Practical Security

Rob Napier

github.com/rnapier/practical-security

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.

All the notes and frameworks will be linked here at this GitHub repository. I'll list it again at the end.

Encrypt Your Traffic

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. **<build>**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.

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.

I've given this talk a few times, and I used to dive into a talk about self-signed certificates here and how to use them correctly. But that's kind of irrelevant now that we have App Transport Security. Basically, if you don't buy a commercial cert, it's going to be a pain for you. So go buy a commercial cert. They're not very expensive, so just do it. You'll save yourself so many headaches.

More importantly, though, don't turn off ATS. If you have some advertising partner who says they can only serve ads over HTTP from random, unknown servers, you need a new advertising partner. I hate to be so blunt about it, but sometimes the answer is yeah, don't do that.

Certificate Pinning

So most of the time, that's fine. Get a cert, use HTTPS, and your golden. But, if you have an InfoSec team, one of them is probably going to come to you and say "you need to pin your certificates." And you're going to say, fine, what's that mean? And they're going to say, "I have no idea, but it's an OWASP best practices, so go do it." So, today I'm going to tell you what it means and why it really is a good thing. But it's something I only do when I have an app that needs a higher level of security, or if you have an InfoSec team that demands it. For most apps, just turn on HTTPS, and you're good.

So what's the problem that pinning is trying to fix?

There are 168 roots that iOS trusts. I was part of getting one of them in there, the Cisco one. I promise you your app has no need to trust anything signed by that Cisco cert. It's a good cert, very secure. It's just it's not used for anything you are ever going to need. And there are a lot of certs like that. Certs controlled by companies I've never even heard of.

IT'S ALWAYS RISKIER TO TRUST
YOURSELF AND SOMEONE ELSE,
THAN TO JUST TRUST YOURSELF.

Why's that a problem? When you have a certificate, you always have to keep the private key secure. You can't avoid that. But also, all the trusted roots have to be kept secure. Well, no matter how much you trust Verisign, it's always riskier to trust both you and Verisign, than to just trust you, right? And there are a 167 other certificates you have to trust too. That's a lot of trust.

Certificate Pinning

Pinning means we limit the certificates we trust. We shrink that trusted list from 168 to 1 or maybe 2 certs. Just our cert, or maybe just the cert that issued our cert. Like if you get all your certs from Verisign, you could say, I only trust certs that came from Verisign, not ones that came from Echoworx or whatever. That's pinning.

```
try! validator = CertificateValidator(certificateURL: certificateURL)

session = URLSession(configuration: .default, delegate: validator,
                     delegateQueue: nil)

task = session.dataTask(with: URLRequest(url: fetchURL)) { … }
```

https://github.com/rnapier/CertificateValidator

The code for it is a bit tedious. It's not that hard, but there's no reason for you to recreate it all. It's maybe half a page of code in it's simplest form. I've made a small helper 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.

https://github.com/rnapier/CertificateValidator

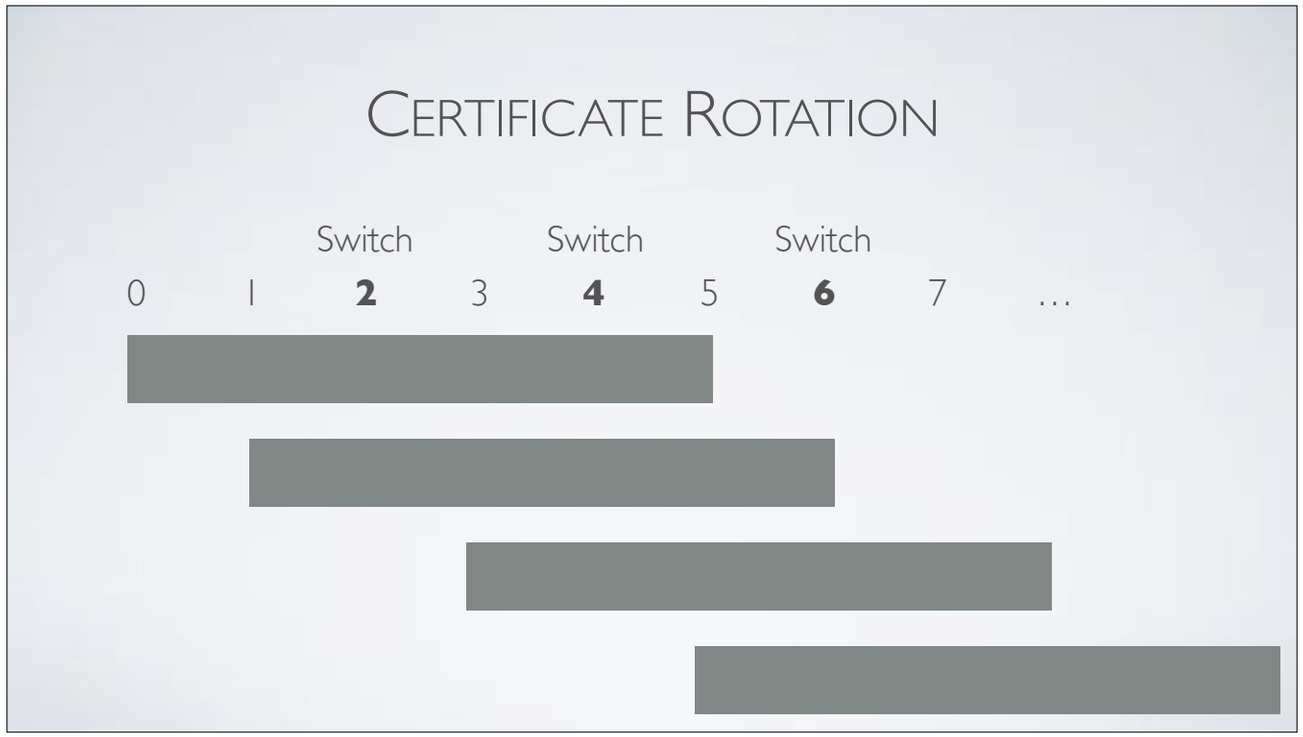
https://github.com/datatheorem/TrustKit

https://talk.objc.io/episodes/S01E57-certificate-pinning

CertificateValidator is pretty close to example code. You can use it as-is if you want, but like I said, it's very simple and designed to be modified to fit your needs.

If you want a more powerful version, look at TrustKit. It's a little more complicated to use in my opinion, but it has a lot more features.

And if you want even more about how to do this by hand, there's a free episode of Swift Talk you can watch that will walk you through more details.



I do want to mention one problem with all certs, and that's the fact that they expire. So everything's going great, and then one day nothing connects. This is always a problem, but you especially have to think about it with pinning. So what do you do?

My advice. Get 5-year certs. Pin to the first one. After a year, get a new 5-year cert and ship that one in the app as well. Pin to both. In year 2, after you've been shipping the new cert for a year, switch to the second cert on the server. In year 3, start shipping another new cert in the app, and in year 4 switch the server. Tick-tock. This assumes your users upgrade pretty frequently, within a year. If your customers are slower to upgrade, you might have to stretch this out.

The goal here is to deal with expiration by overlapping the certs, but why so fast? Why ship a new cert every two years when they're good for five? Because if you wait any longer than two years, no one will remember how to do it. You have to make rotation a regular activity or you're suddenly going to have a panic release to fix all your clients.

ENCRYPT YOUR TRAFFIC

- Use HTTPS for all traffic
- Pin your certs
- Rotate certs regularly

https://github.com/rnapier/CertificateValidator

https://github.com/datatheorem/TrustKit

https://talk.objc.io/episodes/S01E57-certificate-pinning

So that's it
<build>Turn on HTTPS.
And if you have a need for it,
<build>Pin your certs.
<build>and rotate them regularly
That's it. Links will be on the site.

DATA PROTECTION

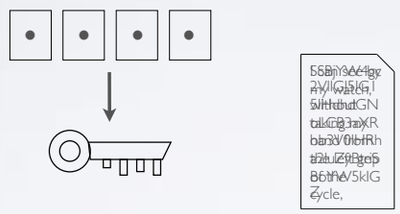
Now that you've downloaded some sensitive data, let's look at how to keep it safe on the device. iOS provides built-in data encryption, and you should be using it.

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.

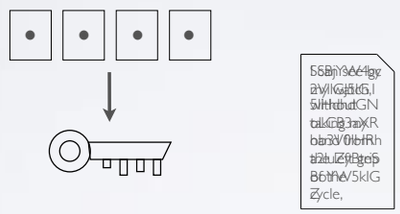
DATA PROTECTION (SIMPLIFIED)



NSFileProtectionComplete

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.

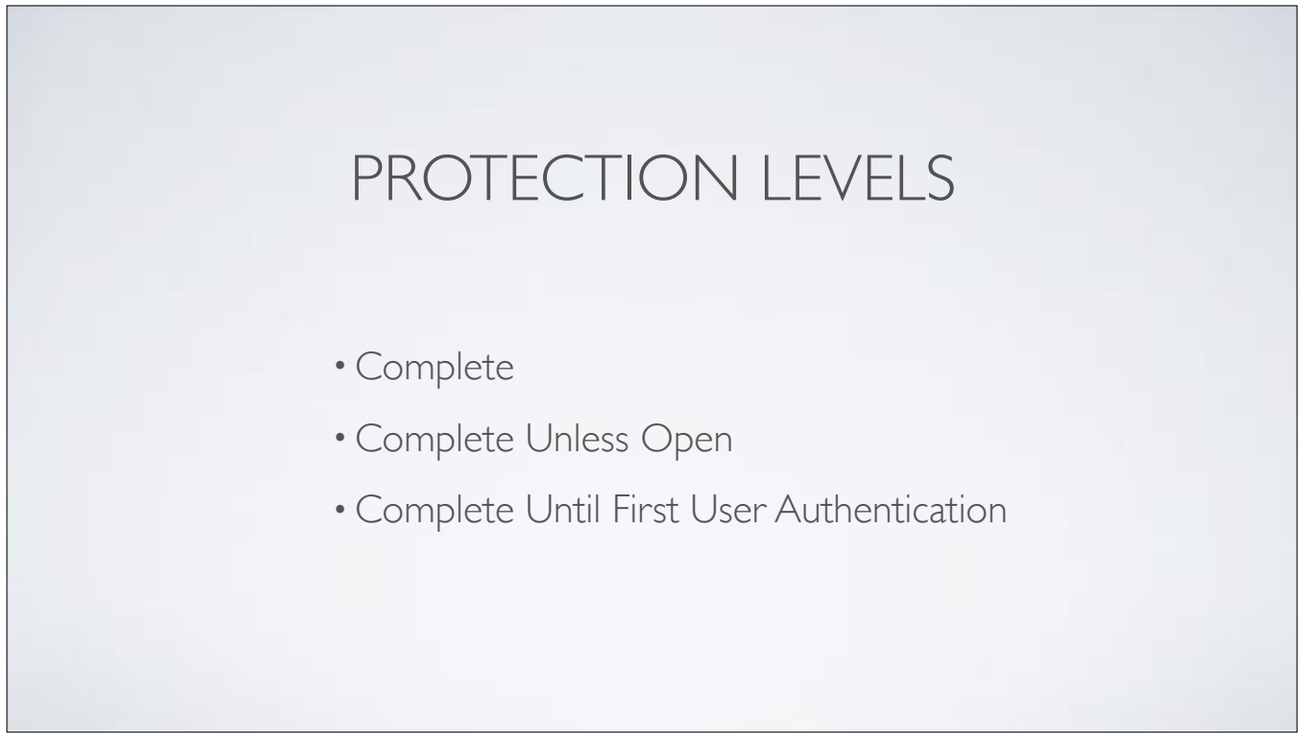
DATA PROTECTION (SIMPLIFIED)



`NSFileProtectionComplete`

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

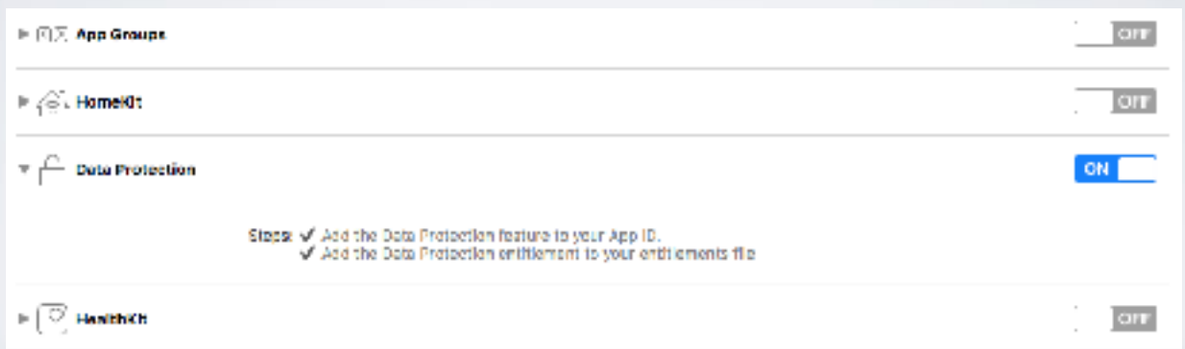
So very shortly after the device is locked, any file marked as ProtectionComplete can no longer be accessed. So what if we need to keep downloading something and writing that to a cache? Well, first ask whether you really need to keep downloading while the device is locked. There are good solutions, 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.

HOW EASY?



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.

DATA PROTECTION IN CODE

```
        try data.write(to: url,
                       options: .completeFileProtectionUnlessOpen)


extension FileManager {
    func protectFileAtPath(path: String) throws {
        try setAttributes([
                .protectionKey: FileProtectionType.completeUnlessOpen
                ],
                          ofItemAtPath: path)
    }
}
```

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.

UIApplicationDelegate Methods

```
func applicationProtectedDataWillBecomeUnavailable(UIApplication)
func applicationProtectedDataDidBecomeAvailable(UIApplication)
```

UIApplication Notifications

```
let UIApplicationProtectedDataWillBecomeUnavailable: NSNotification.Name
let UIApplicationProtectedDataDidBecomeAvailable: NSNotification.Name
```

UIApplication Methods

```
var isProtectedDataAvailable: Bool { get }
```

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.

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

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.

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

DATA PROTECTION

- Turn it on automatically in Xcode
- Use Complete by default
- For background file access, try to use CompleteUnlessOpen
- Upgrade to Complete as soon as you can

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.

Handling Passwords

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.

This isn't the same as a standard hash, like the hashes used for Dictionaries and Sets.
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 are an infinite number of possible documents, and we're going to hash them down to a finite number of hashes. Collisions are obviously rampant. But when you use a cryptographic hash, even though there are still an infinite number of collisions, the assertion is that you will never be able to find one. Given a document, you'll never be able to find another with the same hash, and given a hash, you'll never be able to construct a document that generates it without knowing the document beforehand. That's the kind of hash we want.

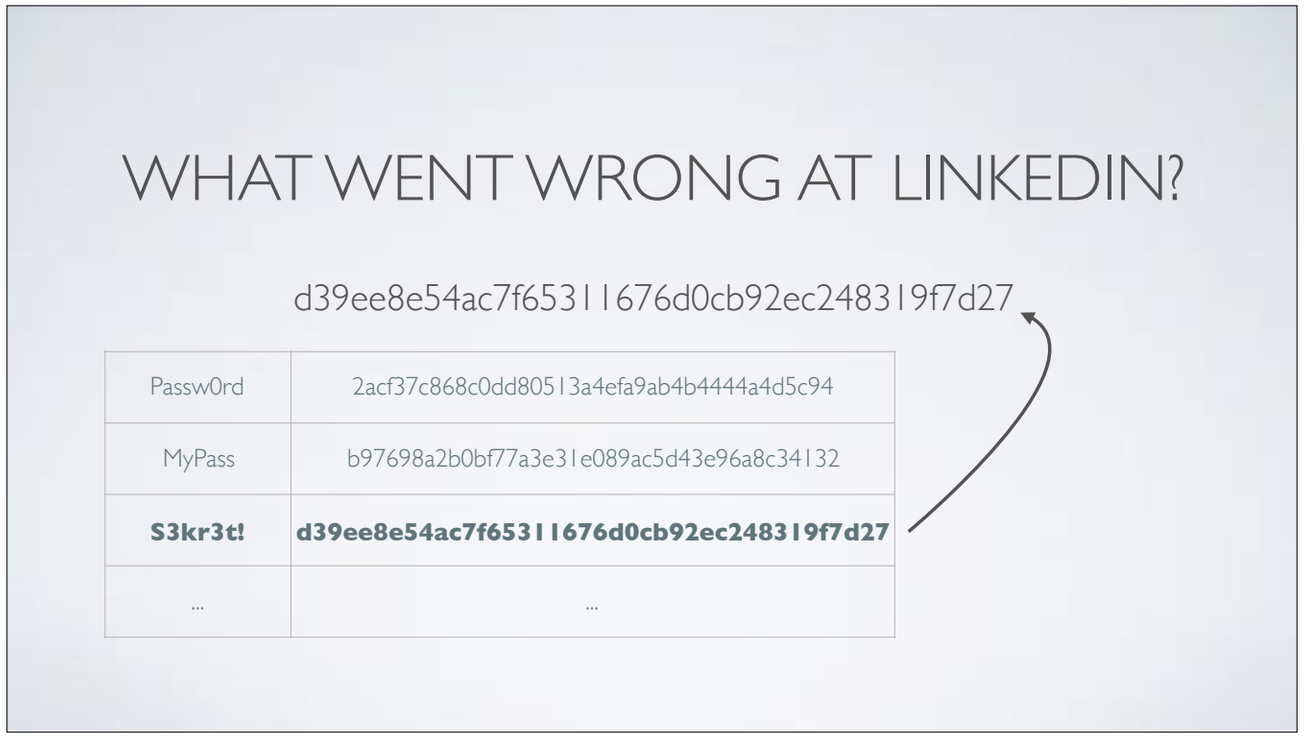
So what should you use?

<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 was broken by Google this year, and that attack will get much faster in coming months. 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.



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.

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. 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.

PBKDF2

```
import CryptoSwift

let password = Array("s33krit".utf8)

let salt = Array("com.example.MyGreatSite:robnapier@gmail.com".utf8)

let bytes = try PKCS5.PBKDF2(password: password,
                             salt: salt,
                             iterations: 10000,
                             variant: .sha256).calculate()

let data = Data(bytes: bytes)
```

https://github.com/krzyzanowskim/CryptoSwift

So how do you actually do it? The standard approach is the Password Based Key Derivation Function, PBKDF2. We'll talk more about why it's called that later, but it's just a function that's designed to be slow. It's easy to use on iOS, and it's widely available on other platforms.

Just build your salt and choose your number of rounds. I generally use 10,000 rounds. 100,000 rounds is even better for modern hardware, but it's too much if you have JavaScript clients. Then you pick a hash function. Doesn't really matter which one, but I like SHA-256.

You'll notice a lot of "whatever you want" going on here. That's because 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 it wrong.

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.

STORE A HASH

- Before storing the key in the database, hash it one more time with SHA-2

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.

GOOD PASSWORD HANDLING

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

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

CORRECT AES ENCRYPTION

Almost every system I see that uses AES encryption does it wrong. Almost any iOS encryption code you find on StackOverflow is wrong. Famous blogs by smart developers I respect tell people to do it wrong. Even articles that explain how to do it correctly leave most of the details as an exercise for the reader. The first edition of my iOS book was missing a key requirement, and I still run into people using that code.

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 leave the key in the lock. 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 what to look for in a correct implementation.

So first, what's the right answer?

USE MY LIBRARY

https://github.com/RNCryptor

Use my library. I hate to shill my own stuff. But I wrote RNCryptor specifically because I couldn't find a readily-available solution that was easy to use correctly for Cocoa. If there were a better solution out there, I'd shill for them. But rather than constantly telling people that they're doing it wrong, I decided to make it really easy to do it right.

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 { . . . }
```

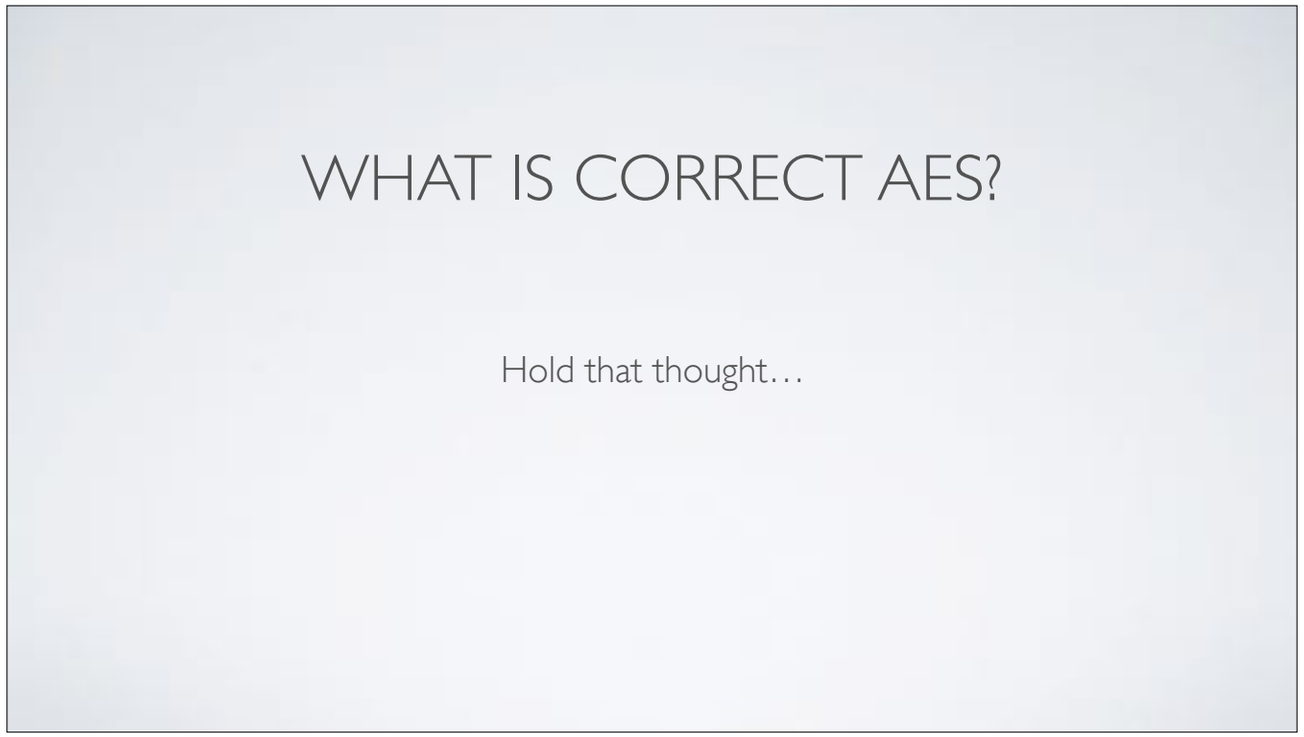
- Swift
- Objective-C
- ANSI C
- C++
- C#
- Erlang
- Go
- Haskell
- Java
- PHP
- Python
- JavaScript
- Ruby

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.

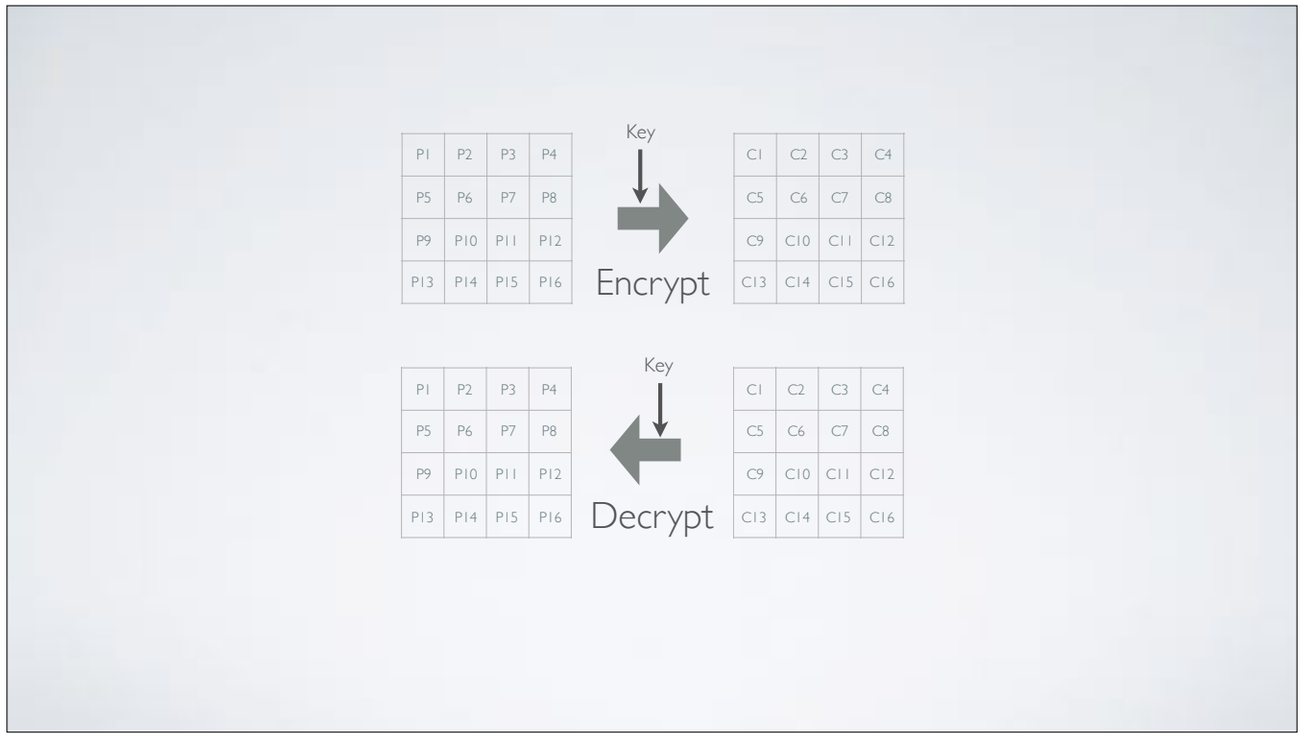
There's an asynchronous, streaming interface specifically designed for working with NSURLConnection, so it's easy to tie to REST services.

<build>And if you need to pass data to another platform, there are implementations of the RNCryptor format for Objective-C, Swift, PHP, Python, whatever.



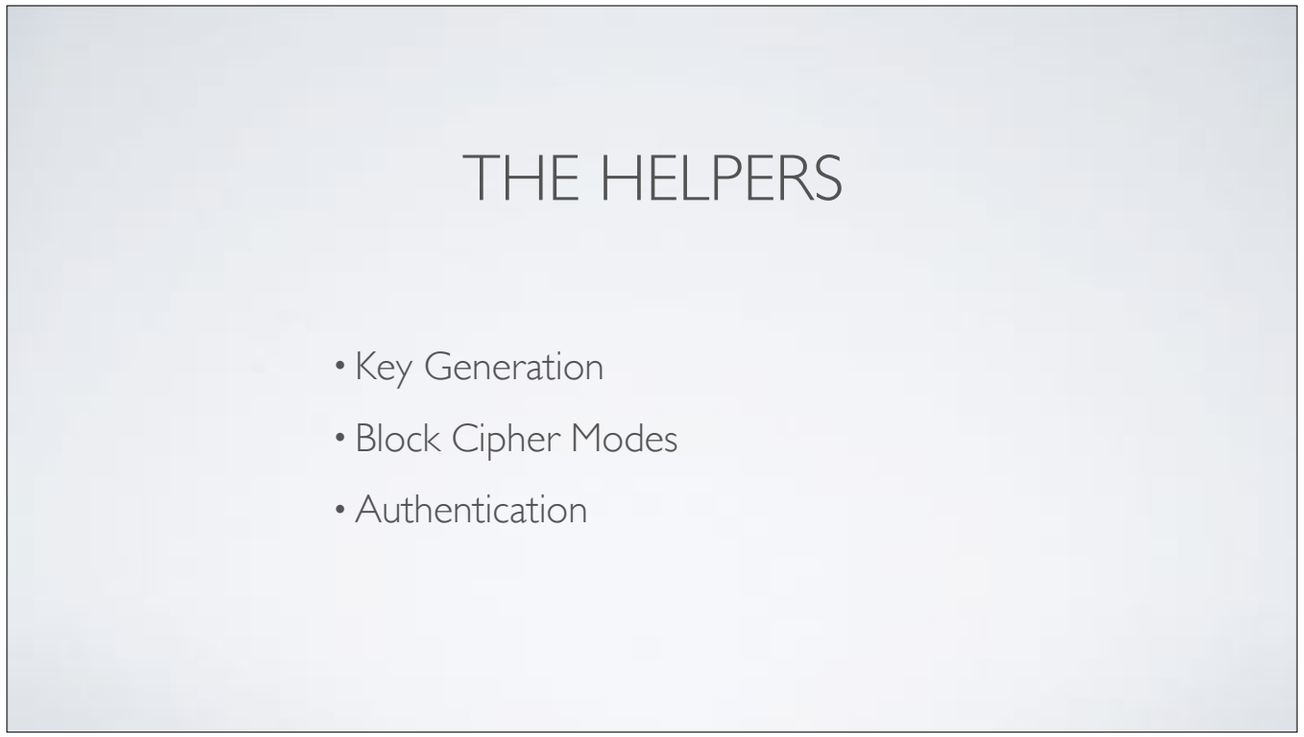
So that's how you use it. But why should you? I can say "trust me, it's right and most everyone else is wrong," and if you're cool with that, then take a nap. But if you're wary of "just trust me" from a guy who's shilling his own code, that's good. You should be. So let's talk about what it means to use AES correctly.

<build>Before we can answer that, I need to explain what AES actually is.



AES can encrypt or decrypt exactly 16 bytes of data. No more, no less. And it needs a key exactly 16, 24, or 32 bytes long. And that key needs to be effectively random.

“But, I use AES all the time to encrypt multi-megabyte files with the name of my dog.” Yes, and I’m telling you, AES can’t do that. It needs helpers to do that, and the helpers are where things go wrong.



Here are the helpers.
<build>First, you need to convert a short, non-random password into a long, effectively-random key.
<build>You need a way to encrypt more than 16 bytes.
<build>And you need a way to make sure the data wasn't modified in transit.

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
```

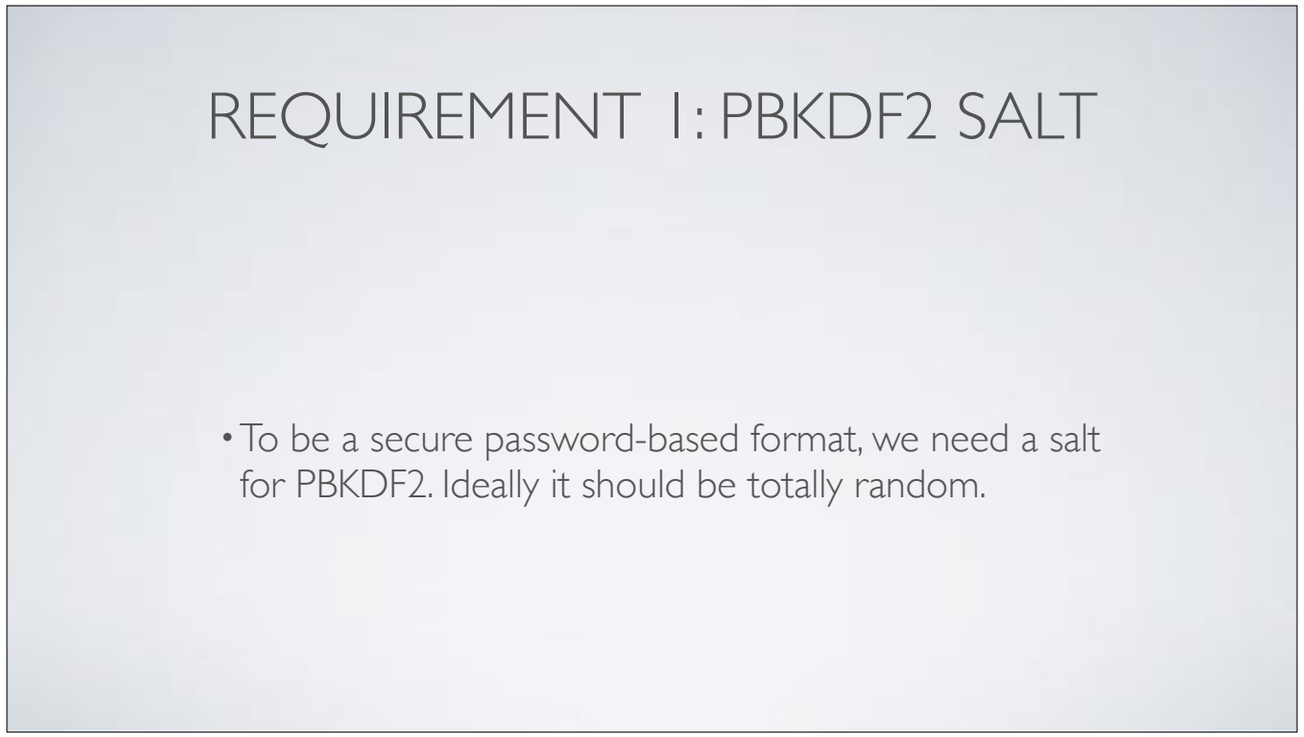
- Truncates long passwords
- Uses only a tiny part of the key space
  - Best case is ~ 0.00001% of a 128-bit key.

The most troublesome helper is key generation. Most of the code I see in the wild does it basically this way, and it's completely broken. It 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. A 128-bit key means there are 2^128 possible keys. That's a crazy huge number. And the 256-bit space completely dwarfs that.
<build> But this code throws away almost all of those keys, because it limits itself to things you can type. We need a way to map our tiny set of passwords into this enormous AES key space in a way that is indistinguishable from random. We already have a solution for that.

Use a PBKDF
(scrypt, bcrypt, PBKDF2)

A password based key derivation function. Remember PBKDF2 from the last topic? There are other choices, but PBKDF2 is good, and it's available just about everywhere. You turn a password into a key by salting and stretching it.



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

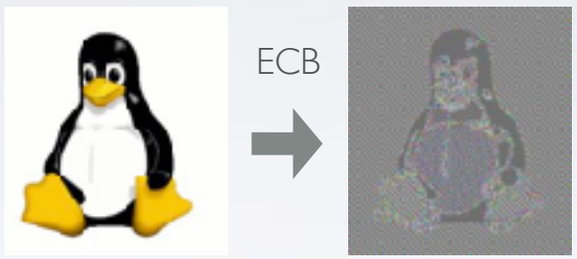
And Modes of Operation

The next helper allows AES to deal with data of any length, rather than just 16 bytes. There are lots of ways to do that.

Electronic Codebook (ECB) mode encryption

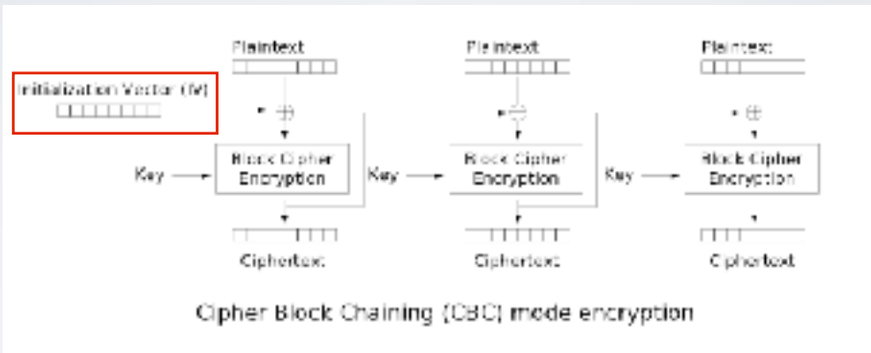
First, the most obvious approach. We'll just take each block of 16 bytes and encrypt them with the key. And then the next 16 bytes. That's called electronic codebook or ECB.

<build>It's almost always the worst possible way to encrypt data with a block cipher. In some cases it barely qualifies as encryption.



Source image by Larry Ewing <lewing@isc.tamu.edu> and The GIMP

Let's take a bitmap and encrypt it with ECB. Here's the result...
<build>Not exactly secret data. The problem is for a given key, any time the same 16 bytes show up in the plaintext, say 16 black pixels, the same 16 bytes show up in the ciphertext. Never use ECB. Buy me a beer and I'll tell you the one place ECB is useful, but it doesn't matter. Never use it.



Cipher Block Chaining (CBC) mode encryption

There are a lot of very good block cypher modes out there. But on iOS, there is really only one viable choice: cipher block chaining, or CBC. CBC breaks the patterns you saw in the last slide. There are other modes that are better, but for general use on iOS, CBC is what you want. Luckily, CBC is the default, so why am I going on about it.
<build>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. */
    CCAlgorithm 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);
```

Use an unpredictable IV, not NULL.

Right here. ...it's not optional. If you pass NULL, which almost everyone does, you wind up with a security hole in the first block. That mistake is related to how the WEP wireless protocol was broken, if you remember that old problem.
<build>You need an unpredictable IV for every message. Generally that means a random IV that you send along with the ciphertext. It doesn't have to be secret. It just needs to be unpredictable.

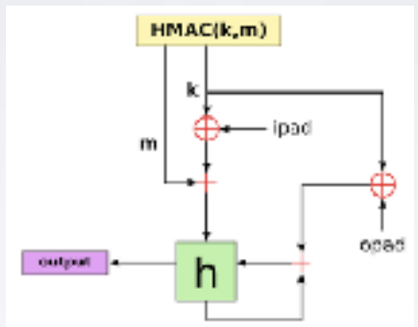
REQUIREMENT 2: RANDOM IV

- To be a secure format with CBC, 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

Just because something is encrypted, doesn't mean an attacker can't modify it, even without knowing the password. The attacks that exploit this are kind of technical and a little difficult to explain without providing some cryptography background. If you're interested, search for "unauthenticated encryption attacks" for some examples, but on this one I'm going to just ask you to trust me a little be: decrypting something an attacker has modified is dangerous.



HASH BASED MESSAGE AUTHENTICATION CODE

So how do we detect if someone in the middle has modified our encrypted data?

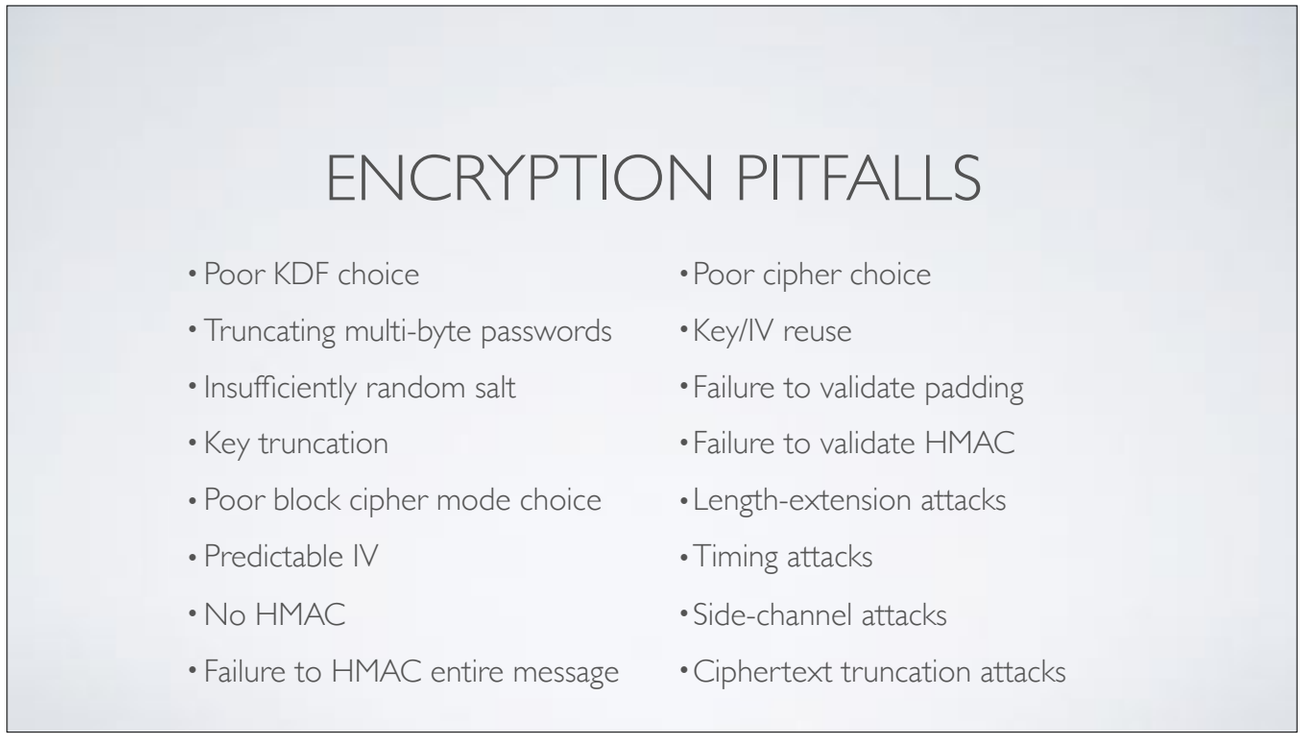
Hash Based Message Authentication Code. An HMAC is sort of an encrypted hash. You can't just use SHA-2 here, because the attacker could just calculate the hash of the new modified message. But HMAC prevents that, because you need a secret key to calculate or verify an HMAC, just like you need a secret key to encrypt or decrypt the ciphertext.

So, if you're using the CBC mode, you should also be using an HMAC with it.

REQUIREMENT 3

- To be a secure format using CBC, we need an HMAC.

That brings us to our last requirement.
Use authenticated encryption, and for CBC, that means you want an HMAC.



I've walked you through all this to give you some sense of how encryption can go wrong and what it takes to do it right. Your takeaway is that it's actually pretty hard to do it right. There are a lot of places you can screw up. Here are some of them.

ENCRYPTION PITFALLS

- Poor KDF choice
- Truncating multi-byte passwords
- Insufficiently random salt
- Key truncation
- Poor block cipher mode choice
- Predictable IV
- No HMAC
- Failure to HMAC entire message
- Poor cipher choice
- Key/IV reuse
- Failure to validate padding
- Failure to validate HMAC
- Length-extension attacks
- Timing attacks
- Side-channel attacks
- Ciphertext truncation attacks

Here are some I've personally made. Getting this stuff right is tough, even when you're paying attention.

DON'T BUILD YOUR OWN AES FORMAT

What I'm saying here is: don't build your own AES format. "But Rob, didn't you build your own AES format?" Yes. But only because there was no good solution available. You don't have that excuse. And I've spent many hours studying the problem and fixing mistakes. I take classes in crypto. I talk to cryptographers. I study the RFCs. And I spend a lot of time designing and redesigning. It's not something you slap together in an afternoon as minor detail of your app.

AES BEST PRACTICE

- Key-Derivation Function (PBKDF2)
- Proper Mode (usually CBC)
- Random Initialization Vector
- Authentication (HMAC or authenticated mode)

PRACTICAL SECURITY

- Encrypt your traffic with SSL
- Pin and verify your certs (CertificateValidator)
- Encrypt your files with ProtectionComplete
- Salt and stretch your passwords
- Use AES securely

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

github.com/rnapier/practical-security
robnapier@gmail.com
@cocoaphony
robnapier.net

Let's be careful out there.