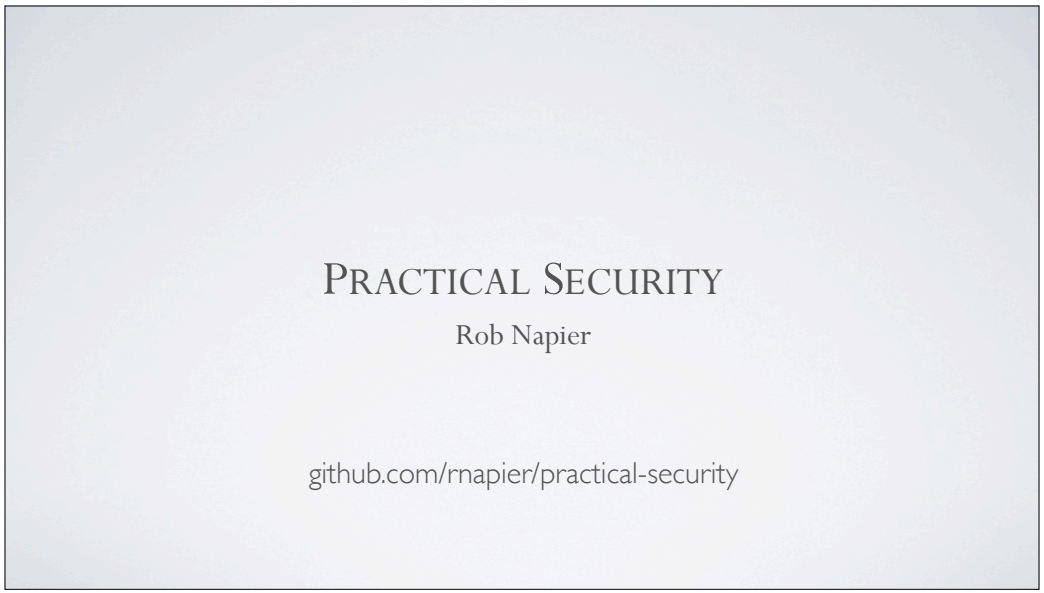


Security is tough. It's hard to know whether you're doing it right. That's a whole profession. It used to be mine before I was a Cocoa dev. But most of you don't plan to study security theory and cryptography. You just want to know how to make your system "reasonably" secure. Secure enough that it's not going to be embarrassing.

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.

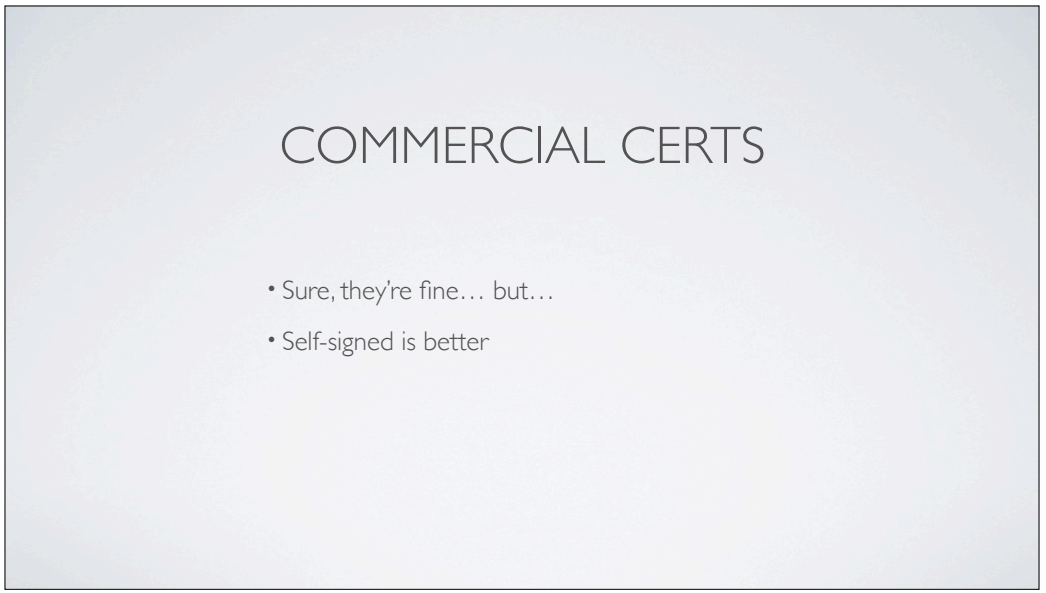
<build> From there, we'll talk about data's gatekeeper, the secret.

<build> And the most common kind of secret, the password, and how to handle them correctly.

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

If you don't do anything else to improve the security of your app, turn on HTTPS. Here's some of the stuff you get for free. It's a great example of a best practice. Done correctly, TLS solves a ton of problems.

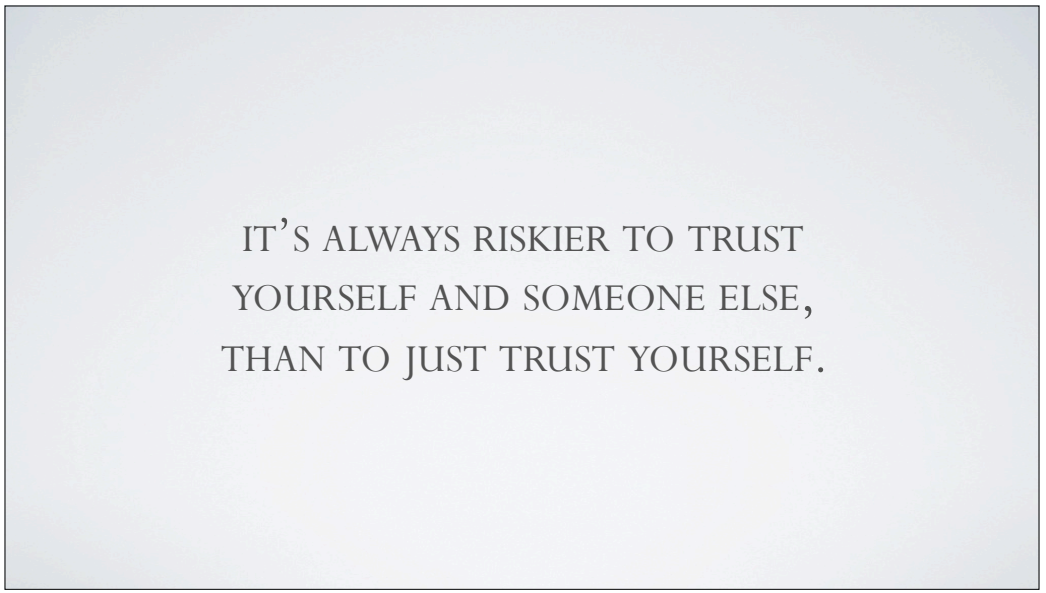
<build> You shouldn't assume that just because you have TLS, you don't have to worry about any other security problems. TLS doesn't fix weak user authentication. And it doesn't magically fix an insecure REST protocol. But it's a great example of a best practice. In one move it makes a lot of problems go away.



Commercial certs are the most common way to implement HTTPS. You can get them pretty cheap these days, and there's nothing wrong with them.
<build> But they're not the most secure solution. The "whatever" you're paying Verisign isn't buying you security. It's buying you trust by browsers. But your app probably isn't running in a browser. It's an app. What do you care if Safari accepts your cert?
<build> Done correctly, self-signed certs are more secure.

"What? Isn't a self-signed cert totally insecure?" No. The Verisign root is "self-signed." That's what it means to be a root certificate.

If you only trust your own cert, that's better than trusting the 168 roots that iOS trusts. When you get a commercial cert, you need to trust yourself to keep your private key secret, and you also have to trust the provider to keep their private key secret.



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

Tell you what. We'll get rid of this technically correct, but misunderstood term, and replace it…

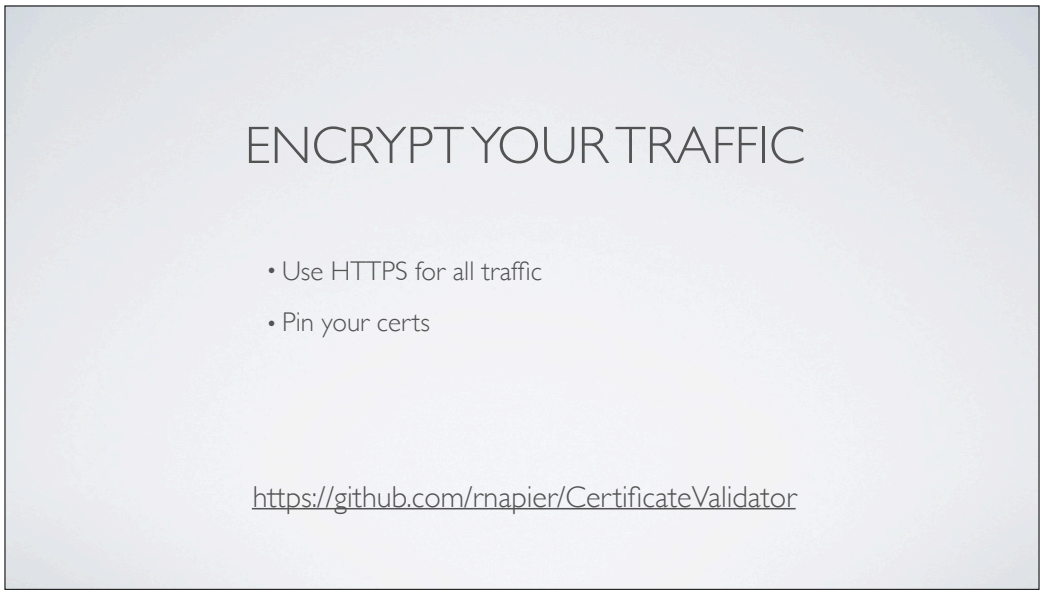
…with another term that explains what we're really doing.

Pinning doesn't mean we turn off certification verification and accept any random cert. Pinning means we accept just one specific cert. Ours.

The code for this is a bit tedious. It's not that hard, but there's no reason for you to recreate it all. I've made a small helper class to do it for you called CertificateValidator, and you can study that if you want to do it yourself.

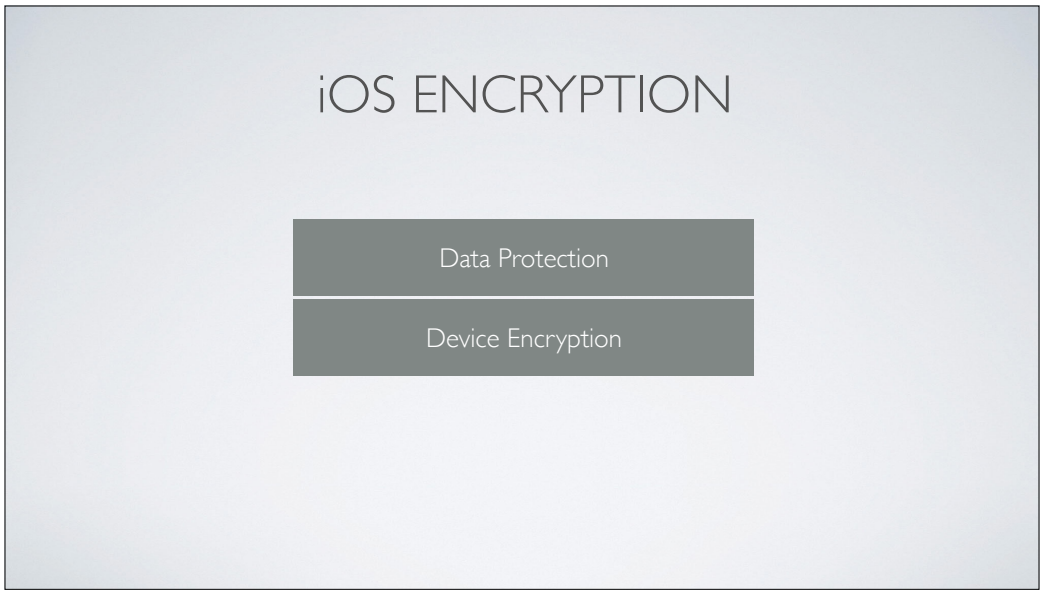
With the helper, this is all you need to do.

First, save your public key in your resources. Then create a validator with your certificate. And then create an URLSession with the validator as the delegate. It'll take care of the rest. If the certificate is wrong, you'll get an error. Otherwise, it's all transparent.



That's it.
<build> Turn on HTTPS.
<build> Pin your certs.
Here's the code. And the link will be on the main site.

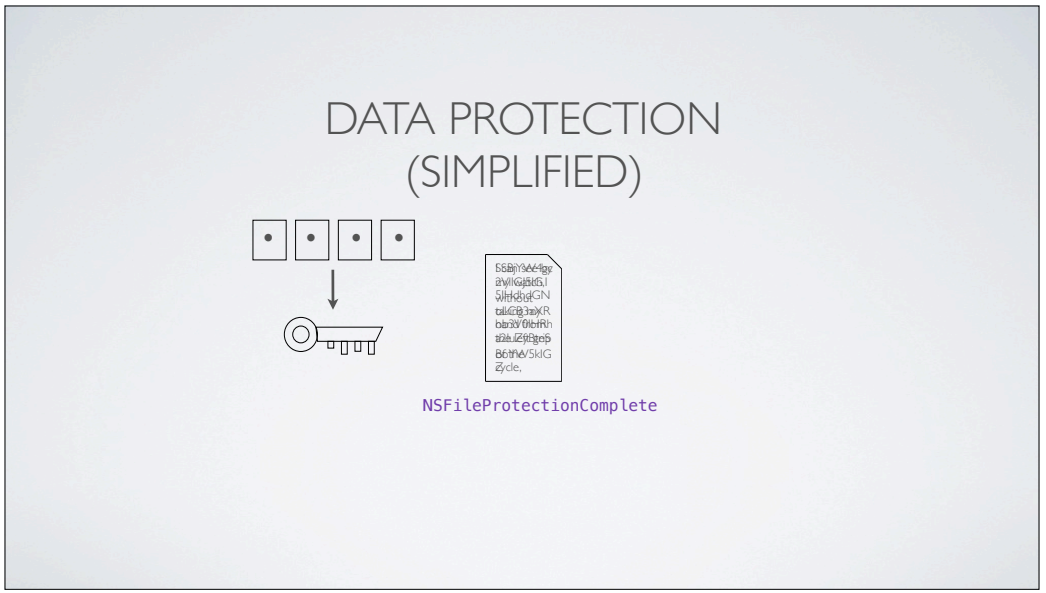
Now that you've downloaded some sensitive data, let's look at how to keep it safe on the device. iOS provides built-in data encryption, and you should be using it.



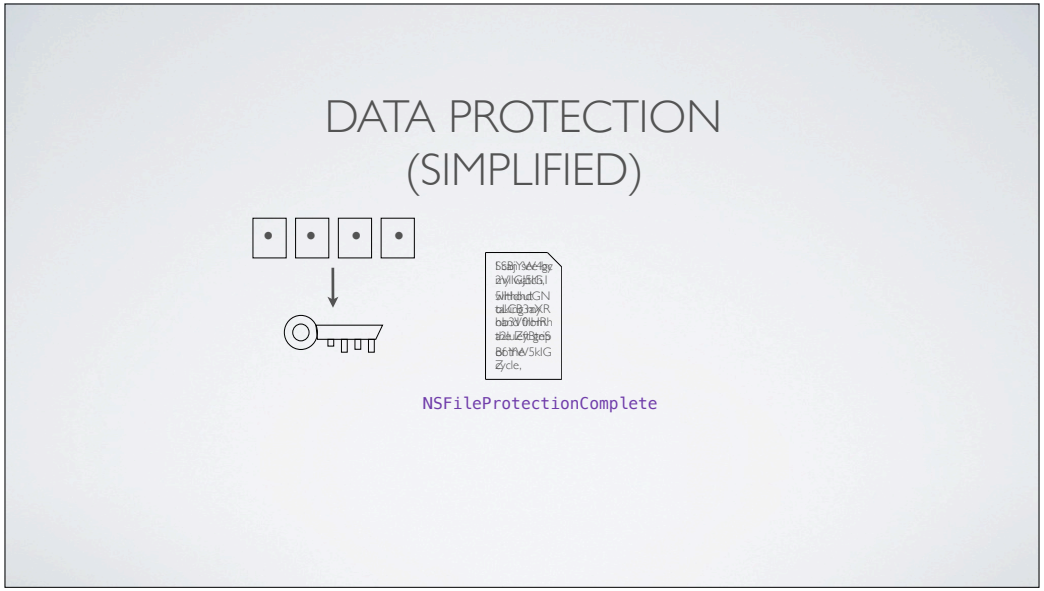
First, a quick intro to encryption in iOS. There are two levels of encryption: device encryption and data protection.

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. 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 an 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 Apple's full explanation.

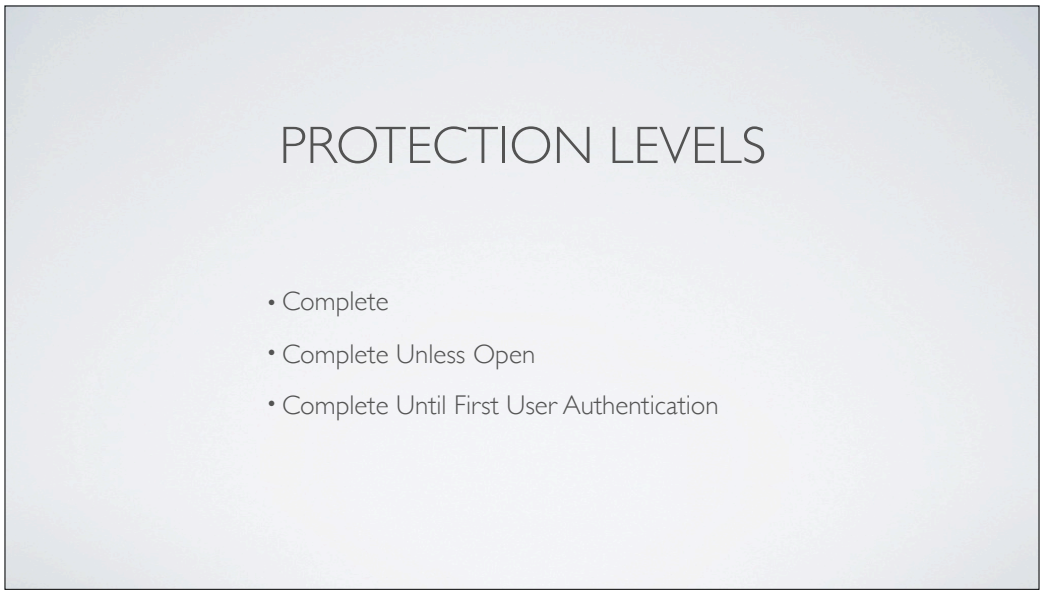


You have a document you want to protect.
<build> You mark it as needing complete data protection.
<build> The system will derive a key for that file based on the user's PIN.
<build> The file will be encrypted immediately, but the key will be held in memory so your application can still access the file.
<build> When the device is locked, then within a few seconds the key will be scrubbed from memory and the file will be inaccessible.



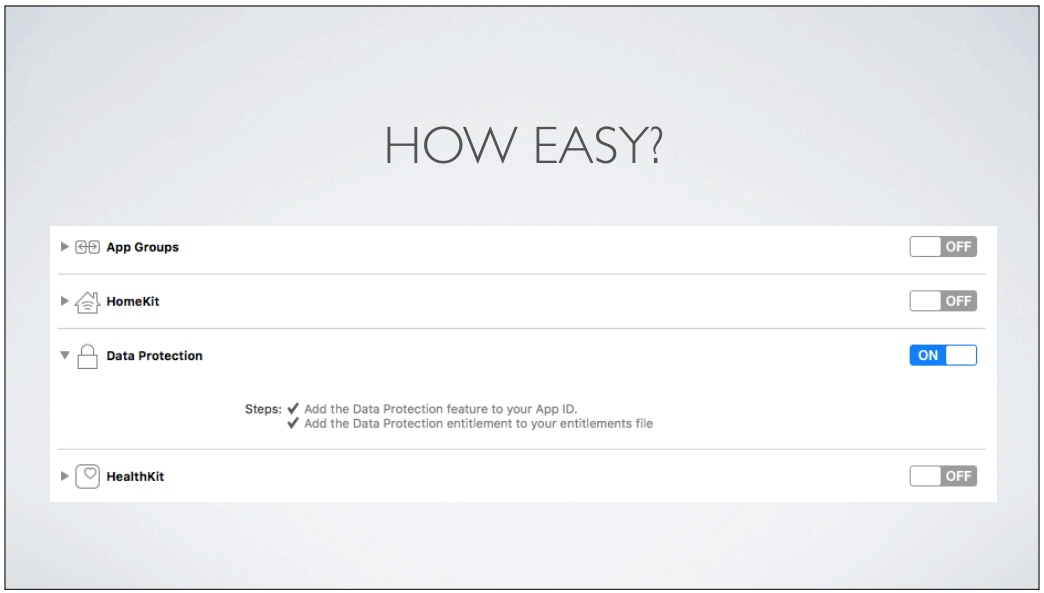
Then when the user unlocks the device, the key and data becomes available again.

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.



But sometimes you do need to access files while the device is locked. So, there are several protection levels available.

<build> First there's Complete. That means the data is encrypted anytime the device is locked.
<build> Then there's Complete Unless Open. That means if the file is open when the device is locked, then it can be accessed until the file is closed. That includes creating new files while the device is locked. This is useful for downloading or uploading and for log files.
<build> And finally, there's Complete Until First User Authentication. These files are only protected from the time the device is booted until the first time the user unlocks it. Only rebooting will protect the file again. It's not much, but it does protect the against some kinds of attacks, so it's better than nothing.

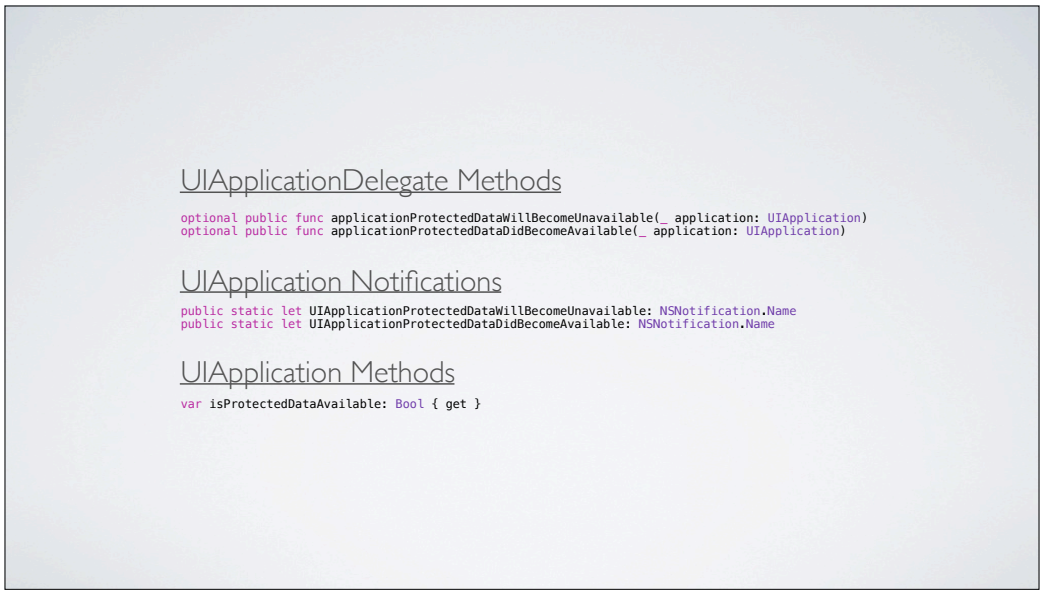


So that's great, but how do you actually use it? For most apps it couldn't be simpler.

Go to the capabilities pane in Xcode. Enable Data Protection. And you're done. The default protection level is complete, so you won't have access to files when the device is locked. But you can fix that for the files you need.

But sometimes you need a little more control. For those, you have two options.
<build> You can apply the setting while writing a Data to disk by passing an option.
<build> Or you can apply the protectionKey attribute to existing files using with FileManager. You can do this even if the file is open. I'll post these examples online, but I hopefully nothing here is that surprising once you know the feature exists.

You can apply these to any kinds of files you want. Core data files, sqlite databases, text files, anything.

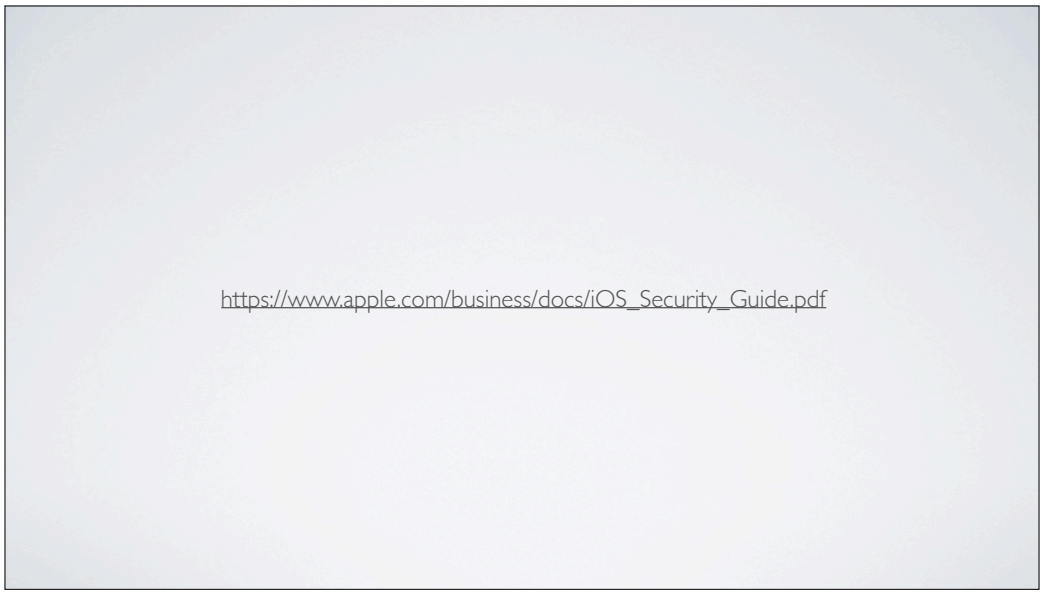


You can also find out when data is going to become available or unavailable. That let's you make decisions about what files to write or when to change a file's protection level.

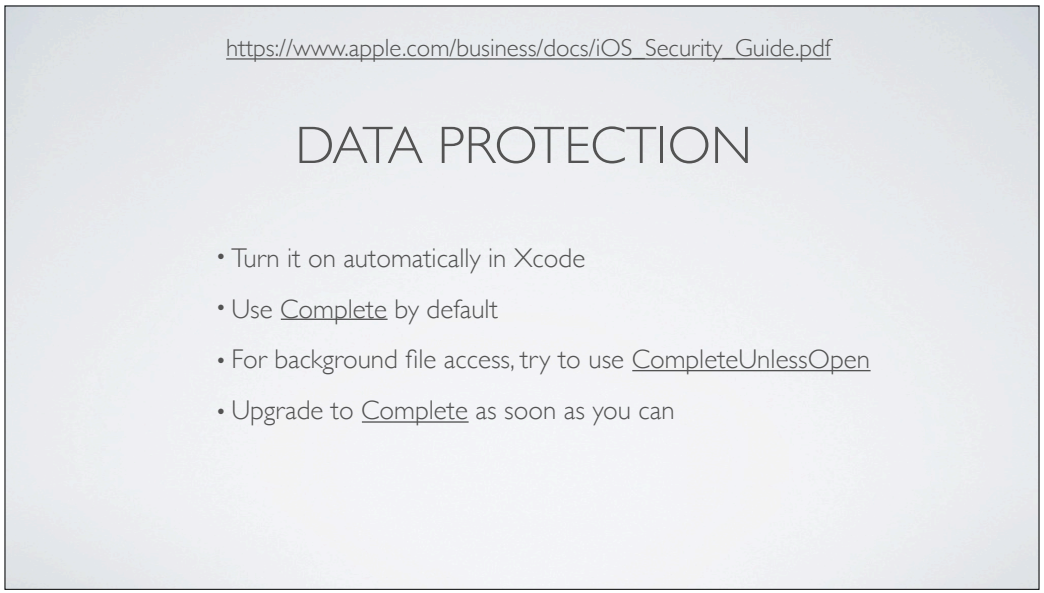
I've made a big deal about how easy data protection is, and it is, and you should use it. But I also want to be honest about a really common bug that happens when you turn it on. It's easy to assume that the device will always be unlocked when you're launched, but sometimes that's not true. There are a lot of features that can cause you to launch at surprising times. The one I've personally run into the most is background fetch. You suddenly find your app waking up at midnight to download things, but the device is locked so you can't read any of your files.

The usual solution to this is to only write data, never read data, while the device is locked. You can create a file with the protection level Complete Unless Open. And that let's you write to it all you want until you close it. Then, when you are properly launched and the device is unlocked, either upgrade that file to Complete using the file manager, or merge it into your larger database and delete the temporary file.

Fixing these problems usually isn't that hard. The hard thing is remembering that it can happen. Things like background fetch are difficult to test, so sometimes this is something you only find out about in the field, and



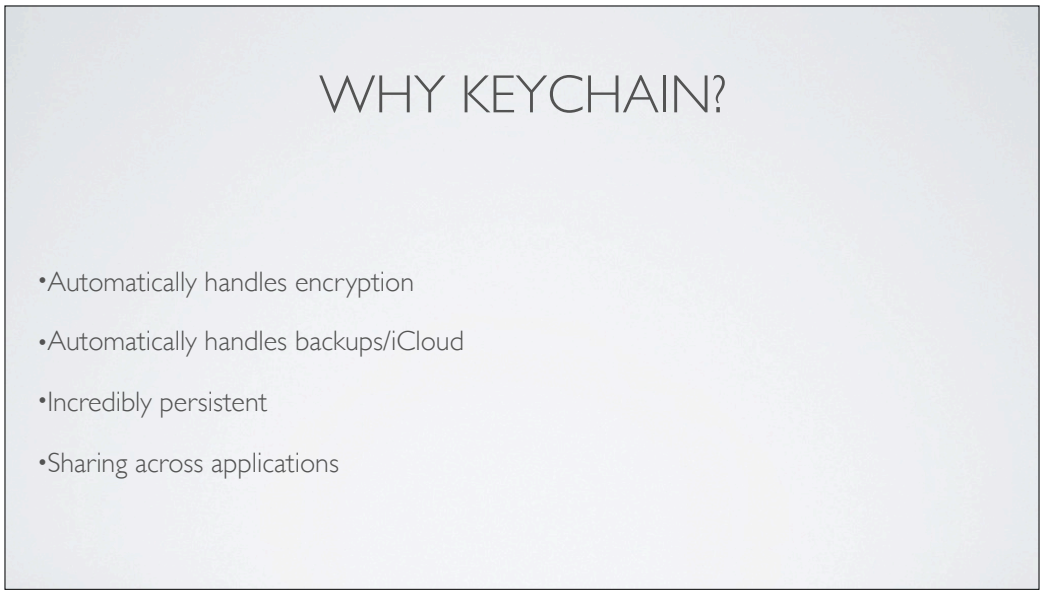
My description here was a little over-simplified. It's enough to get how it all works, but it's not actually exactly how iOS implements it. If you want the gory details, see this PDF from Apple. I'll include it in the notes.



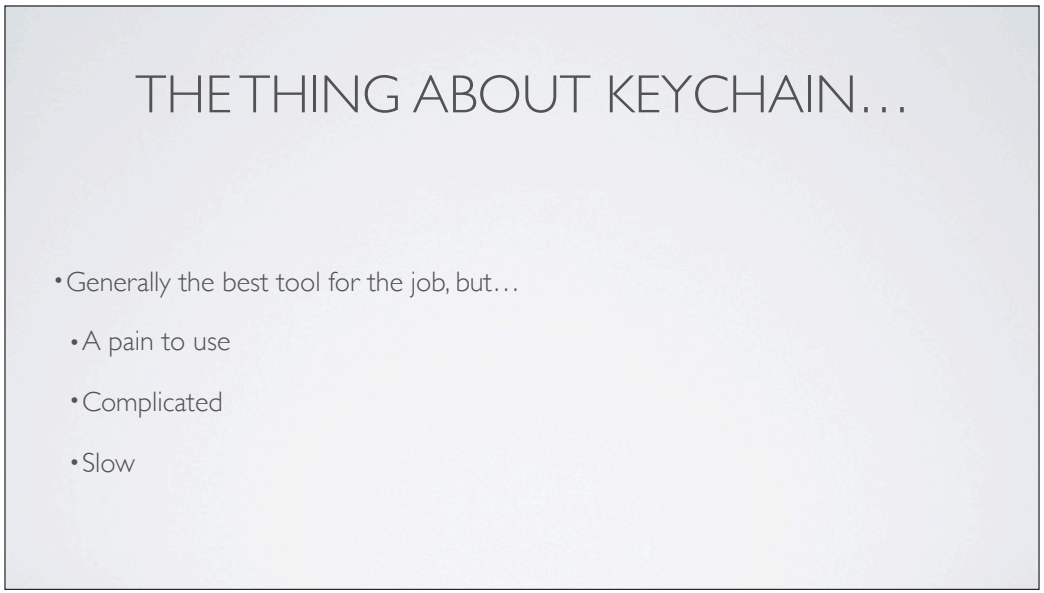
But in most cases you should:
<build> Turn data protection on for your project
<build> Set it to Complete by default unless you have a really good reason not to.
<build> For specific files that you need to access in the background, set them to CompleteUnlessOpen.
<build> And when you're done working on them in the background, upgrade them to Complete using file manager.

And that's data protection.

A “secret” is a small piece of data used to access other data. Think passwords and private keys, and usually the best way to store them is with Keychain.



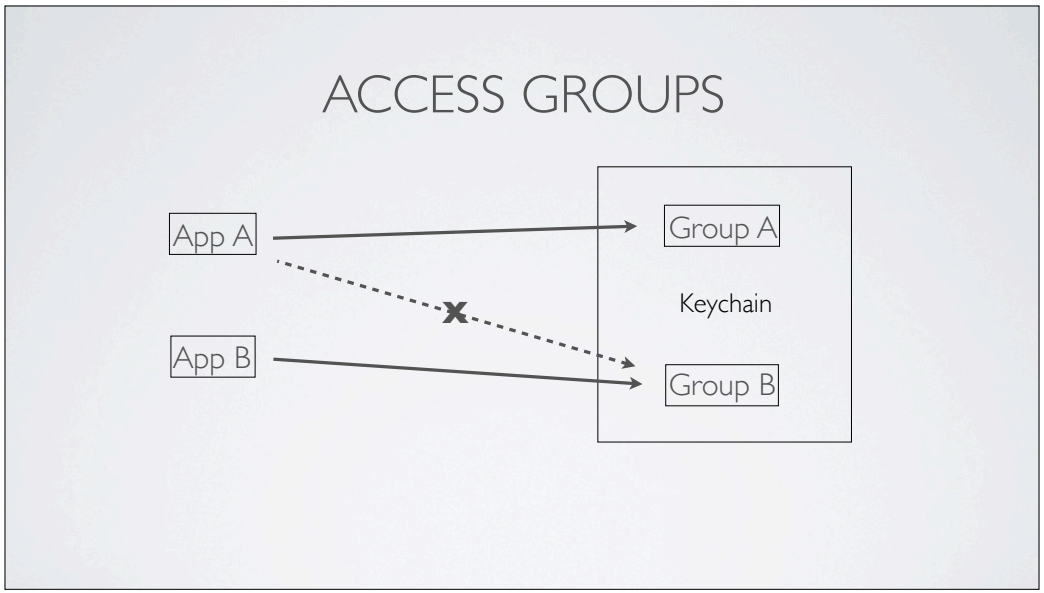
Why Keychain? Well, it's what Apple gives us, and it handles most of the details pretty well.
<build> It automatically handles encryption
<build> It automatically handles securing backups to iTunes and iCloud
<build> It persists on the device over a reinstall.
<build> And keychain let you share credentials between your apps.
But…



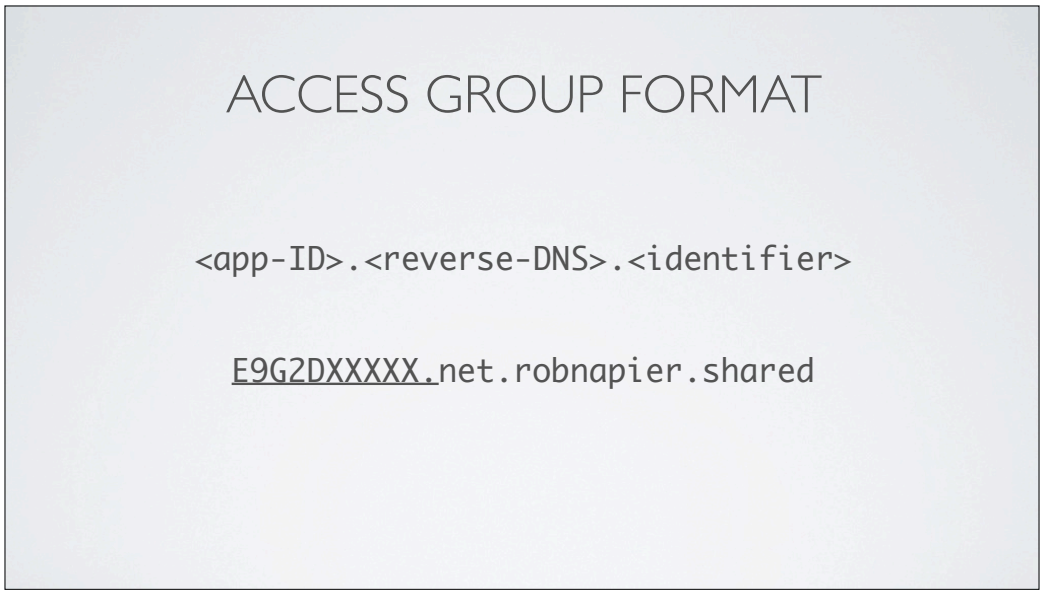
Unfortunately, it's a bit of a pain to use. It has a confusing API. And it's really slow.

The best way to deal with Keychain is a wrapper. 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.

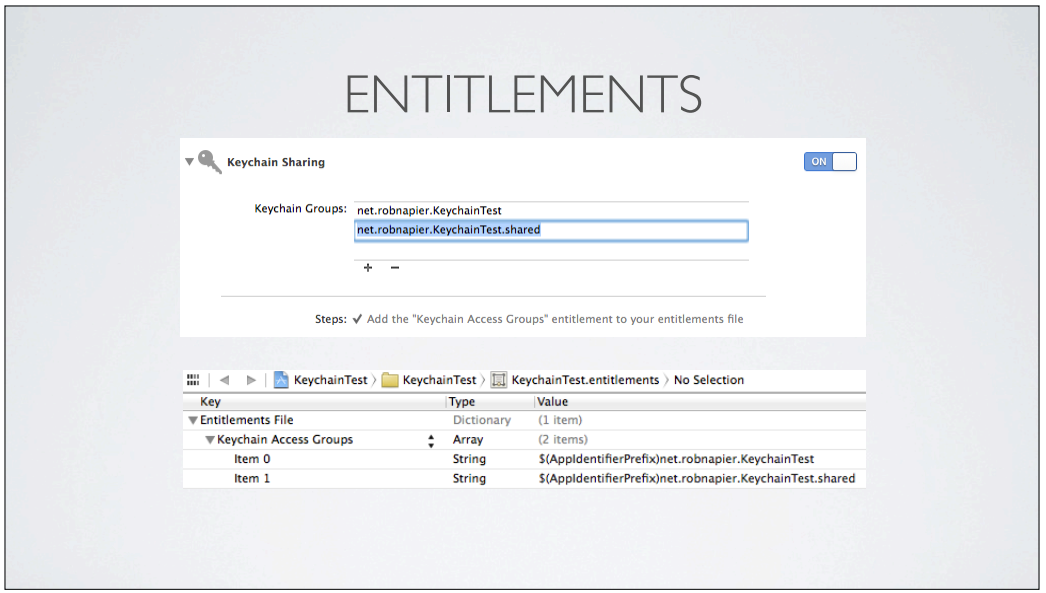


So, access groups. This is a really powerful feature in iOS, but it takes a few tricks to use it well.
<build> In iOS, there is really just one keychain shared by all apps. Each app automatically gets its own access group, and that's what makes it look like you have your own keychain.
<build> If you create a keychain record in one access group, other apps don't normally see it. But you can share access groups across apps, and so share records.



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

<build> Access groups have to begin with the App ID you signed the app with. So you can't read 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.



Each of your apps needs to include an entitlement to use this access group. In the Capabilities pane, turn on "Keychain Sharing" and add a new group. Don't include the app id here. Xcode will quietly insert it and a dot onto the front of whatever you type here.

<build> 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.

Here's where it get a little tricky. As long as all your apps list exactly the same access groups in their entitlements, everything will probably work. 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 identical items with different access groups. It's actually common, since unrelated apps might have the same account information. If you don't include an access group, that means "give me the first item you find that I have access to." This can cause you to read from one access group, but write to another.

<build> Because of that, I 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.

```swift
// Based on Objective-C code from David H
// http://stackoverflow.com/q/11726672/97337
extension UIApplication {
    func applicationIdentifier() -> String {
        let query: NSDictionary = [
            kSecClass: kSecClassGenericPassword,
            kSecAttrAccount: "applicationIdentifierQuery",
            kSecAttrService: "",
            kSecReturnAttributes: 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],
            let accessGroup = resultDictionary[kSecAttrAccessGroup as String] as? String,
            let identifier = accessGroup.components(separatedBy: ".").first
            else {
                preconditionFailure("Found garbage in keychain: \(result)")
        }

        return identifier
    }
}
```

But there's a trick you can use to get this at runtime. I will post this online, don't try to copy it. But here's how you do it. You create a random keychain item. It doesn't matter what it has in it. You then search for it, and ask what access group it's in. You can then strip off the first part and that's your app ID. It's ugly, but effective.

Here’s how you use it. Fetch the application ID and append your shared identifier. Use that access group from all your applications, and they’ll share their keychain items.

So those are my recommendations for Keychain.
<build> Use a wrapper. Please, use a wrapper.
<build> And if you want to share keychain items between apps, always use an explicit access group.

It seems every time you turn around, there's another password leak. No matter how trivial the information on your site, you have got to treat passwords with care. Users reuse their passwords all the time. If your cat lovers' site is hacked and the bad guys use that to break into your users' bank accounts, you don't get to say "well, they shouldn't have reused their passwords." They do and you need to deal with it. Always assume that a username/password combination is highly sensitive.

The first step towards handling those passwords is avoiding passwords. Ideally, your server should never see actual passwords. It should see some kind of hash of a password. Remember, you don't want to know what the user's password is. You just want to know that the user knows what their password is. So, at a minimum, you should be logging in with hashes of passwords, not passwords themselves. Just doing that alone gets you a lot of security and requires almost no changes to your backend. Just store base-64 encoded hashes as the password, and it just works. We can do better than that, but it's a start.

First though, we need to pick the right kind of hash. What we want is a cryptographic has.

<build>This isn't the same as a standard hash, like the hashes used for Dictionaries and Sets.
<build> What makes cryptographic hashes special is that they make it practically impossible to find collisions or to reverse. Cryptographers like to say "find a preimage."

If you think about it, you'll realize that any hash function must have an infinite number of collisions. There

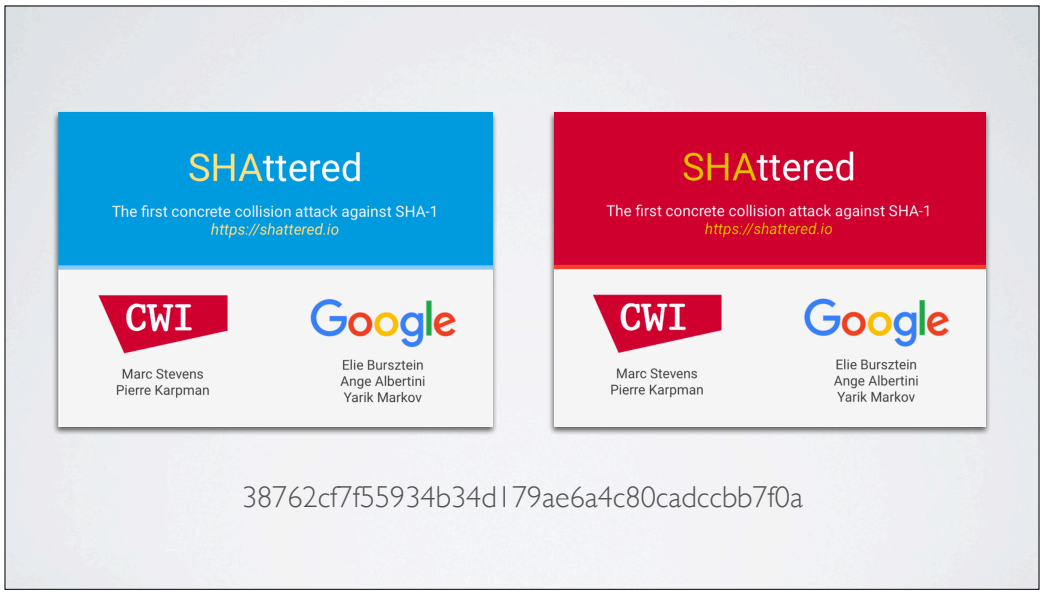
So what should you use?

<build>SHA-2. I used to have a more complicated slide here. But there really isn't any need to make a hard decision here. The answer is SHA-2. MD5 is completely broken. You can find collisions in seconds. SHA-1, I'll talk about more in a second, but it's somewhere between broken and very-soon-to-be broken, so you have to stop using it. SHA-3 exists, but there aren't that many implementations of it, and don't worry about it. Just use SHA-2.

<build>But SHA-2 comes in a lot of flavors, which can be confusing. All of these are SHA-2. They're differently lengths, and some are truncated versions of SHA-2, but they're all SHA-2. For your purposes, you probably only need to worry about these two.

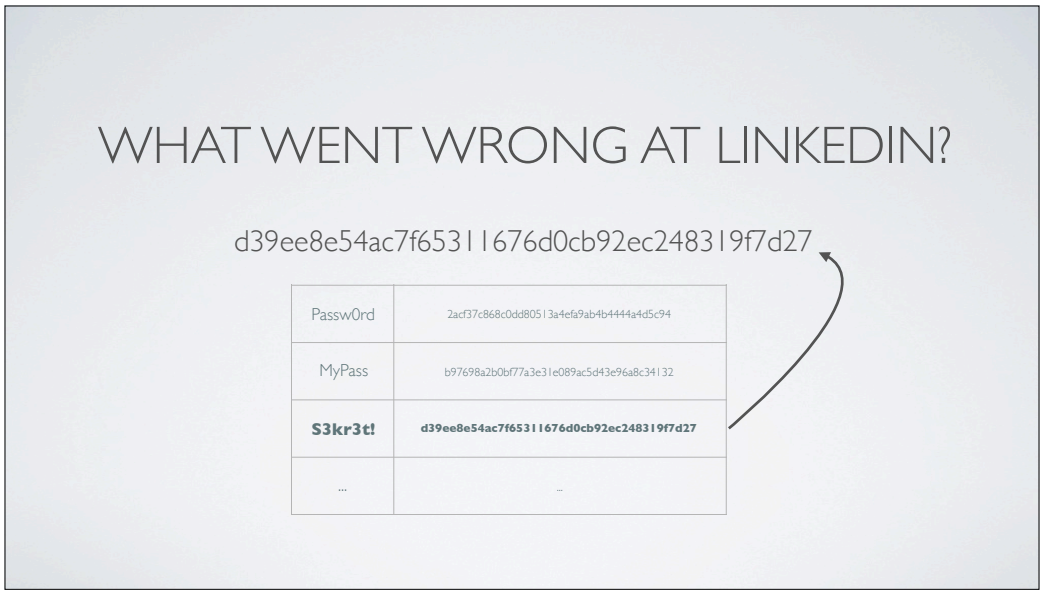
SHA-256 or SHA-512. Again, these are just lengths of SHA-2 hashes. The others, 224 and all, are truncated versions, mostly to deal with special situations. So short answer, if you can afford 64 bytes, use SHA-512. If you would rather 32 bytes, use SHA-256. They are both excellent hashes. SHA-256 is very strong, SHA-512 is just stronger.

But let's talk a little about SHA-1.



Earlier this year, Google generated two PDFs, shown here, with the same SHA-1 hash. That would be bad by itself, but what they developed was a general way for constructing colliding PDFs. It takes about 100 GPU-years currently. That means 100 years on one GPU, or 100 GPUs for a year, or 1000 GPUs for a month. For cryptography, that's a really fast break. I told you there are an infinite number of hash collisions, so obviously it's always been possible to find them. But this is 100,000 times faster than the normal approach.

You may hear people say this doesn't matter because it's too expensive or too hard, only Google could do, or it's just one document or just PDF, or it's not a "preimage attack." This was a very, very important collision. Once someone finds this kind of attack, future attacks get faster and cheaper very quickly. The first practical demonstration of an MD5 collision was in March, 2005, and it took days to compute. One year later, it took less than a minute on a laptop. Once a crack happens, it gets faster very quickly. The time to move to SHA-2 is now.



But cryptographic hashes aren't enough. LinkedIn, a few years ago, had a massive password leak. They hashed their passwords, but still, attackers were able to extract them. They used SHA-1, yeah, but this was long before anyone had found collisions. The same attack would work against SHA-2. So what's the attack? How do I find a password if I only have the hash? Well, let's say I'm an attacker.
<build> Just given this hash, I can't work it backwards to find a string that will generate it.
<build> But, 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, and that's what happened to LinkedIn.

These tables of hashes are called rainbow tables and you should assume that most attackers have easy access to the SHA-1 and SHA-2 hashes of all common passwords and lots of uncommon ones.

We fight this problem with salting.

Salting adds something unique to the password so that two uses of the same password don't generate the same hash.

You see here I've added XXX-colon to one, and YYY-colon to the other, and the hashes are completely different. So I just need a way that every user gets a different salt.

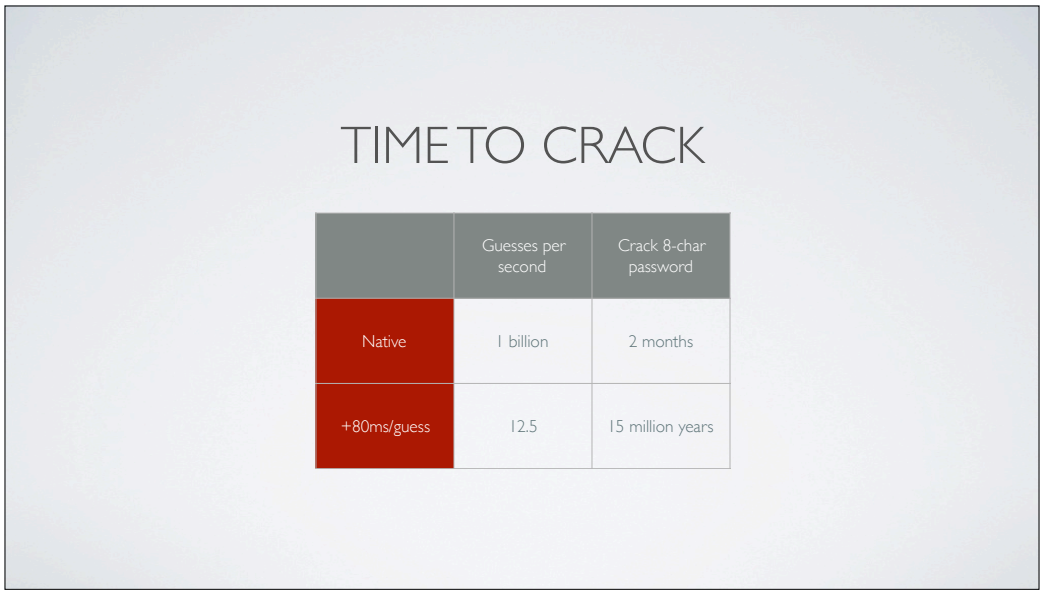
The best salts are totally random, and are generated when the user sets their password. But that's a little complicated for login. Clients have to first connect, ask the server what the salt is for a given user, then compute the password hash, and then login, and that back and forth can be a bit inconvenient, so I'm going to discuss how to build a good deterministic salt, which is often nearly as good, but much easier to implement.

The point of a salt is that it be unique for your site and unique for each user. That way, if multiple users have the same password on your site, they'll get different hashes, and if a user has the same password on your site as another, the hash will still be different.

<build>You can get that with a salt like this one. You pick a unique, static string for your password database like com.example.MyGreatSite, and you append the userid. So as long as no one else uses the same identifier, and there are no duplicate userids on my site, then this is going to be a unique salt. Now this isn't a secret. It doesn't matter if other people know how your salts are created. So it's ok that attackers can guess what the salt is going to be. All we're trying to do is make sure that even if two people have the same password, they'll get different hashes.

That said, the entire universe of likely passwords is very small in cryptographic terms. If an attacker is trying to crack a specific account and has stolen your database with all the hashes, even with salting it can be practical to brute force one account by just guessing lots and lots of passwords. People just aren't that creative.

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

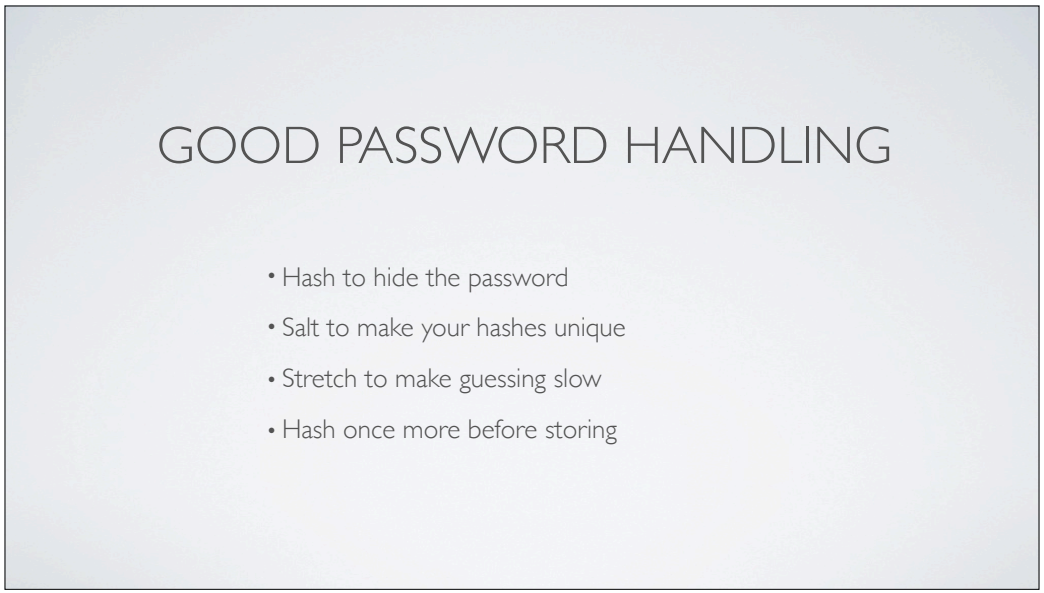


Say we increase the time to guess by 80ms. That's hardly noticeable for one password test. But if you're doing billions of tests like a password cracker does, it adds a lot of time. I mean going from a couple of months to millions of years. Of course it doesn't solve the problem of very weak passwords,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

So how do you actually do it? Here's an example using CryptoSwift, which is a nice, low-level crypto library written in Swift. You need to be a little careful when using CryptoSwift, because it's a low-level library. It assumes you basically know what you're doing, and so it's easy to build things with it that are very insecure, as we'll discuss later. But it's a nice library.

I know the client just hashed the password thousands of times with PBKDF2, but now that it's been sent to the server, do me a favor. Hash it one more time before storing it to your database or comparing it. Those thousands of iterations protected the user. And the salting protected the user and other sites. But this last hash is for you.

If someone steals your database, then you don't want them to be able to login with what 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 number they need to provide. And since SHA-2 is not practically reversible, there's no way for them to find that number.



So that's password handling.
<build> Hash your passwords with a cryptographic hash
<build> Salt them to make them unique
<build> Stretch them to make them hard to guess
<build> Store and compare a hash of the final result

So that wraps it up.
<build> HTTPS
<build> Pin and verify your certs
<build> Use Data Protection with ProtectionComplete
<build> Use Keychain for storing passwords
<build> Always salt and stretch your passwords before sending them to the server

Let's be careful out there.