FileProtectionComplete just means protected whenever the device is locked. This is the highest level of protection and usually the one you want.

For many apps, that's it. But for apps that do need to read and write files while they're in the background, let's go a little deeper. Let's say you're uploading or downloading a file, so you need access to that file even if the device happens to lock. That's not uncommon, but it's not hard to deal with.

iOS ENCRYPTION

Data Protection

Device Encryption

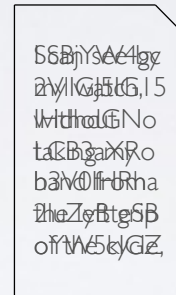
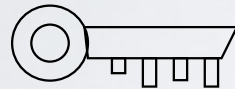
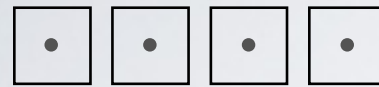
First, a quick intro to encryption in iOS. There are two levels of encryption: device encryption and data protection.

<build>Device encryption is completely transparent to you, and links a given CPU to its flash storage. It's what's used to enable remote wipes and the fast “erase all contents and settings” option. It's completely managed by the device and you have no control over it, so we're not going to dive into that today.

<build>Everything we're talking about today is called data protection or file protection, and it's per-file encryption.

What I'm about to describe is a gross over-simplification of the actual system, and in a couple of places is intentionally wrong, but it captures the main points while dodging some of the complexities of key wrappers, elliptic curve cryptography, and other technical rat-holes of the real implementation. At the end I'll provide a link to the full, correct explanation of how Data Protection works. Consider the following to be conceptual.

DATA PROTECTION (SIMPLIFIED)



NSFileProtectionComplete

<build> Say you have a document you want to protect.

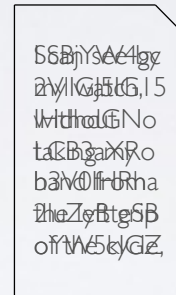
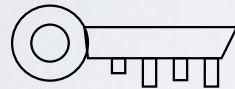
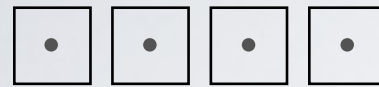
<build> You mark it with NSFileProtectionComplete.

<build> The system will derive a key for that file based on the user's PIN.

<build> The file will be encrypted immediately, but the key will be held in memory so your application can still access the file.

<build> When the device is locked, then within 10 seconds the key will be scrubbed from memory and your app won't be able to access the file any more.

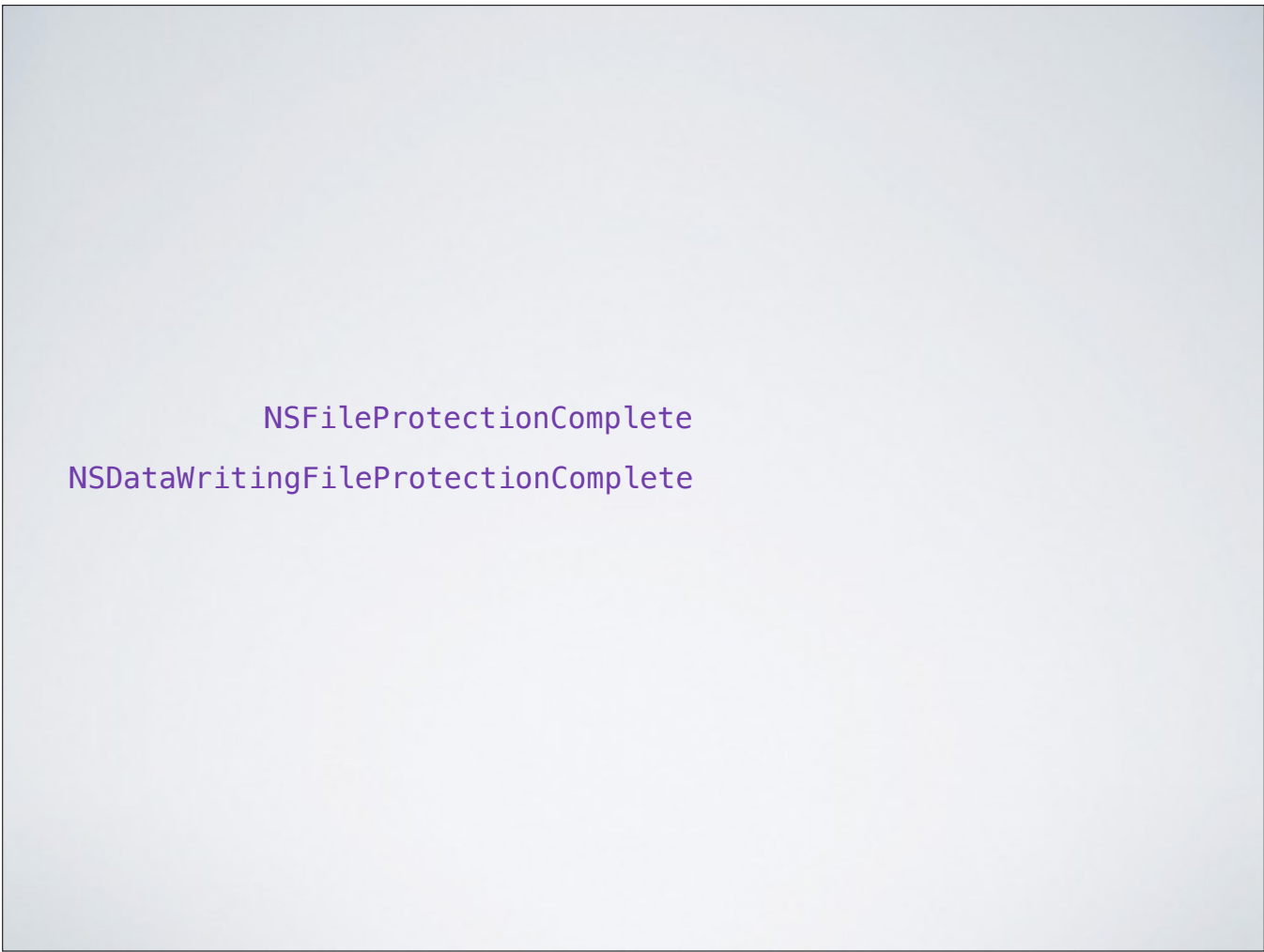
DATA PROTECTION (SIMPLIFIED)



NSFileProtectionComplete


Then when the user unlocks the device, the key will become available.

So very shortly after the device is locked, any file marked as ProtectionComplete can no longer be accessed. So what if we need to keep downloading something and writing that to a cache? Well, first ask whether you really need to keep downloading while the device is locked. There are good solutions, but they are all less secure than ProtectionComplete. If your information is very sensitive, it may be worth suspending uploads and downloads while the device is locked.



NSFileProtectionComplete
NSDataWritingFileProtectionComplete

But in most cases, the best solution is to use the option `CompleteUnlessOpen`.




```
NSFileProtectionCompleteUnlessOpen  
NSDataWritingFileProtectionCompleteUnlessOpen
```

This means that if the file is open when the device is locked, then the decryption key is kept around until the file is closed. This is a generally a good level of protection for files you're downloading or uploading or for log files. It's also the highest protection level you can use for files created while the device is locked.

CompleteUnlessOpen only lets you continue accessing files that were open when the device is locked, or files you create while the device is locked. Most of the time that's all you need. But let's say you have a location app that runs in the background all the time. And it has a database that is usually closed. When some event happens, even if the device is locked, you need to open that database and perform some action. CompleteUnlessOpen won't help here, because the file wasn't open when the device went into the background.

My first recommendation is to keep the database open so you can use CompleteUnlessOpen. But maybe that causes some other problem, or you have to close the file at various points to copy it or whatever. There is one last protection level. It doesn't give a lot of protection, but it does give some, and is useful for this case.



```
NSFileProtectionCompleteUntilFirstUserAuthentication  
NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication
```

It's called `CompleteUntilFirstUserAuthentication`. When a file is marked this way, it is only protected from the time the device is booted until the first time the user unlocks it. From that point on, even if the user locks the device again, the file is unprotected. Only rebooting will protect the file. But this does protect the data against attacks that require a reboot, so it's still useful if you have no other option.

```

extension NSFileManager {
    func upgradeFilesInDirectory(dir: String) throws {
        let desiredProtection = NSFileProtectionComplete

        let url = NSURL(fileURLWithPath: dir)

        guard let dirEnum =
            enumeratorAtURL(url,
                            includingPropertiesForKeys: [NSFileProtectionKey],
                            options: [], errorHandler: nil) else {
            throw ...
        }

        var lastError: ErrorType? = nil
        var resourceValue: AnyObject? = nil

        while let element = dirEnum.nextObject() as? NSURL {
            do {
                try element.getResourceValue(&resourceValue,
                                            forKey: NSFileProtectionKey)

                let currentProtection = resourceValue as? String ?? ""
                if currentProtection != desiredProtection {
                    try element.setResourceValue(desiredProtection,
                                                forKey: NSFileProtectionKey)
                }
            } catch { lastError = error }
        }

        if let lastError = lastError { throw lastError }
    }
}

```

You can change data protection levels at any time using NSFileManager, even if the file is currently open.

So for a file you're downloading, or a file you create while the device is locked, you could initially set it to CompleteUnlessOpen, then upgrade it to Complete once it's finished writing. During application startup you can also look at all your documents and upgrade any that have the wrong setting.

UIApplicationDelegate Methods

```
func applicationProtectedDataWillBecomeUnavailable(application: UIApplication)
func applicationProtectedDataDidBecomeAvailable(application: UIApplication)
```

UIApplication Notifications

```
public let UIApplicationProtectedDataWillBecomeUnavailable: String
public let UIApplicationProtectedDataDidBecomeAvailable: String
```

Note the missing “Notification”
rdar://13387084

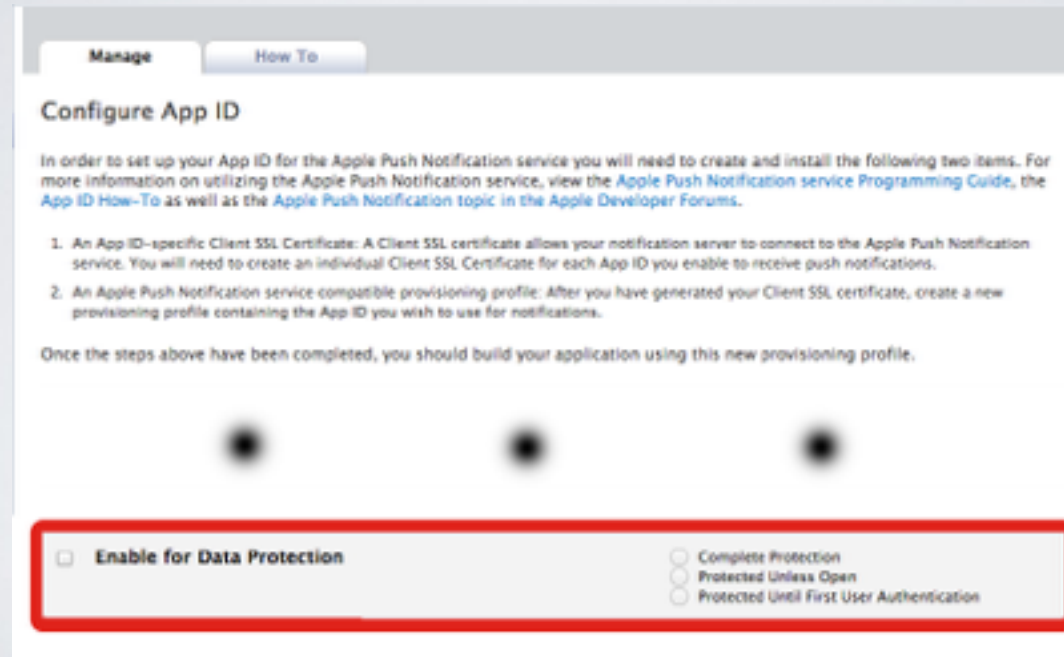
UIApplication Methods

```
public var protectedDataAvailable: Bool { get }
```

You can also find out when data is going to become available or unavailable by using the application delegate methods, or the notifications, or by querying the application’s `isProtectedDataAvailable` property. That lets you make decisions about what files to write or when to change a file’s protection level.

But really, most of the time you don’t need all of this. You just need to set `ProtectionComplete` and you’re done.

HOW EASY? (PART 2)



And that's just one extra option every time you write a file. But could it be easier? I mean, it does mean you need to mess with this every time you create a new file. Wouldn't it be nice if you could set this once and be done for the whole app? Sure it would.

<build>Go to the provisioning portal. Configure your App ID and select a default protection level. Download your new provisioning profile, and it'll automatically apply the right protection for every new file you create.

DATA PROTECTION

https://www.apple.com/business/docs/iOS_Security_Guide.pdf

- Turn it on automatically in your App ID
- For foreground-only programs, just use Complete
- For background file access, try to use CompleteUnlessOpen
- If all else fails, use CompleteUntilFirstUserAuthentication
- Upgrade to Complete as soon as you can
- Use NSData to create protected files rather than upgrading them with NSFileManager

Here's that link I promised earlier. It provides a good overview of the technical details, targeted at corporate Infosec departments.

You've seen all the ways to implement it in your product.

<build> Ideally, just go to iTunes Connect and set the default for your app.

<build> If your program does all of its work in the foreground, you use Complete.

<build> If you read and write files in the background, you may have to think about it a little more, but it's still pretty easy, especially if the file stays open the whole time you need access.

<build> And finally, whenever possible, use NSData's write methods to create your protected files. That way they're protected from the very beginning.

Questions?

PROTECTING SECRETS WITH KEYCHAIN

Data protection protects data. Keychain protects secrets. A “secret” in this context is a small amount of information used to access other data. It’s a cryptographic term, and in a cryptographic system a “secret” is the only part that’s kept hidden. The most common secrets are passwords and private keys. And the best way to protect them is Keychain.

THE THING ABOUT KEYCHAIN...

- Generally the best tool for the job, but...
- A pain to use
- Complicated
- Slow

Many applications need to store passwords, so they should be using Keychain.

<build> Unfortunately, it's a bit of a pain to use directly.

<build> It has a really complicated API with lots of overlapping functionality coupled with lots of missing functionality, that makes it hard to know how to use it correctly.

<build> And it's very slow.

- Introduction to Keychain
- The Wrappers
- Access Groups

So I'm going to give a quick, very high level overview of what you need to know about Keychain, without diving into the rat's nest of details, <buid>then swing through the wrappers you should be using that hide most of that from you anyway, <buid>and then focus on the piece that most people don't know how to use well, access groups.

KEYCHAIN ITEMS

- Generic Password - Account, Service, Password
- Internet Password - Designed for URLs
- Certificate - Mostly used for validating servers
- Key - Public-key encryption
- Identity - A cert plus a key, generally for client certificates

First, there are several kinds of Keychain items:

- **<build>** Generic Password
- Internet Password
- Certificate
- Key
- Identity

Almost always what you want are generic passwords, and that's what all the wrappers use. Internet passwords are designed specifically for web browsers. Certificates, keys, and identities are used if you're managing client certificates or public-key crypto. I'm not going to go into those today. But mostly what you really want are simple, generic passwords.

KEYCHAIN ITEM

Attributes

kSecAttrAccount

kSecAttrService

kSecAttrAccessGroup

kSecAttrCreationDate

kSecAttrModificationDate

kSecAttrDescription

kSecAttrComment

kSecAttrCreator

kSecAttrLabel

kSecAttrGeneric

...

Value

The Password

All the types have a bunch of attributes, and then a piece of data called the “value.” The value is the only part that’s encrypted in the Keychain. The attributes are semi-public and are what you can search for. For a generic password, the most important attributes are the account and the service. You search for these to find the correct record, and the encrypted value for that record is the password. So if you’re using Keychain directly, remember that only the value is really protected. Everything else is more-or-less public information, particularly on Mac.

ADD VERSUS UPDATE

- Adding an item that already exists is an error
- Updating an item that doesn't exist is an error
- Generally before adding, you need to query to check
- Wrappers generally do this automatically

The next important thing is that Keychain strongly separates add versus update. Adding an item that already exists is an error. Updating an item that doesn't exist is an error. So you generally need to search for the old record, and then decide whether to add or update. Luckily most of the wrappers handle that for you.



SLOW, SLOW, SLOW ...

Finally, Keychain is slow. Incredibly slow. It's not so slow that you can't access it on the main thread, but don't go querying it repeatedly. If your wrapper doesn't cache values, you should keep that in mind.

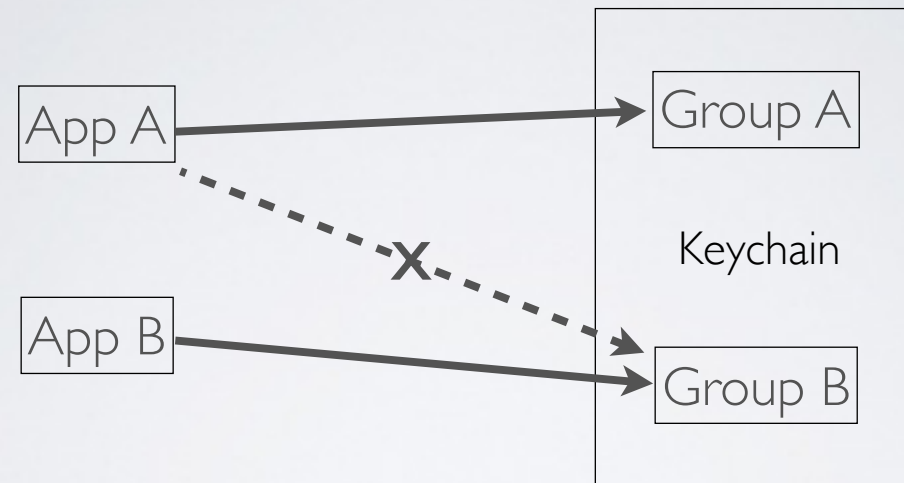
WRAPPERS

- SGKeychain (secondgear/SGKeychain)
- SwiftKeychainWrapper (jrendel/SwiftKeychainWrapper)

OK, I've mentioned wrappers several times now, and so let's get to that. There are a lot of wrappers out there. I don't really love any of them. But there are two I do recommend: SGKeychain in Objective-C, and SwiftKeychainWrapper in Swift. I like these wrappers mostly because they support access groups. We'll be discussing access groups much more in a moment and why they can be important.

That said, there are a ton of wrappers and they're mostly all fine. If you're using one already and like it, I certainly wouldn't switch. They're all pretty equivalent security-wise. All the wrappers have easy-to-read API docs, so I'm not going to dwell on that here. You guys can figure out how to use them. But I do recommend using one unless you have a very special problem.

ACCESS GROUPS



What I will discuss is access groups. This is a really powerful feature on iOS that requires a few tricks to use well. So what are they?

<build> Well, on iOS each app effectively has its own private keychain. This is implemented by tagging the entry with an access group.

<build> If you create an account in one app, other apps can't get access to those credentials. That's a real problem if you have a collection of apps that are meant to work together. Shared access groups solve this.

ACCESS GROUP FORMAT

`<app-ID>.<reverse-DNS>.<identifier>`

`E9G2DXXXXX.net.robnapier.shared`

An access group is really just a string. If two apps pass the same access group to Keychain, they'll select the same records, as long as they both have permission for that group.

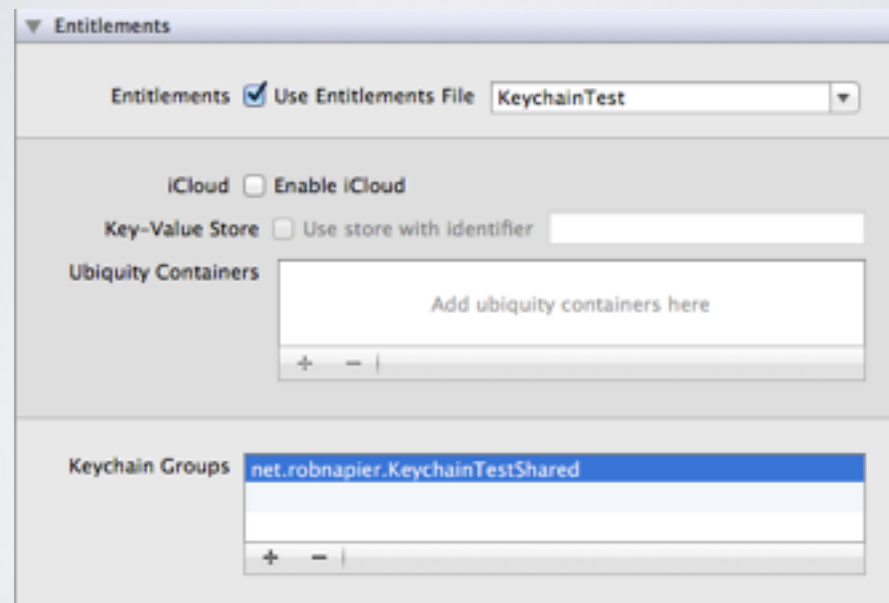
The access group has to begin with the App ID you signed the app with. This prevents you from reading account information from other people's apps, but if you have a group of apps, and sign them with the same App ID, they can work together. Just pass the same access group.

<build> Generally the access group has a format like this.

The first part is your app id. The reverse DNS is the same as you would use in your bundle ID. And the identifier is whatever you want.

<build> You can use that to create separate shared keychains between your apps, though there's seldom a reason to do that. But again, as long as it starts with your app id, the rest is up to you.

ENTITLEMENTS



But each of your apps needs to request permission to use this access group. This is handled as an entitlement.

To set this up, you need to add entitlements to your app. Go to your project settings, select the target, and then select the Summary pane. Select “Use Entitlements File” which will create an entitlements plist. Then in Keychain Groups, add a new group. Don’t include the app id. Xcode will quietly insert it and a dot onto the front of whatever you type here. If you look in the entitlements plist, you’ll see it as `AppIdentifierPrefix`, but unfortunately it won’t show it to you in the UI.



Keychain Sharing

☐ OFF

Allows this application to share passwords from its keychain with other applications made by your team.

- Add the "Keychain Sharing" entitlement to your entitlements file
- Add the "Keychain Sharing" entitlement to your App ID

EXPLICIT ACCESS GROUPS

- If you're not explicit, it may work, but it may create duplicates
- I recommend requesting explicit access groups

Here's where things get a little tricky. As long as all your apps list all the same access groups in their entitlements, they'll be able to share keychain items without writing any additional code. But if you're not careful, you can wind up with duplicate entries, which can be very hard to debug.

<build> It's legal to have two items with the same user account and service but different access groups. This is actually really common, since unrelated apps may have the same account information. When you don't pass an access group, that means "give me the first one you find from my entitlements that I have access to." If you have entitlements for multiple access groups, you can wind up with a messy situation.

<build> Because of that, if I plan to share information between apps, I usually prefer to explicitly put the records into an access group, and then limit my searches to that one group rather than all groups I have entitlements for. This makes sure I don't wind up with inconsistent records.


```
KeychainWrapper.accessGroup = "..."
```

To do that, you need a wrapper that allows you to pass an access group. You then also have to know the full name of your access group. That includes your app identifier, which isn't easily accessible at runtime. You could hard-code your app identifier, but that's fragile.

```

// Based on Objective-C code from David H
// http://stackoverflow.com/q/11726672/97337

extension UIApplication {
    func applicationIdentifier() throws -> String {
        let query = [
            kSecClass as String : kSecClassGenericPassword,
            kSecAttrAccount as String: "applicationIdentifierQuery",
            kSecAttrService as String: "",
            kSecReturnAttributes as String: true,
        ]

        var result: AnyObject? = nil
        var status = SecItemCopyMatching(query, &result)
        if status == errSecItemNotFound {
            status = SecItemAdd(query, &result)
        }
        precondition(status == errSecSuccess,
            "Could not read or write to keychain: \(status)")

        guard let
            resultDictionary = result as? [String: AnyObject],
            accessGroup = resultDictionary[kSecAttrAccessGroup as String],
            identifier = accessGroup.componentsSeparatedByString(".").first
        else {
            preconditionFailure("Found garbage in keychain: \(result)")
        }

        return identifier
    }
}

```

The better solution is to get it at run time. You can do that by creating a random keychain item, and then seeing what access group it was put into. You can then strip off the first part and that's your app ID. It's ugly but it works. I'll have a link to this code on my site.

With that in place, using it is pretty simple.

```
let sharedKeychainIdentifier = "com.example.mygreatappsuite"  
let applicationIdentifier = UIApplication.sharedApplication().applicationIdentifier()  
KeychainWrapper.accessGroup = "\(applicationIdentifier).\(sharedKeychainIdentifier)"
```

Fetch the application ID, also called the bundle seed ID, and append whatever you want your shared identifier to be. Then pass that as your access group from all your applications, and they'll share their keychain items.

KEYCHAIN

- Use the Generic Password type
- Use a wrapper such as SwiftKeychainWrapper or SGKeychain
- Use explicit access groups when sharing

And that's most of the really common things you need to know about Keychain, at least related to passwords.

- **<build>** You want the Generic Password type unless you have a special need
- **<build>** Use a wrapper. I recommend SwiftKeychainWrapper or SGKeychain.
- **<build>** If you are going to share between apps, use an explicit access group.

Questions?

HANDLING PASSWORDS

Did I mention LinkedIn?

The recent LinkedIn password fiasco is a great object lesson on the importance of good password handling, and their mistakes weren't the worst I've seen. No matter how trivial the information on your site, you must always treat user passwords with the utmost care. Users reuse passwords all the time, and if your site is hacked and the bad guys use that to break into your customers' bank accounts, no one is going to care when you say "but they shouldn't have reused passwords." They do and it's up to you to deal with it.

HASHING

Password

Hash

S3kr3t! → d39ee8e54ac7...

The first step to proper password management is hashing. You should never, ever store actual passwords on your server. You should avoid even sending them to your sever if you can help it.

<build> You should only accept and store cryptographic hashes. What's special about cryptographic hashes is that it's impractical for an attacker to find data that creates a given hash. Regular hash functions, like NSString's hash method, don't promise that. In fact, it's very easy to find NSStrings with the same hash.

CHOOSE YOUR HASH

- SHA-2 – Best commonly available
 - Pretty widely supported
 - No-known attacks
 - Also called SHA-224, -256, -384, and -512
- SHA-1 – Acceptable for most uses
 - Widely supported
 - Has known attacks, but not easy attacks
- SHA-3 – Someday
 - Can be faster than SHA-2
 - Few implementations

The best choice right now is SHA-2. When you see SHA-224, -256, -384, and -512, these are all different sizes of SHA-2.

<build> But SHA-1 really is fine for most purposes, and is more widely supported. There are some attacks against it, but for many uses the attacks don't actually apply. That said, attacks only ever get better. They never get worse. So when you have the opportunity, you should move from SHA-1 to SHA-2, and you should definitely use SHA-2 for any new code.

<build> SHA-3 was just selected last month and there aren't a lot of implementations out there yet. iOS doesn't support it, and there's no real reason for you to pursue it today for almost any application you're likely to write. Eventually there will be attacks against SHA-2, and then we'll all be glad that SHA-3 is already available, but today, SHA-2 is looking very good.

There's no reason to be using hashes other than the SHA series today, so I'm not going to talk about them. (By the way, SHA, like AES, doesn't refer to a specific algorithm. SHA and AES are government selection processes. So while the math behind SHA-1 and SHA-2 are related, SHA-3 is very different.)

WHAT WENT WRONG?

d39ee8e54ac7f653 | | 676d0cb92ec2483 | 9f7d27

Passw0rd	2acf37c868c0dd805 3a4efa9ab4b4444a4d5c94
MyPass	b97698a2b0bf77a3e3 e089ac5d43e96a8c34 32
S3kr3t!	d39ee8e54ac7f653 676d0cb92ec2483 9f7d27
...	...

All that said, cryptographic hashes can still go wrong and be cracked. This happened to LinkedIn a few years ago. Let's say I'm an attacker.

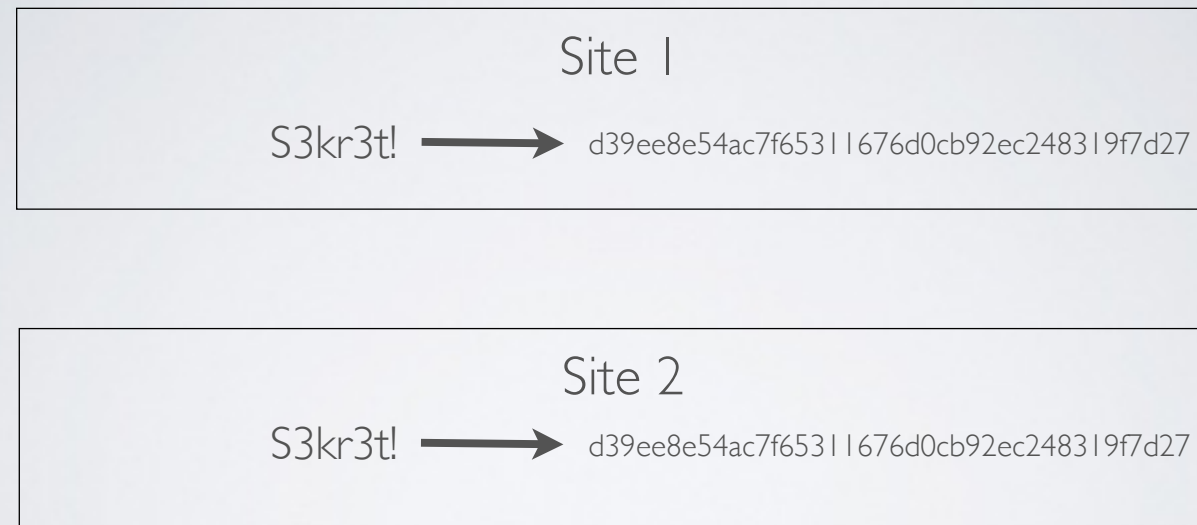
<build> Just given this SHA-1, it's practically impossible for me to figure out a string that will generate it.

<build> But, on the other hand, it's very easy for me to guess a lot of passwords and calculate their hashes.

<build> When I steal a list of password hashes, I can then see if any of hashes I calculated are in the list.

I may not be able to break a specific account this way, but I'll be able to break a lot of accounts nonetheless.

SALTING



The real problem is that user passwords are very un-random. We address this with two techniques that work together: salting and stretching. Salting just means adding something unique to the password so that two uses of the same password don't generate the same hash.

SALTING

Site 1

XXX:S3kr3t! → 48fc6c1a82882c0084185c3e6f317d6cdabfbc88

Site 2

YYY:S3kr3t! → 7802cd6060f13349da21652e4bc8cd31e3058842

The best salts are totally random bytes, but that's sometimes inconvenient to implement for password authentication systems, so I'm going to discuss how to build a good deterministic salt.

A good salt should be unique for your site and unique for each user. That way, if many users have the same password on your site, they'll get different hashes, and if a user has the same username and password on your site as another, the hash will be different there as well.

DETERMINISTIC SALT

Prefix + userid



com.example.MyGreatSite:robnapier@gmail.com

A very simple but effective salt is a unique string for your password database plus the userid.

I'm adding the colon here just for readability. It doesn't really matter if you use the salt the way I'm about to describe.

STRETCHING

- Real passwords are easy to guess
- To protect against that, make guessing expensive

Remember I said that passwords are very unrandom. The entire universe of passwords that your users will chose is really small in cryptographic terms. If an attacker is trying to crack a specific account and has stolen your database, even after salting it's practical to brute force it.

How to we protect against that? We make guessing expensive.

TIME TO CRACK

	Guesses per second	Crack 8-char password
Native	1 billion	2 months
+80ms/guess	12.5	15 million years

Say we increase the time to guess by 80ms. That's hardly noticeable for one password test. But if you're doing billions of tests like a password cracker does, it adds a lot of time. Of course it doesn't solve the problem of very weak passwords, but at least it slows things down and protects at least decent passwords.

```

let hashPrefix = "com.example.MyGreatApp" // Not a secret, but should be unique to you

func hashedPasswordForUsername(username: String, password: String) -> NSData {

    // Scale rounds based on your hardware. 10k for iPhone 4. Otherwise 100k.
    let rounds = UInt32(100_000)

    // This can be just about anything "long enough." 256-bits is nice.
    let hashLength = Int(CC_SHA256_DIGEST_LENGTH)

    // Construct our salt from the prefix and username
    let salt = "\(hashPrefix):\(username)".dataUsingEncoding(NSUTF8StringEncoding)!

    // Convert our password into data
    let passwordData = password.dataUsingEncoding(NSUTF8StringEncoding)!

    // The final result storage
    let hash = NSMutableData(length: hashLength)!

    // Perform the hash
    let result = CCKeyDerivationPBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        UnsafePointer(passwordData.bytes), passwordData.length,
        UnsafePointer(salt.bytes), salt.length,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1), rounds,
        UnsafeMutablePointer(hash.mutableBytes), hash.length)

    guard result == CCCryptorStatus(kCCSuccess) else {
        fatalError("Could not derive hash for user: \(username) (\(result))")
    }
    return hash
}

```

The standard approach to this is the Password Based Key Derivation Function, or PBKDF2. It's easy to use on iOS, and is widely available on other platforms.

Just build your salt and choose your number of rounds. I recommend ten thousand if you need to support iPhone 4. If you only run on desktops, then a hundred thousand is better. My target is around 80-100ms.

You also need to choose a final length for your hash. Anything "long enough" is fine. The goal is to make sure that real passwords represent a tiny subset of the available space. 32-bytes is great, but if that's too big for your database, going down to 16 or 20 bytes is still going to be fine.

Then call CCKeyDerivationPBKDF. It takes as a parameter a pseudo random function or PRF. Any cryptographic hash is fine here. Here I'm using SHA-256.

You'll notice a lot of "it'll be fine" going on here. That's because the point of PBKDF2 isn't cryptography. It's point is to be slow. As long as you select a unique salt and a decent number of rounds, it's hard to use PBKDF2 incorrectly.

STORE A HASH

- Before storing the key in the database, hash it one more time
- It hardly matters how, but hash it (might as well SHA-256)

I know we just hashed the password ten thousand times, but now that it's been sent to the server, do me a favor. Hash it one more time before comparing it to your database. Those ten thousand hashes protected your user. And the salting protected your user and all the other sites your user visits. But this last hash is for you.

If someone steals your database, then you don't want them to be able to log-in just with whatever they find in the password field. You want them to have to calculate something. So looking at your database, they only know the hash of the random number they need to provide. And they're never going to find that.

If you follow this, then even if you published your password database freely to everyone, you could be safe from attack... Don't do that by the way.

GOOD PASSWORD HANDLING

- Hash to hide the password
- Salt to make your hashes unique
- Stretch to make guessing slow
- Hash once more before storing

And that's about it for basic password handling.

Hash your passwords so that you can't leak the real password

Salt the hash so that people with the same password don't get the same hash, and people who reuse their password on multiple servers don't get the same hash

Stretch your passwords with PBKDF2 to make it hard for attackers to steal passwords even if they can steal your database.

And save the hash of the final value, so that a stolen database is no worries.

Questions?

CORRECT AES ENCRYPTION

Almost every system I see that uses AES encryption does it incorrectly, which isn't surprising since almost every piece of iOS sample code I find on the Internet does it incorrectly, too. I hear way too often "it's impossible to break AES," but it's definitely possible to break AES if you use it wrong. The strongest lock in the world is pointless if you store the key under the welcome mat. But that's going to end today.

First, I'm going to jump to the end and tell you how to easily do AES correctly. Then, I'm going to run through why it's correct and what to look for in a correct implementation. So first, what's the right answer?



USE MY LIBRARY

<https://github.com/rnapier/RNCryptor>

I hate to shill my own software. I really do. But I wrote RNCryptor specifically because I could not find an open source solution that was easy to use correctly on iOS and Mac. If there were a better solution out there, I'd shill for them instead.

First, I want to focus on the “easy” part. Then I'll talk about why it's “correct.”

USING RNCRYPTOR

```
// Encryption
let data: NSData = ...
let password = "Secret password"
let ciphertext = RNCryptor.encryptData(data, password: password)

// Decryption
do {
    let originalData = try RNCryptor.decryptData(ciphertext, password: password)
    // ...
} catch {
    print(error)
}
```

- | | | | |
|---------------|-----------|--------------|--------|
| • Swift | • C# | • Java | • Ruby |
| • Objective-C | • Erlang | • PHP | |
| • ANSI C | • Go | • Python | |
| • C++ | • Haskell | • JavaScript | |

Here's the encryption code in its simplest, most common use. That's it. It doesn't get a whole lot simpler than that.

<build>And the decryption is just as simple.

There's an asynchronous, streaming interface specifically designed for working with `NSURLConnection`, so it's easy to tie to REST services.

<build>And if you need to pass data to another platform, there are implementations of the `RNCryptor` format for OS X, iOS, PHP, Java, and pure ANSI C. I'm currently working on porting it to Python, and I've heard rumors of a JavaScript implementation.

WHAT IS CORRECT AES?

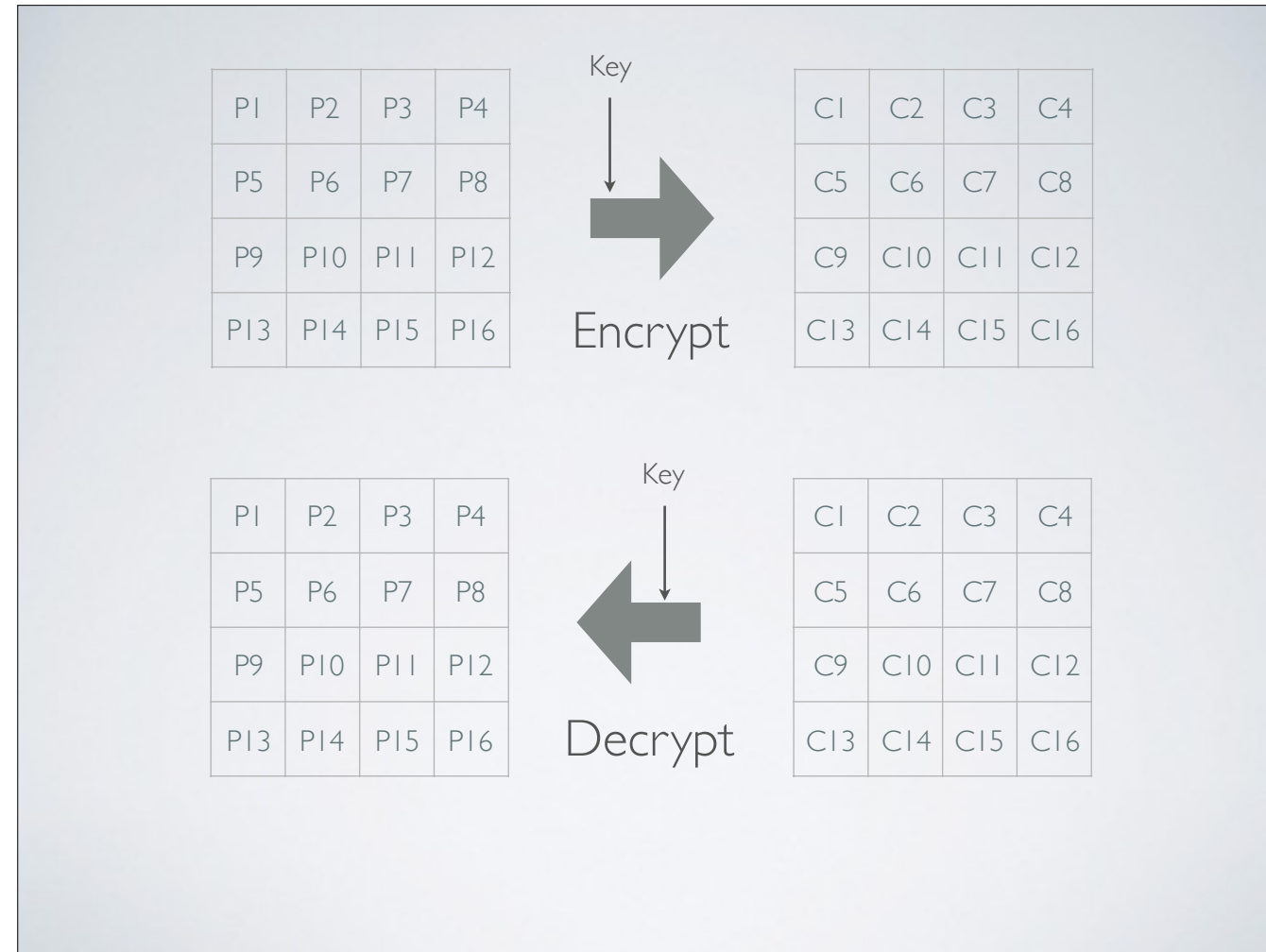
Hold that thought...

So that's how you use it. But *why* should you use it? I can say "trust me, it does it correctly and most everyone else does it incorrectly," and if you're cool with that take a short nap and I'll wake you at the end. But if you're a security guy, you're pretty wary of "just trust me," and that's good. So let's talk about what a correct AES implementation must have, and why, if you build that for iOS or Mac, you are basically going to rewrite RNCryptor.

So what is correct encryption?

<build>

Before we can answer that, I need to explain what AES actually is and what it isn't.



AES is a block cipher with a block-size of 16 bytes. That means it can encrypt exactly 16 bytes of data. No more, no less. That's all AES does. It takes 16 bytes of data, and based on a key, maps it to some other 16 bytes of data in a way that's reversible with the same key. That's it. That's all AES can do.

That key also has to be a specific length; exactly 16, 24, or 32 bytes long. No more, no less. And to be secure, that key needs to be effectively random.

"But," you say, "I use AES all the time to encrypt multi-megabyte files with my 8 byte password which is the furthest thing in the world from random." And I'm telling you, AES can't do that. AES needs helpers to do that, and the helpers are where things tend to go wrong and we wind up with insecure AES.

THE HELPERS

- Key Generation
- Block Cipher Modes
- Authentication

And here are the helpers.

<**build**> You need a way to convert non-random passwords into totally random keys, or *key generation*.

<**build**> You need a way to encrypt more than one block. There are lots of ways to do that, and the algorithm we use is called the *mode*.

<**build**> And finally you need a way to make sure that no one modified your data in transit. That's called *authentication*.

These are the three things that most implementations do incorrectly. I'm going to cover them really quickly because this is a short session. If you're around tonight, buy me a drink and I'll go on for hours on this stuff.

INCORRECT KEY GENERATION

```
// This is broken
NSString *password = @"P4ssW0rd!";

char key[kCCKeySizeAES256+1];
bzero(key, sizeof(key));

[key getCString:keyPtr maxLength:sizeof(keyPtr) encoding:NSUTF8StringEncoding];
// This is broken
```

- Truncates long passwords
- Uses only a tiny part of the key space
- Best case is ~ 0.00001% of AES-128

Use PBKDF2

The most troublesome helper is the piece that converts a password into a key. Almost every example I've seen in the wild does it basically this way, and it's completely broken. This code just copies the password byte by byte into the key, padding with zeros if it's too short and truncating if it's too long.

<build> The fact that it's throwing away part of the password should be the first clue that something is terribly wrong. But the whole approach is broken.

<build> A critical part of AES's security is the size of the key space. All the claims that AES is uncrackable rely on the fact that there are at least 2^{128} possible keys. That is a really enormous number. I'm sure you've all heard that it would take trillions of years or something to brute force it. And 256-bit dwarfs even that.

<build> But the above code throws away almost all of those keys. We need a way to map our password into the AES key space in a way that is indistinguishable from random. I'm not going to go into computational equivalency here, but what I'm describing is a cryptographic hash function, and we already have a good way to convert passwords into cryptographic hashes.

<build> We use PBKDF2. Remember? Password Based Key Derivation Function. A function for converting a password into a key. Problem 1 solved.

REQUIREMENT 1: PBKDF2 SALT

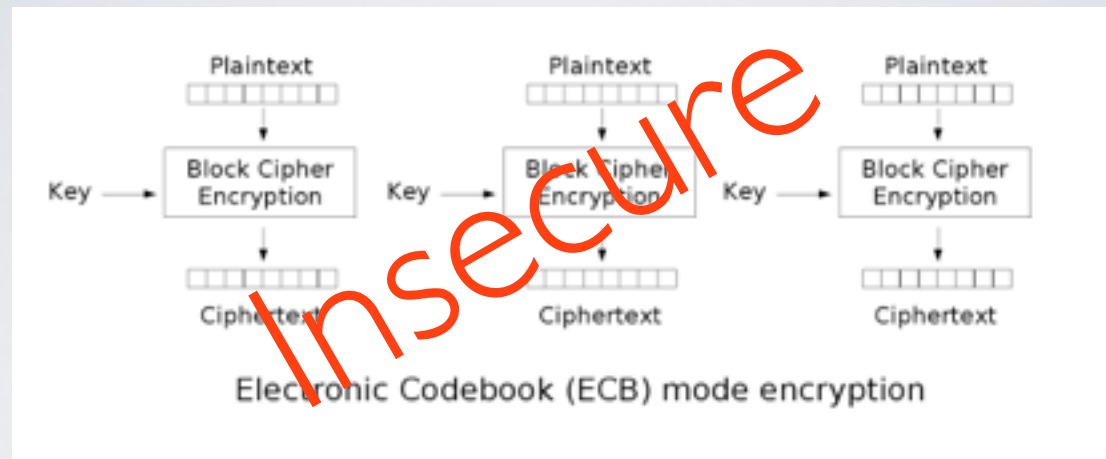
- To be a secure password-based format, we need a salt for PBKDF2. Ideally it should be totally random.

And that brings us to requirement 1. If the format accepts passwords, then there should be a random PBKDF2 salt somewhere in the file format. If there isn't, something's fishy.

INITIALIZATION VECTOR

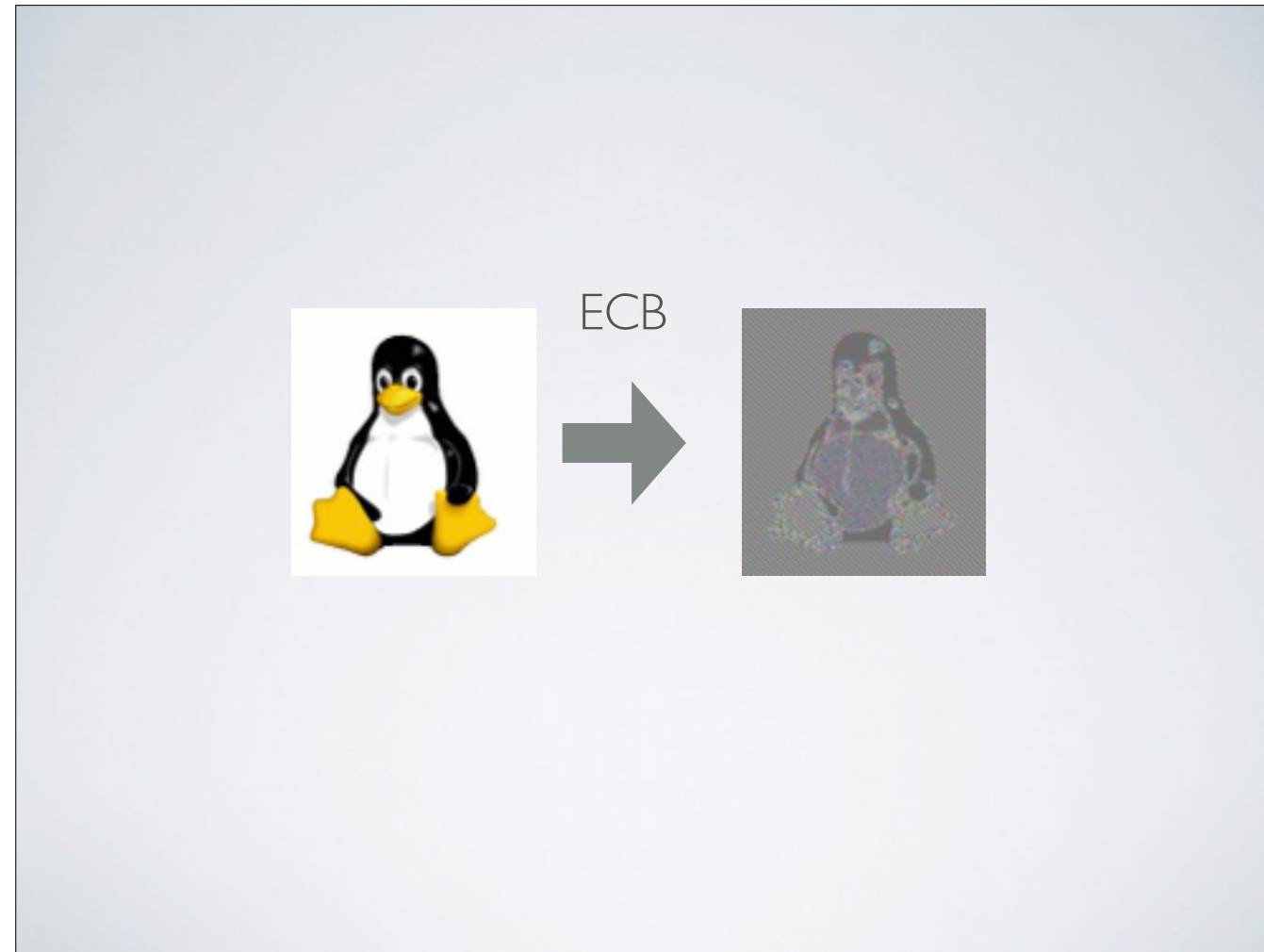
And Modes of Operation

Now that you have a decent key to work with, the other helper is the one that allows AES to deal with data of any length. There are lots of ways to do that, collectively called *modes of operation*.



First, let's discuss the most obvious mode. We'll take the first 16 bytes and encrypt them with the key we were given. Then we'll take the next 16 bytes and encrypt those with the key we were given. And so forth until the end. Great. That's easy and it has a name. It's called electronic codebook or ECB. It's also the worst possible way to encrypt data with a block cipher.

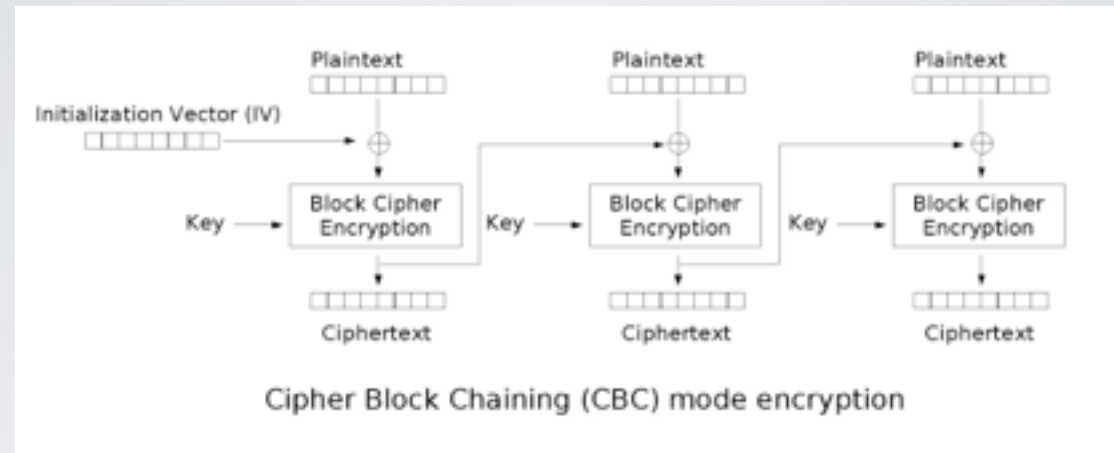
<build> In some cases it barely qualifies as encryption.



Let's take a bitmap and encrypt it with ECB. Here's the result... Not exactly secret data.

<build>The problem with ECB is that given the same key, any time the same sixteen bytes show up in the plaintext, say sixteen black pixels, the same sixteen bytes will show up in the ciphertext. I can use that to recover a lot of the plaintext, possibly all of it. Never use ECB. Yes, there is one obscure case where it's useful, but it never happens in the kinds of apps you're likely to write so don't even worry about it. Just never use ECB.

OK, ECB is off the table. So what should you use?



On iOS you should use cipher block chaining, or CBC. I'm trying to keep this brief, so I'm not going to go into how CBC works, but the point is that it breaks the patterns you saw in the last slide, and it's the easiest mode to use correctly that is also broadly available and specifically available for Common Cryptor. So just use CBC.

Luckily, CBC is the default in Common Cryptor, so why am I going on about it. Well, see that Initialization Vector the upper left-hand corner? Well, despite what it says in the header...

SO MUCH CONFUSION FROM ONE COMMENT

```
CCCryptorStatus CCCryptorCreate(
    CCOperation op,          /* kCCEncrypt, etc. */
    CCAAlgorithm alg,        /* kCCAlgorithmDES, etc. */
    CCOptions options,       /* kCCOptionPKCS7Padding, etc. */
    const void *key,          /* raw key material */
    size_t keyLength,
    const void *iv,           /* optional initialization vector */
    CCCryptorRef *cryptorRef) /* RETURNED */
__OSX_AVAILABLE_STARTING(__MAC_10_4, __IPHONE_2_0);
```

...it is not optional. If you pass NULL, which you probably are because almost everyone does, that creates an IV of all zeros, and that throws away some important protections. Most notably, if you encrypt the same data with the same key and the same IV, such as 0, then the ciphertext will be identical, and the attacker can use that to note that the same message was sent. If you encrypt lots of messages with the same IV and key combination, the attacker can actually begin to decrypt the data. That's how WEP was broken.

REQUIREMENT 2: RANDOM IV

- To be a secure format, we need a random IV.

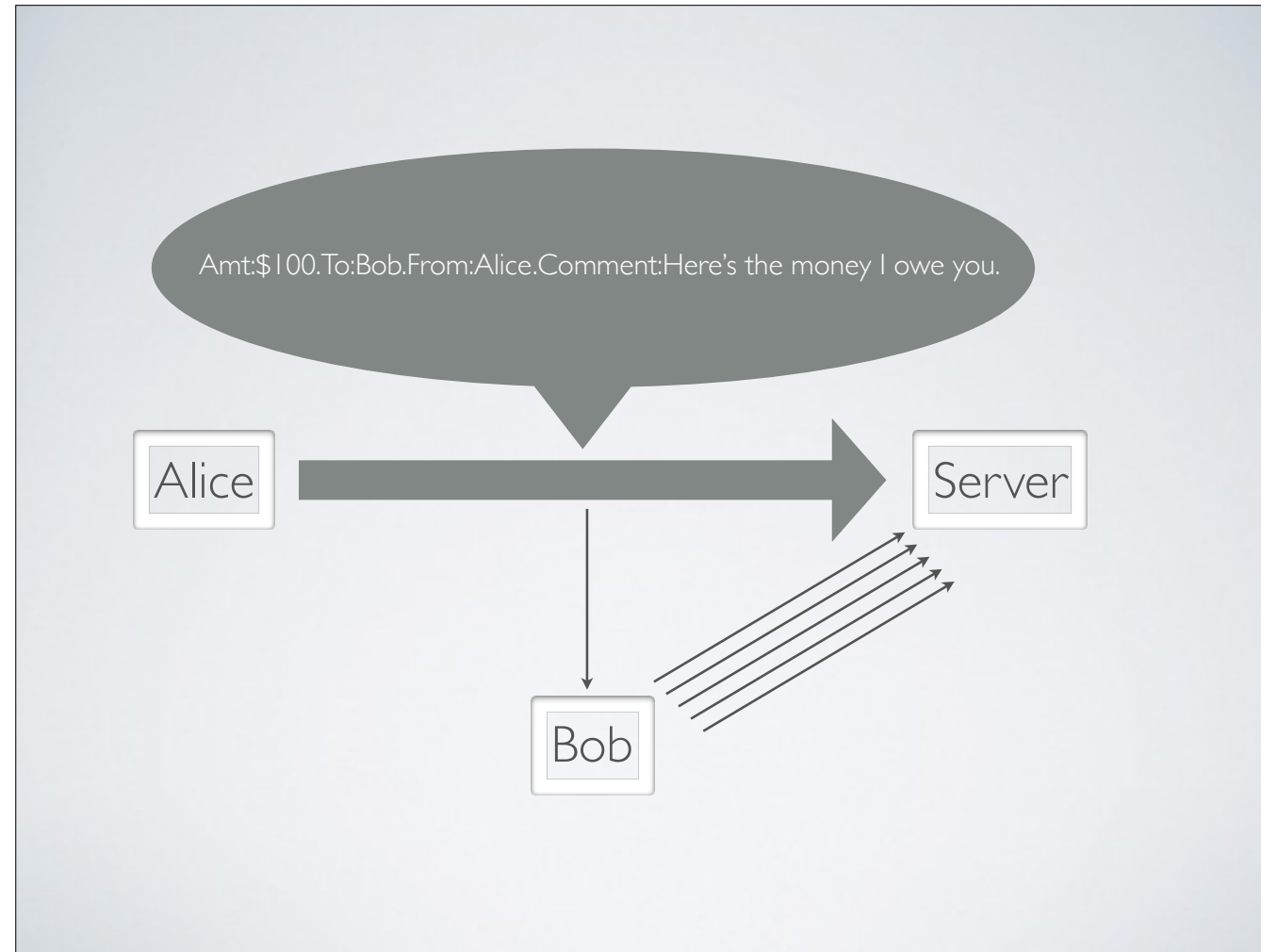
I'm not going to dive into the math here, but the point is that to be secure, you need a random IV for every message. So your data format had better include one.

UNAUTHENTICATED ENCRYPTION

This last issue is fairly complicated, and we don't have a ton of time, so I'm going to give you a feel for the attack, and I have some code on my site that demonstrates exactly how this works.

The key takeaway is that just because something is encrypted, that doesn't mean an attacker can't modify the contents, even if the attacker doesn't have the password, and this can be used to create some important attacks.

Say Alice and the server share a secret key, and Alice encrypts this message and sends it to the server.

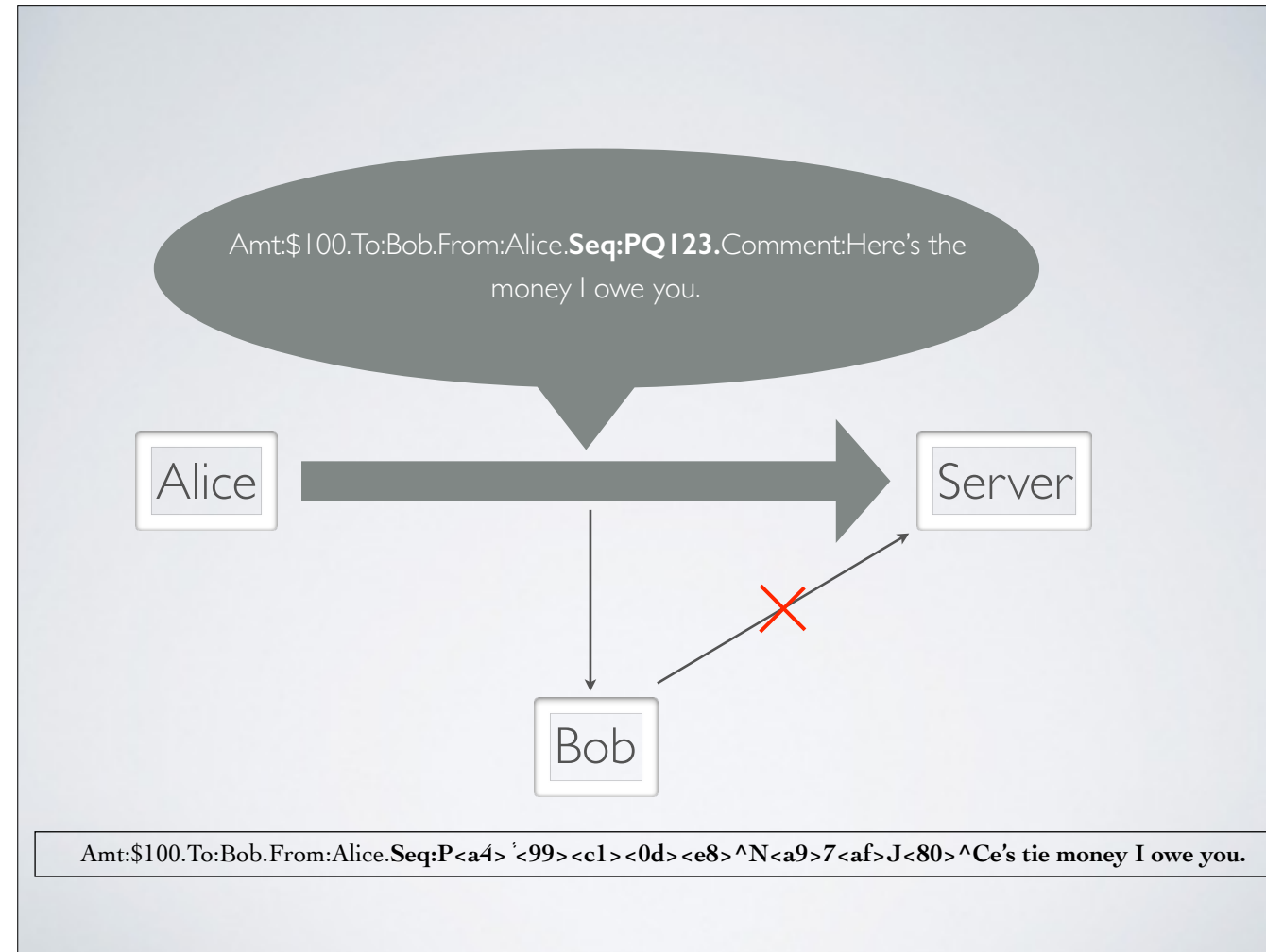


The server takes \$100 from Alice and gives it to Bob. Great. But there's a problem.

<build> What if Bob is eavesdropping on the conversation. He can't read it, but he can still make a copy and send it to the server again.

<build> And again. And again, until Alice is out of money. Poor Alice.

Let's try to fix this. We'll have Alice add a random sequence number to the end like this...

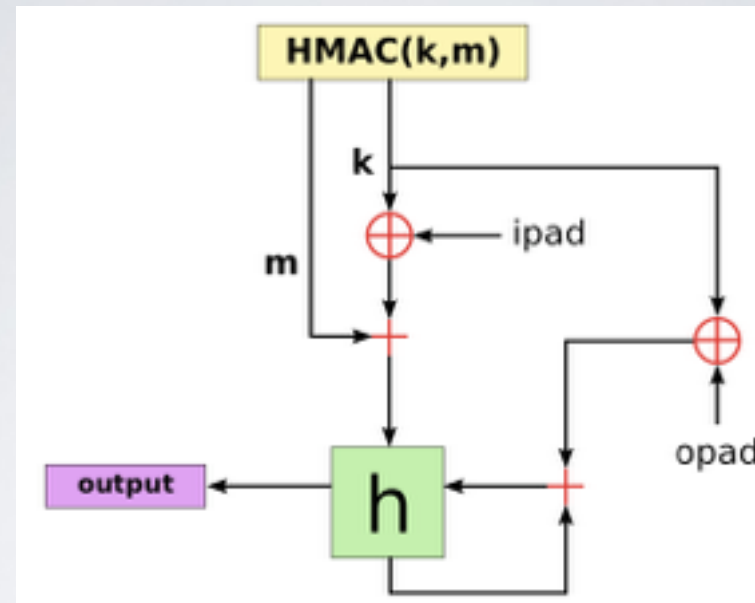


If the server sees the same number more than once, that's an error.

But there's still a problem. Notice that the sequence number starts at byte 32. That's means the entire sequence number is in its own block. Just as AES can encrypt every possible 16-byte block, it can decrypt every possible 16-byte bock. It might decrypt to garbage, but it'll decrypt to something. So if Bob modifies the encrypted message at byte 32, it might decrypt to something like this.

<build> Depending on the parser, this might be a legal sequence number, so Bob could still replay attacks.

But it gets worse.



HASH BASED MESSAGE AUTHENTICATION CODE

Since we're using CBC as our mode, the right answer is a Hash Based Message Authentication Code, or HMAC. An HMAC is sort of like an encrypted hash. You can also think of it as a cryptographic signature. Don't worry too much about the math right now. What's important is to calculate an HMAC, you need a secret key shared between the sender and receiver, and it shouldn't be the encryption key. The sender hashes the data using that secret key, and the receiver can use the key to verify the hash. If the data was modified in the middle, the hash won't verify.

But now we need two keys, right? An encryption key and an HMAC key. No problem. We can just generate a new random salt, hand the password to PBKDF2, and out pops a completely new, random key to use for HMAC.

REQUIREMENT 3

- To be a secure format using CBC, we need an HMAC.
- To use HMAC with a password, we need an HMAC Salt.

That brings us to our last requirement.

A secure format needs an HMAC, and to create its key from a password, it needs a salt.

ENCRYPTED MESSAGE CONTENTS

- A password salt for the AES key
- A password salt for the HMAC key
- An initialization vector for CBC mode
- The encrypted data itself
- The HMAC of the encrypted data

Unsurprisingly, this is the RNCryptor format.

So summing it all up, a proper encryption bundle based on a password will include the following things:

<build...>

If your password-encrypted message doesn't have at least those five things, you're probably doing something wrong.

And of course, this is almost exactly the data format of RNCryptor, plus a couple of bytes for the file format version and options. And so, unless you want to build your own file format, I suggest RNCryptor.

Questions?

PRACTICAL SECURITY

- Encrypt your traffic with SSL
 - Verify the cert
 - Commercial certs are good, but you can use your own
- Encrypt your files with ProtectionComplete
 - Configure this for the whole app
 - Turn it on per-file with NSData
 - Set it on existing files with NSFileManager

So that wraps it up.

<build> Encrypt your network traffic with SSL

<build> Make sure the certificate is trusted

<build> Get a commercial cert, or manually verify your own

<build> Encrypt your local files with Data Protection, specifically ProtectionComplete

<build> Ideally, turn it on for the whole app at iTunes Connect

<build> Or use NSData to protect individual files

<build> Or NSFileManager if the files already exist

PRACTICAL SECURITY

- Use Keychain for passwords
 - Use a wrapper like SwiftKeychainWrapper or SGKeychain
 - Use access groups to share information between your apps
- Never handle unhashed passwords
 - Use PBKDF2 to hash them securely
- Use AES correctly
 - Salts, IV, HMAC
 - Or just use RNCryptor

<build>Store your passwords in Keychain

<build>But use a wrapper

<build>And if you have multiple apps with the same passwords, use access groups to share them.

<build>Never use raw passwords. Always salt and hash them

<build>with PBKDF2

<build>And make sure you're including all the necessary parts of AES

<build>If you have passwords, you'll need an encryption and HMAC salt. And in any case you'll need an IV and an HMAC.

<build>Or just use RNCryptor which does it all for you.

robnapier.net/ cocoaconf
robnapier@gmail.com
@cocoaphony

Hey, let's be careful out there.

Any questions?