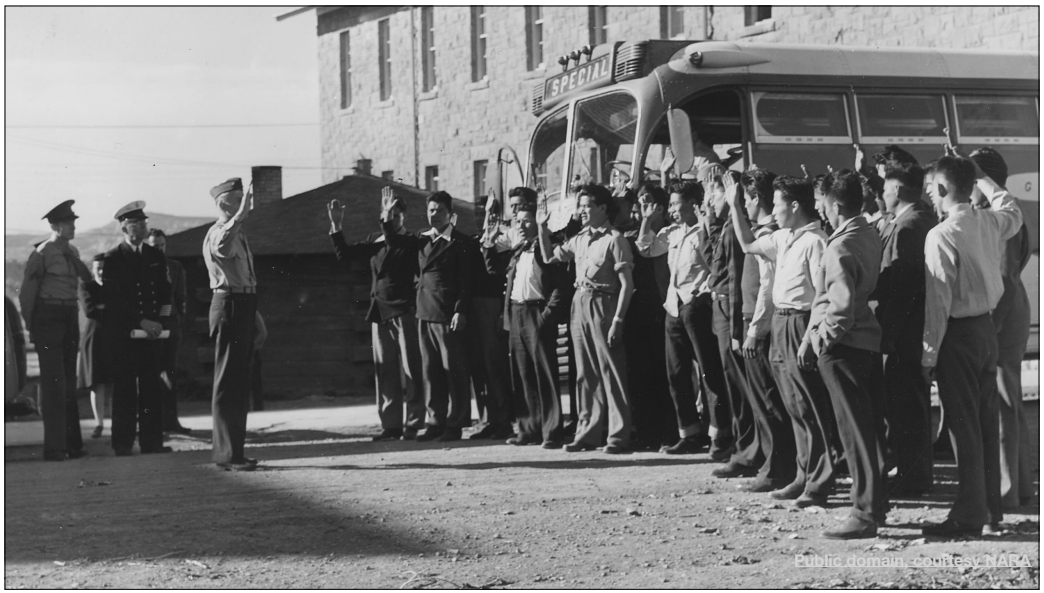
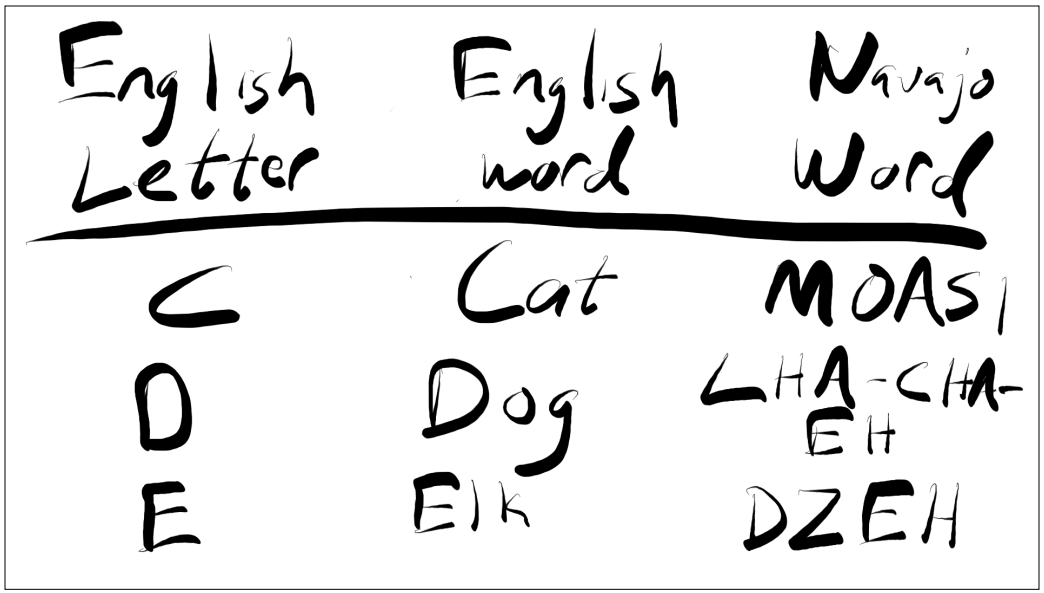

Secrets
&
Lies



In 1942 the US Marines enlisted 29 Navajo speakers to send secure messages over the radio. Eventually there were 421 of them, serving in World War II, the Korean War, and part of the Vietnam War.



Their “secret code” was more or less just “speaking Navajo” with some very basic substitutions thrown in to spell things.

Why Navajo? Because almost nobody outside of the Navajo Nation spoke Navajo or anything like it. It’s a really obscure language. In 1942, they estimated maybe 30 non-native-speakers knew it. It has complicated grammar, and uncommon consonant sounds. And there wasn’t a writing system until nearly 1940, so there weren’t any books you could study. Navajo is the most famous, but the same was true of a lot of Native American tribes, and there were Hopis, Cherokees, and others doing the same thing. It is the ultimate security through obscurity.

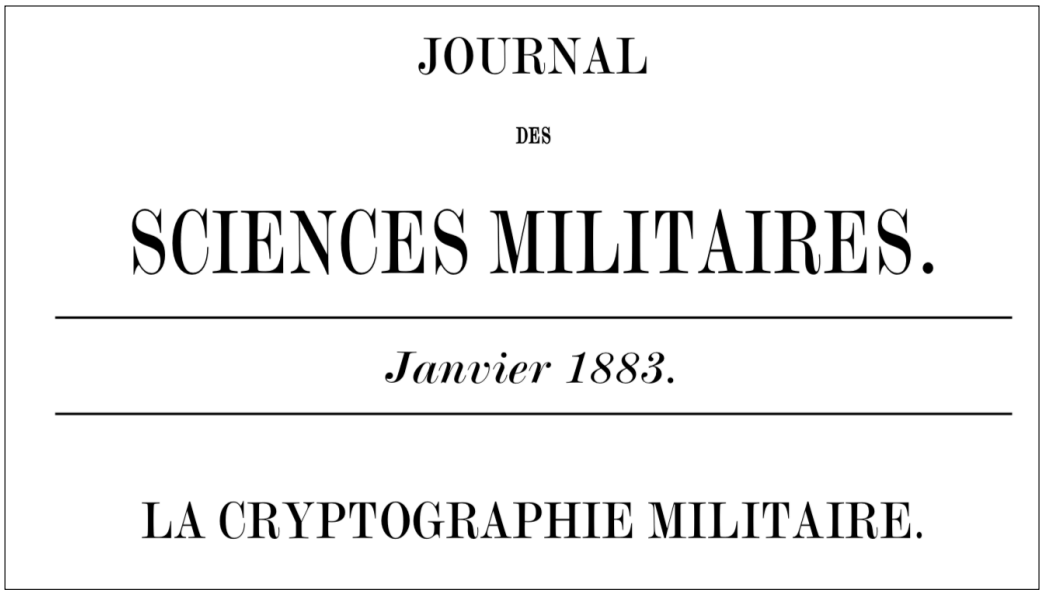
Also in 1942, the pinnacle of German cryptography was the Enigma machine, which had been in use for about 20 years, with numerous iterations and improvements. The Enigma was what we call “security by design.” While they didn’t tell anyone how they worked, the design was that the only important secret was the key. Depending on the model, the operator would program the key into the system by rearranging rotors or patching a plugboard, or however you configured it, and like any good cryptosystem, that was the only critical secret.

So we have the ultimate security by obscurity vs the best security by design of the day. How’d that work out?

Well, through three wars, we don’t believe the Navajo Code Talkers were ever deciphered. Through World War II, Enigma messages were pretty regularly cracked. Now there are a lot of caveats to that, but still…

Security through obscurity. That's what I'm going to talk about today.

So who am I? Why am I talking about this? My name is Rob Napier and I've been a Cocoa dev for over 10 years, but before that I worked in anti-counterfeiting, mostly for networking products. I did a lot of general information security work, but my primary focus was making it hard for people to steal things that would let them counterfeit our products. There's a lot of money in counterfeiting, and there are a lot of bad things you can do if you can sneak your own routers into a sensitive network, so I was dealing with very motivated and well-resourced attackers, in some cases with the backing of governments or organized crime.


JOURNAL

DES

SCIENCES MILITAIRES.

Janvier 1883.

LA CRYPTOGRAPHIE MILITAIRE.

And like all highly-trained security people, I was taught Kerkhoff's principle, first published in 1883

LA CRYPTOGRAPHIE MILITAIRE. (1883)

- The system must be practically, if not mathematically, indecipherable;
- It should not require secrecy, and it should not be a problem if it falls into enemy hands;
- It must be possible to communicate and remember the key without using written notes, and correspondents must be able to change or modify it at will;
- It must be applicable to telegraph communications;
- It must be portable, and should not require several persons to handle or operate;
- Lastly, given the circumstances in which it is to be used, the system must be easy to use and should not be stressful to use or require its users to know and comply with a long list of rules.

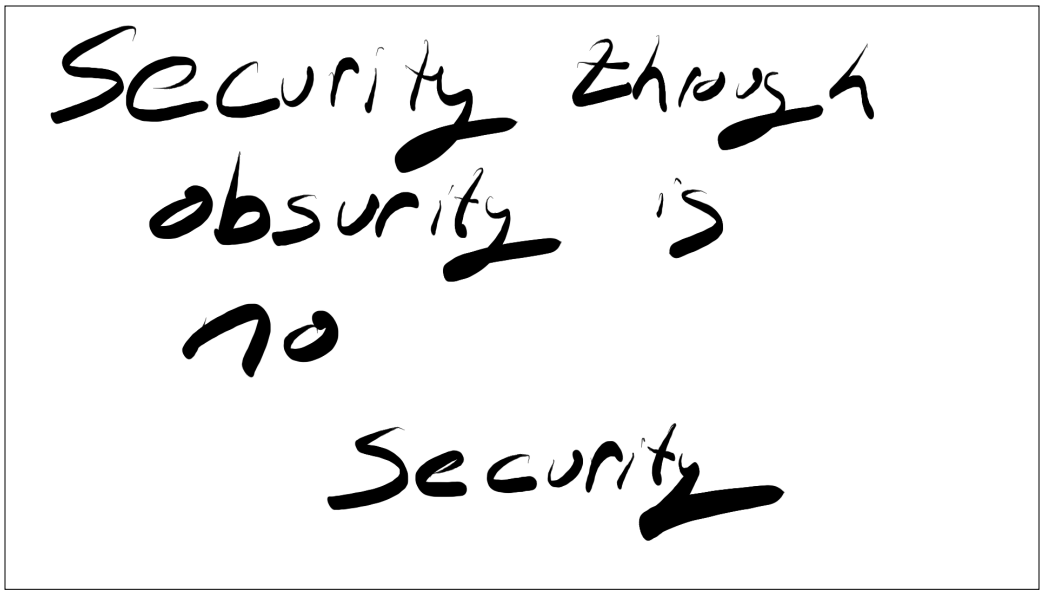
In modern terms it's generally given as

"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge."

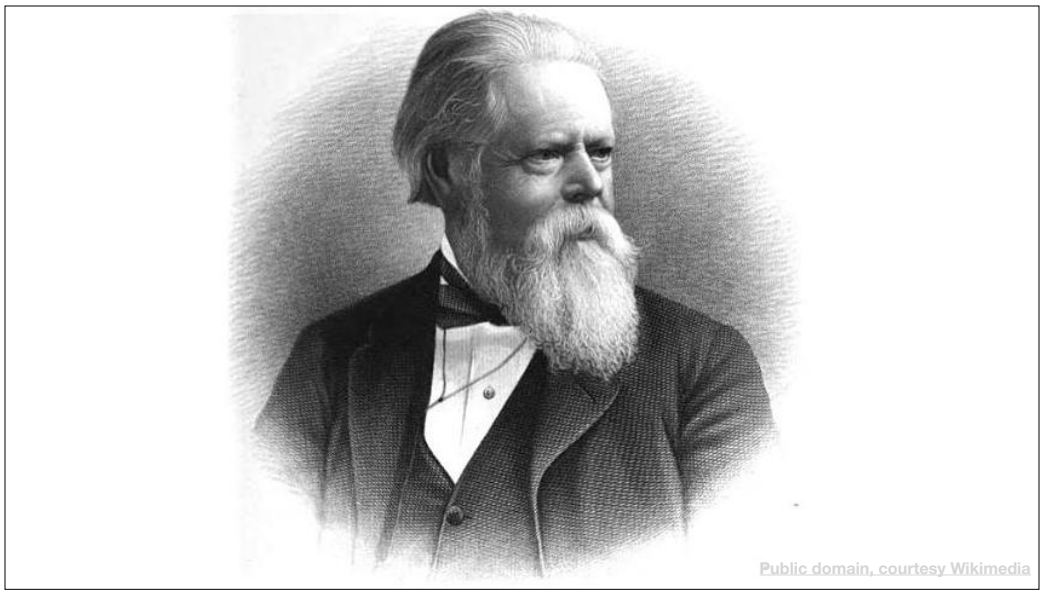
Or my preferred form, which is called Shannon's maxim:

"The enemy knows the system."

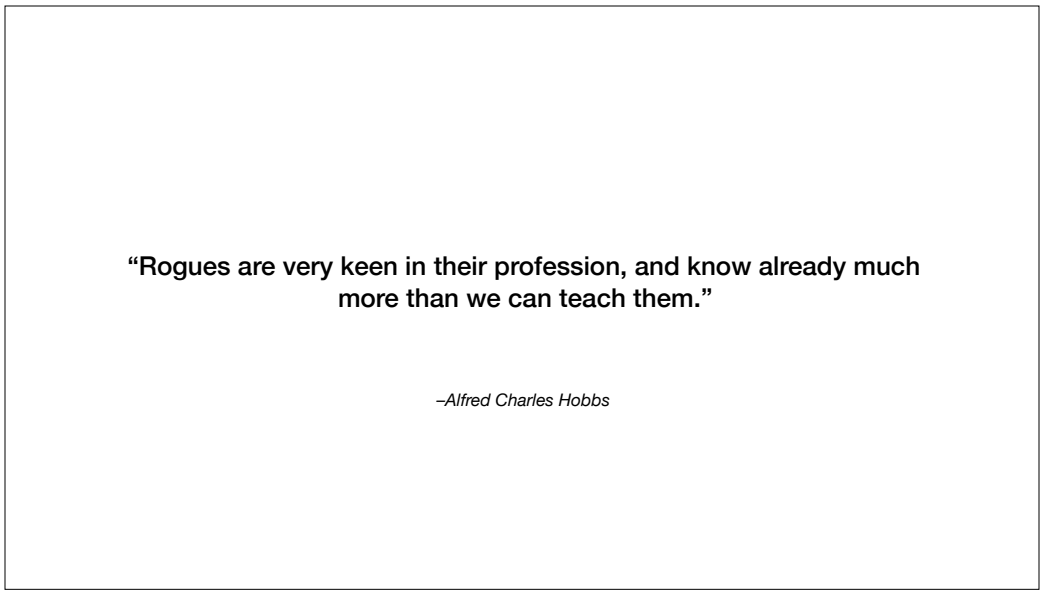
Or as it's commonly said:



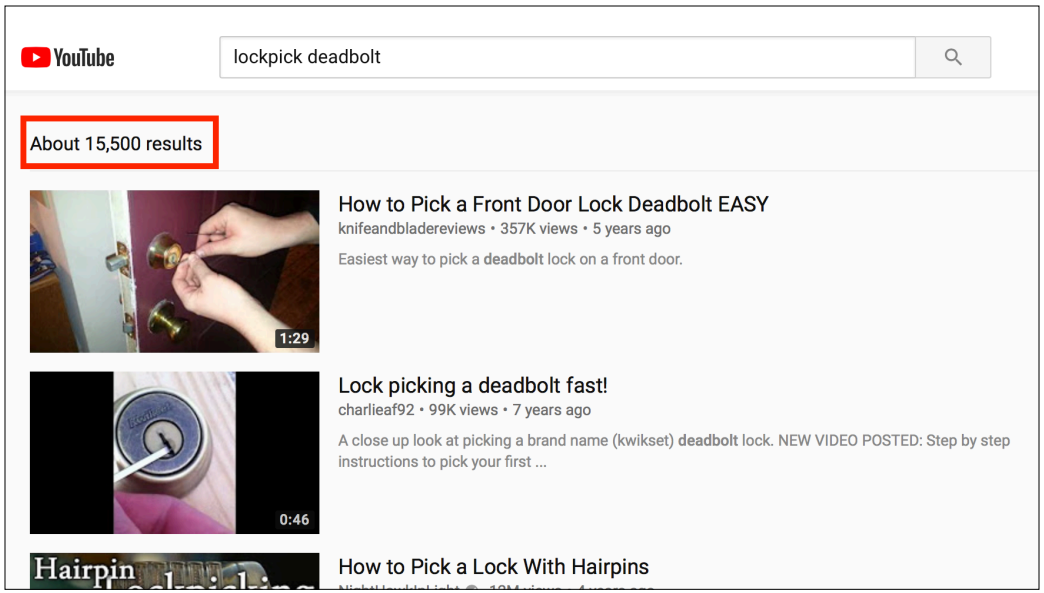
“Security Through Obscurity is No Security.”



Let me tell you another story. Alfred Charles Hobbs. At the Great Exhibition of 1851, in London, he demonstrated how to pick some of the most advanced locks of the day. He was questioned about whether it was wise to publish the weaknesses of existing locks, since of course that might make it easier for thieves to figure out how to do it. He replied



“Rogues are very keen in their profession, and know already much more than we can teach them.”
This continues to be true today. Most door locks are basically pointless. They can be picked in less than thirty seconds.



Go on YouTube, search for “lockpick deadbolt.” I’ve known locksmiths who’ll mess around for 20 minutes on a lock so you feel you’re getting your money’s worth. And that’s all by hand. Then you get into lock pick guns and bump keys. Lock manufacturers know the problems. They just don’t fix them.

A lot of institutions have a master-key system, where there are regular keys that open one door, and master keys that open all the doors.

mattblaze.org

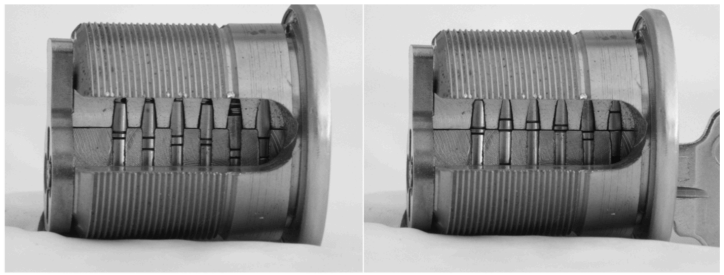


Figure 4: A master keyed pin tumbler lock. *Left:* Each of the six pin stacks has two cuts. *Right:* With the correct *change key* inserted, one of the cuts on each pin stack is aligned at the shear line. Observe that the other cut is sometimes above and sometimes below the shear line.

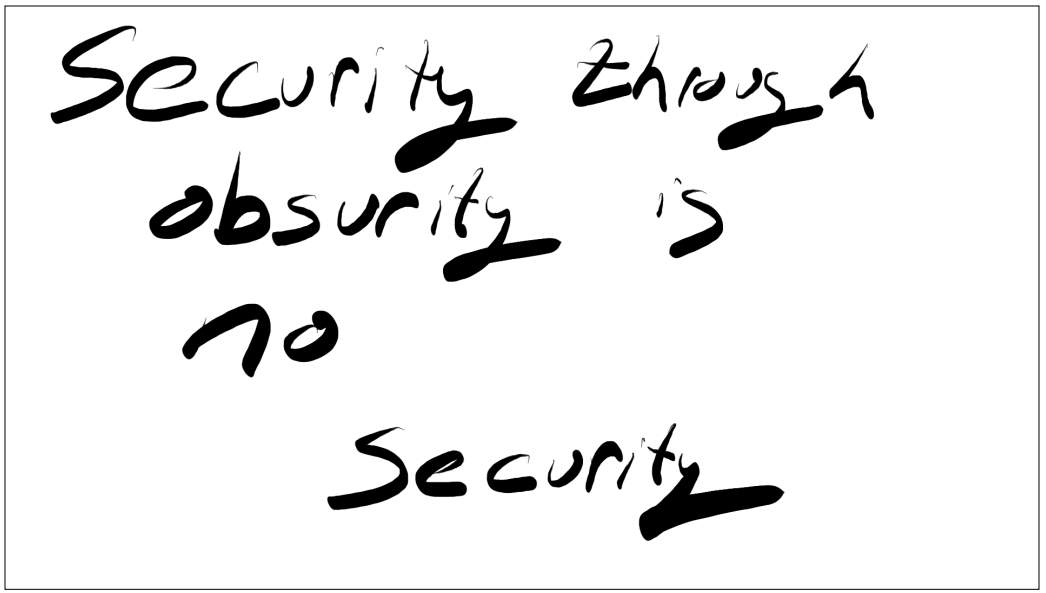
positions, although the positions will vary (rotate) from lock to lock. Both these schemes can implement a directed graph with several levels of master keys: "sub-master" keys that open a subset of locks in the system and "grand master" keys that open more[1]. The highest-level master key, which opens all locks in a multi-level system, is sometimes called the *Top Master Key (TMK)*.

Master keying has long been understood to reduce security in several important ways. First, of course, the master key represents a very valuable target; compromise of the master key compromises the entire

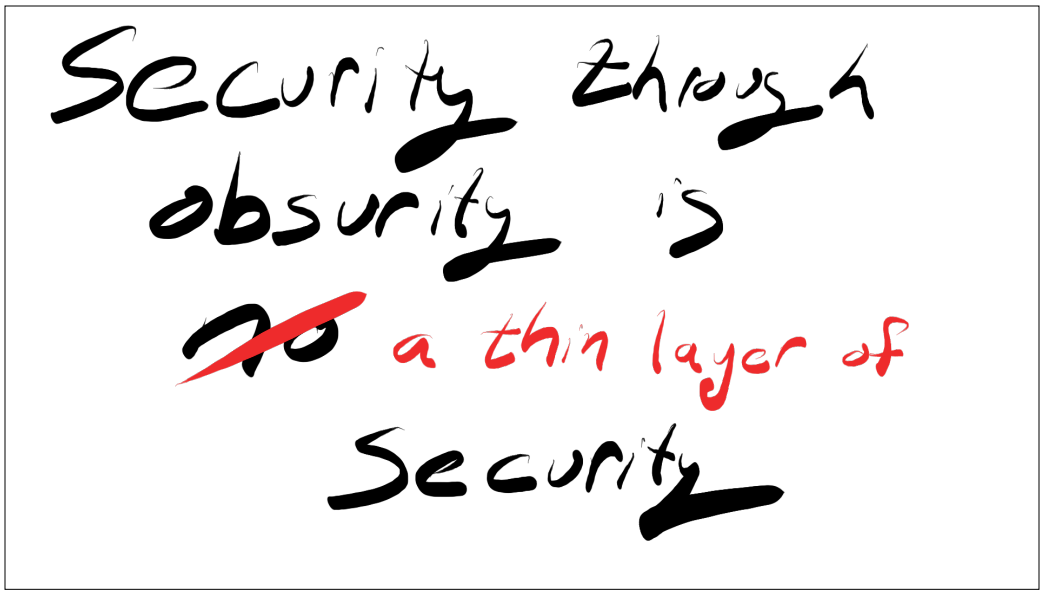
In 2003, Matt Blaze published a paper on how you could take a regular key and upgrade it to a master key. It was a pretty easy attack. Didn't require a lot of skill or time or equipment, and it basically applied to every master key system out there.
Nothing too surprising there. Vulnerabilities are found all the time. But what was really interesting was the reaction from the locksmithing community. A lot of people were really angry with Matt for publishing something that had been known to locksmiths for over a century. This vulnerability had been around for 100 years and no one had fixed it. Locksmiths knew. Thieves knew. But people who buy locks didn't know.

“Rogues are very keen in their profession, and know already much more than we can teach them.”

–Alfred Charles Hobbs

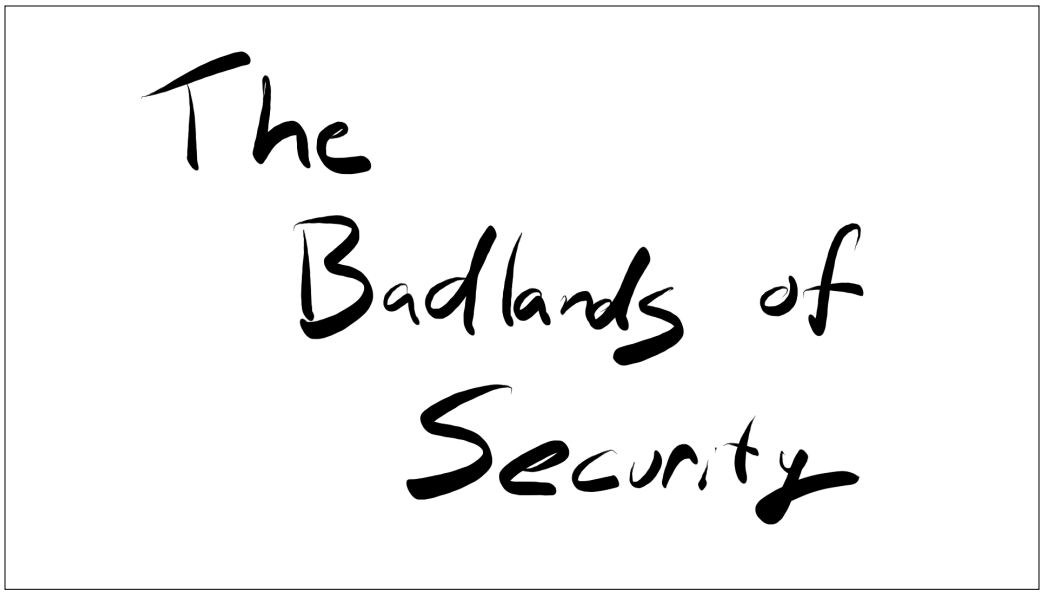


But this is a lie. Or at least not completely truthful.

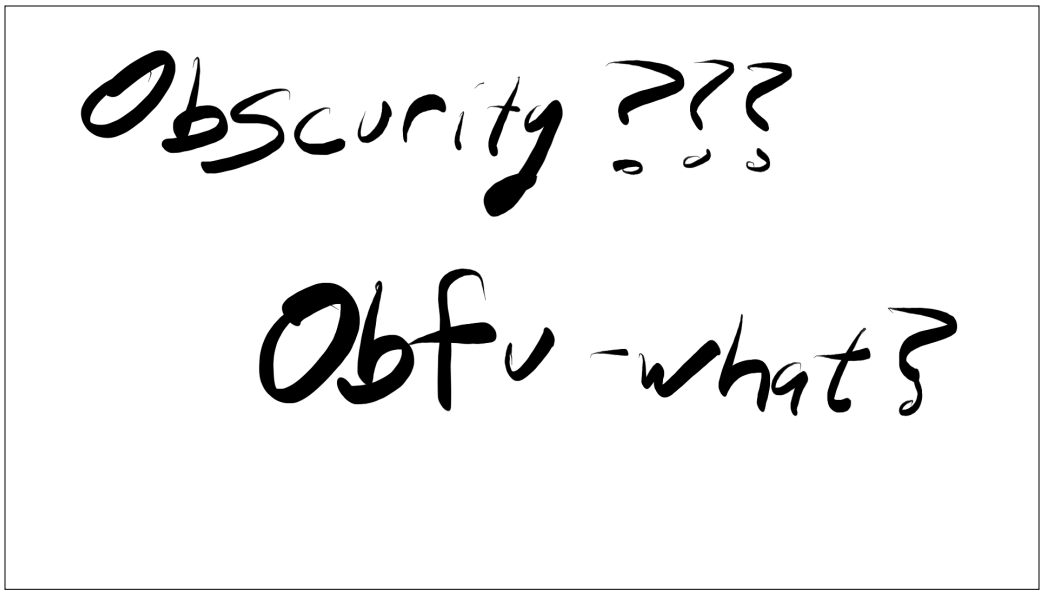


There's a big difference between those two stories. In the first, the Marines were using obscurity as a tool. We had cryptographic ciphers in World War II. Why did we use Navajo and other Native languages? Because it was faster. Encrypting documents to send over the radio took too long in the field. A Navajo speaker could translate in their head. So they were making a trade-off in a case where speed was the more important requirement.

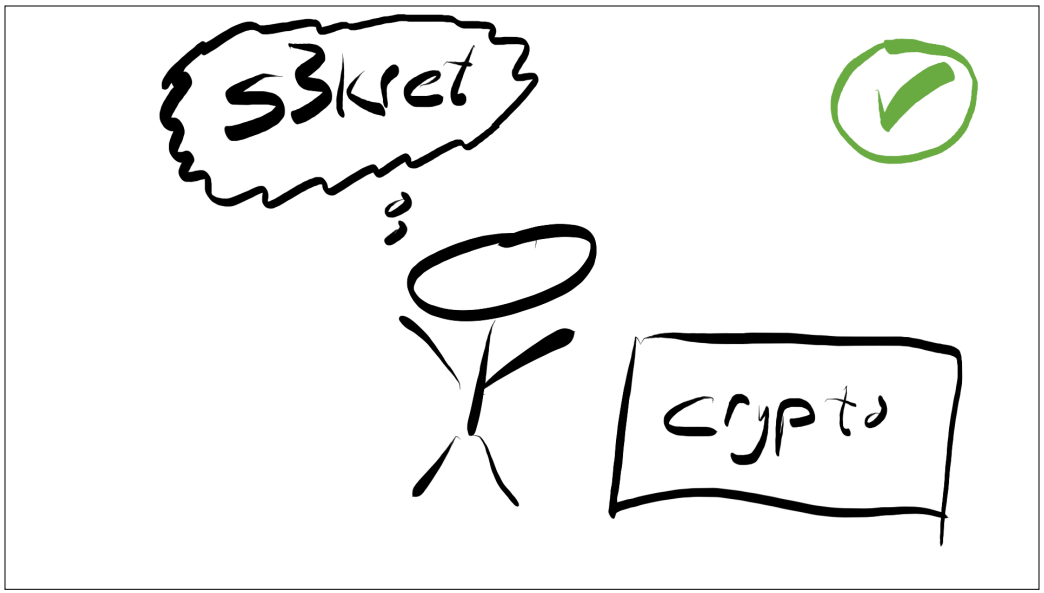
In the case of locks, manufacturers and locksmiths don't want to redesign the locks, and they want to keep the locks cheap. We know how to build locks that are much more pick-proof, but they're more expensive. Software companies are the same way. They'd rather keep vulnerabilities secret then to have to ship a new version they can't easily charge for. Or they want to avoid the hard and expensive work of designing a secure system, and rely instead on hoping no one finds the problems.



Obscurity it a tool. It's not as powerful a tool as cryptography, but that doesn't mean it has no place. Today we're going to take a trip through the badlands of security to learn about the discredited and the disparaged.
Let's talk about obfuscation.

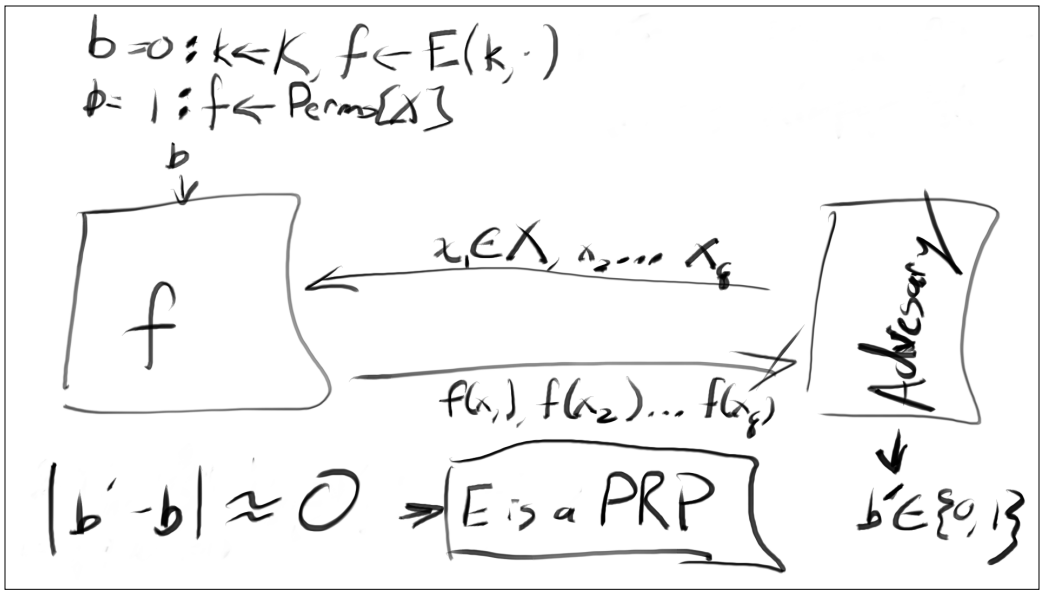


I keep using “obscurity” and “obfuscation” interchangeably. But they’re not the same. Obscurity means I just don’t tell anyone how it works. Obfuscation means making it hard to reverse-engineer the system. But they’re related ideas, and they’re very different than cryptography.



In cryptography there is some secret held outside the system. Usually that's a password held in someone's brain. There are other things it could be, but the basic rule for apps developers is that if you can decrypt something without the user's help, it's probably not actually encrypted.

That means if you store an encryption key in your app, then using AES isn't encryption. We call what you're doing "scrambling." It's just mixing it up to make it hard to read, but it's not cryptography, because there's no actual secret. Anyone who can access the app can reverse the scrambling, because all the instructions to reverse the scrambling are in the app.
Obfuscation is trying to make it harder to read those instructions.

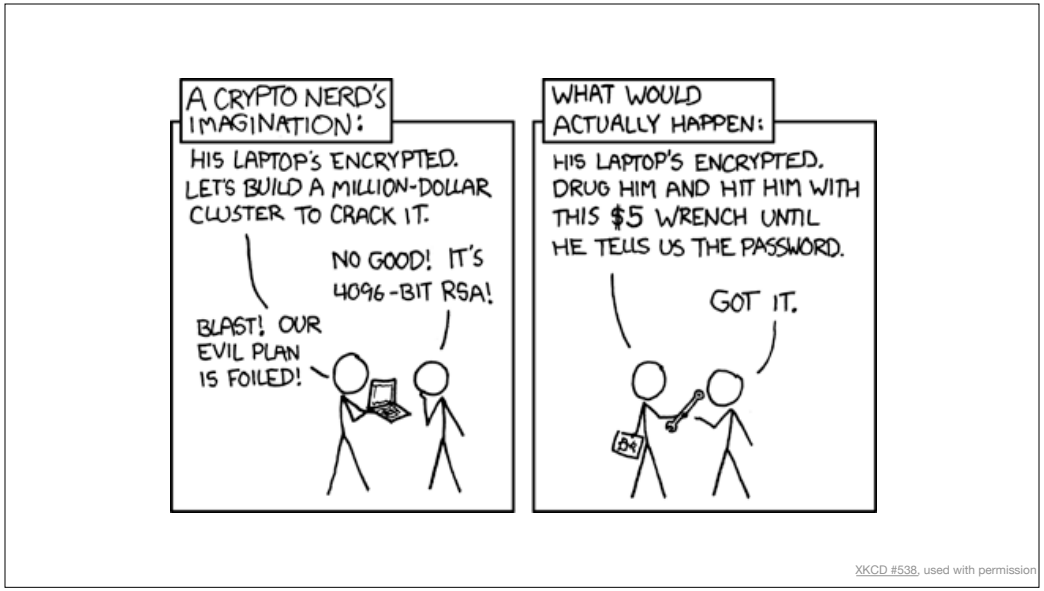


Cryptography makes strong, provable claims, about the security of a system, based on highly technical and mathematical definitions of what it means to be “secure.” They look like this.

This is often called a cryptographic game. The defender has some encryption algorithm, E, and is provided a series of messages of a given length, chosen by an adversary. For each message, the defender flips a coin, and if it’s heads they encrypt the message with a random key, and send it back, and if it’s tails, they send totally random data of the same length instead. If the adversary can guess better than 50% which messages are encrypted and which ones are random, the algorithm fails and is not a secure block cipher.

You’ll notice nowhere in there did I say the adversary decrypts even a single message. That’s not a requirement of showing a cipher is broken. You just have to show that you can distinguish from random noise.

So the security world has a kind of different meaning of “secure” than the general public. That sometimes makes it really hard for people to know when to panic. You’ll hear that SHA-1 is broken, and it is, but that doesn’t mean that everything that uses it is in immediate danger. For some uses, like in PBKDF2, SHA-1 is still totally secure in the cryptographic sense, and it isn’t a problem at all.



Of course, real-world security doesn't always care about those definitions anyway. Having a cryptographically secure cipher doesn't protect you against all attacks.

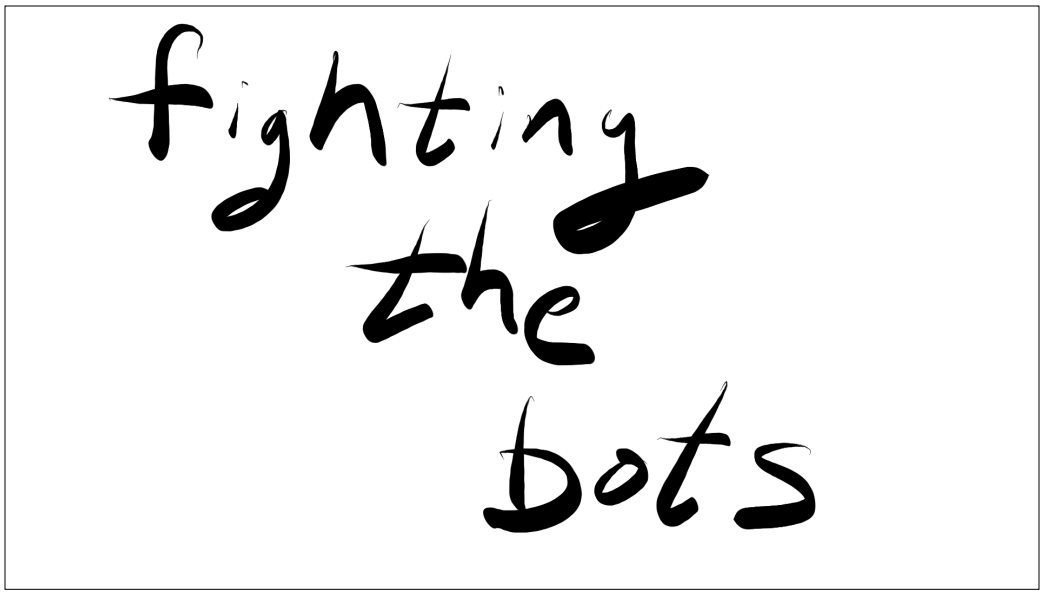
Even so, even with this attack, getting someone to tell you their password doesn't make RSA any less secure. The millions of others who use it aren't impacted.

On the other hand, if your app's obfuscation scheme is figured out, every copy of your software is vulnerable. That's the big difference.

Practical security is really just risk management. And common way we talk about risk management is “raising the bar.” I kind of hate that phrase. Technically it’s great advice, but it’s used in some really pointless ways.

The idea of raising the bar is that if it’s high enough, attackers will go somewhere else. As I said, most door locks are trivial to circumvent. And if you have windows, why do you even bother locking your door. But you do it because you hope if a thief tries the door and it’s locked, they’ll move on to the next house. And you’re probably right.

There's an old joke. I don't have to be faster than the bear. I just have to be faster than you.
But "raising the bar" only makes sense if you know what is hard and what is easy.
A six inch fence is twice a high as a three inch fence. But there's no major group of attackers who would step over the first one, but not the second. There are "attackers," if you consider anyone who is doing something you don't want, who won't step over a three-inch fence. There are attackers who will stopped by a sign. But replacing a three fence with a six inch fence is just wasting time and money.
So when you say "raise the bar," you need to be thinking in orders of magnitude. Not twice as hard. Ten times, a hundred times as hard. And you can only do that if you know something about how attackers do what they do.



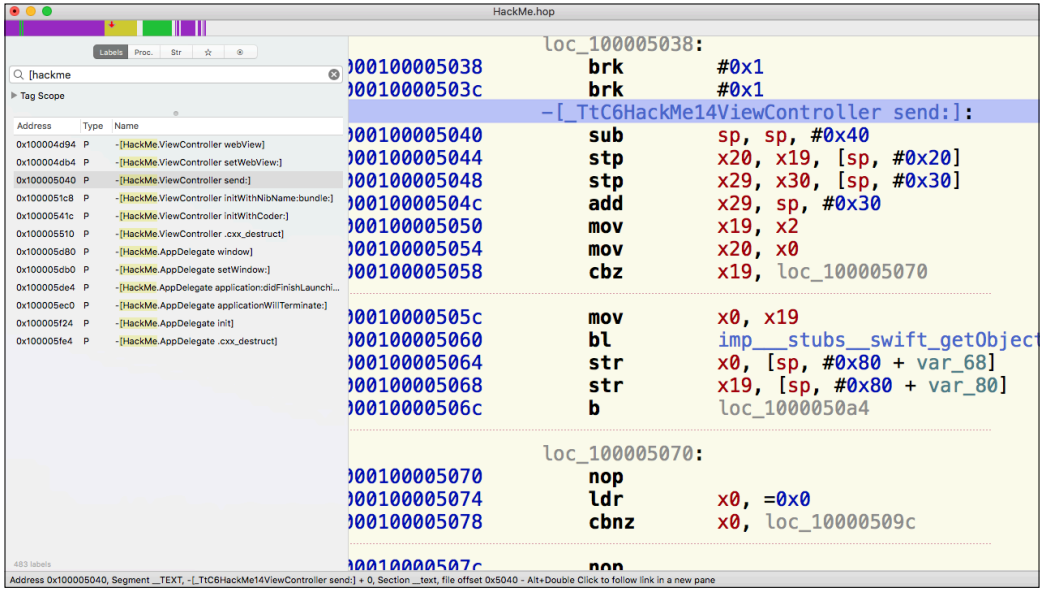
There are bots that download every free app on the App Store and at least some of the pay apps, automatically crack them on a jailbroken device, and scan them for security vulnerabilities, plaintext keys, API tokens, and passwords. They're the equivalent of walking down the hall trying doors to see if any are unlocked. This is an attack every app faces. And if you have secrets lying around in your app, you should assume they are publicly known.

```
$ strings -n 8 HackMe
UIApplicationLaunchOptionsKey
ViewController
AppDelegate
_TtC6HackMe14ViewController
v24@0:8@16
@32@0:8@16@24
@24@0:8@16
T@"WKWebView",N,W,VwebView
https://o0rq0uweih.execute-api.us-east-1.amazonaws.com/secrets/secrets?token=
Ha!U'llNevar(Find)Me
TYYLWKMUAPZHLYXJMPUW
TlqI7CIrzHp4/pY1wpRErEytlqs0IKeS4BzD9/3quqQ=
_TtC6HackMe11AppDelegate
B32@0:8@16@24
...
```

Let’s see what kinds of things you get with the most basic tools available.

strings is a simple unix tool that looks for ASCII strings.

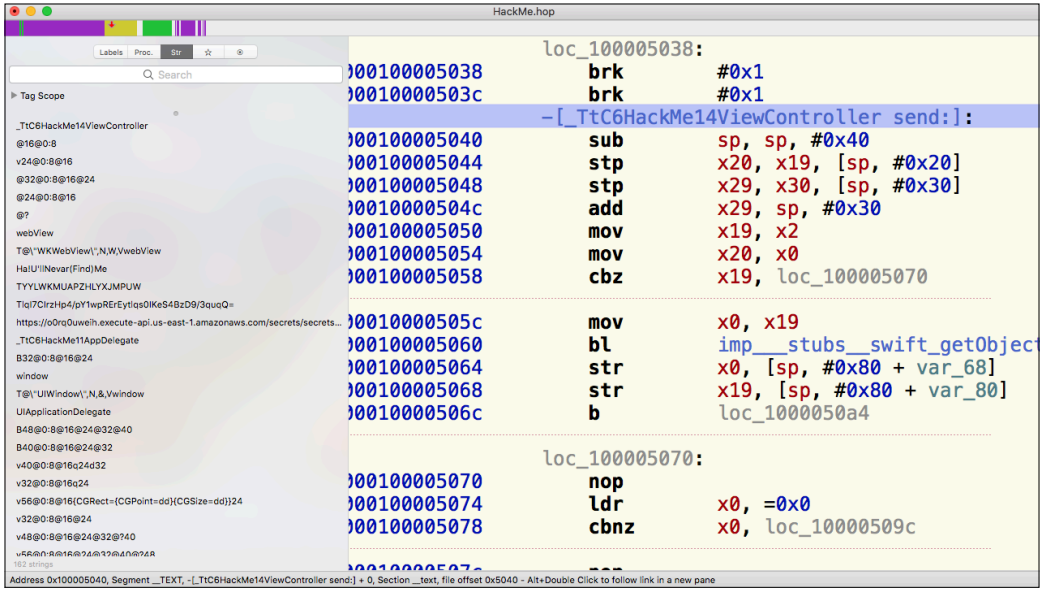
It’s usually the first thing I run. It spits out a lot of strings. I often use “-n 8” to restrict it to at least 8 character strings, but even in a trivial app it’s going to generate hundreds. But that’s ok. It’s really easy to write a script to filter out all the ObjC and Swift symbols and the like and look for “interesting” strings like these. And again, this is the simplest possible tool I could use. And there you go, I see an interesting URL , and some kind of password, and maybe an API key, and some Base64-encoded data. So I’d run a script to decode that and see if it’s anything interesting. Maybe it’s another string that someone is trying to hide. And I can see in your Frameworks directory what services you’re using on top of the URLs, so that helps me figure out the API keys. So with that, and I’m just taking about strings and a little python script, I can create a little dump of possibly interesting things for a human to look at later. I can make a database of possible API keys for when I need some S3 storage that gets billed to someone else.
A lot of modern hacking is this. Just scanning everything, and looking for easy targets. It’s walking down the hall and trying all the doors to see if they’re locked. It’s not glamorous. It’s not spycraft. For a lot of you, this is going to be your most common attack. Automated scripts. So locking your door, even though, as I said, most door locks are pointless, is still worth it so they



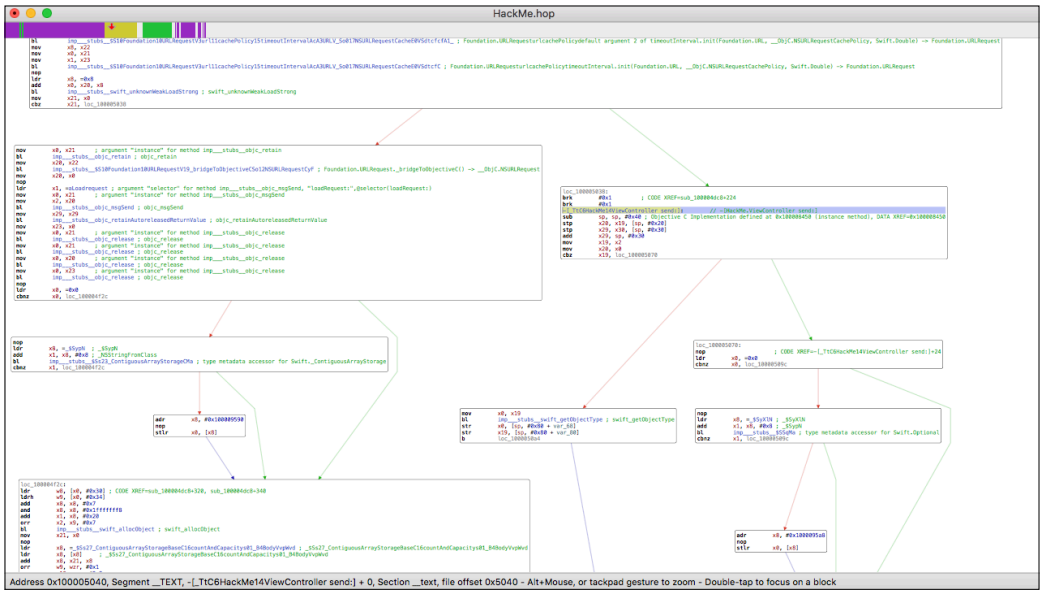
Let’s say your attacker is trying just a little harder. They’re actually interested in your app. What might they use? My favorite reverse engineering tool is Hopper. It’s $100 for a decent disassembler, decompiler, and debugger. If you have more money, the gold plated premium tool is called IDA Pro which can debug iOS apps on the device and has a very nice decompiler. But we’re talking thousands of dollars. But I don’t attack systems professionally. I defend them. So I use Hopper.

What’s a program look like in Hopper?

You can quickly get to all the symbols.

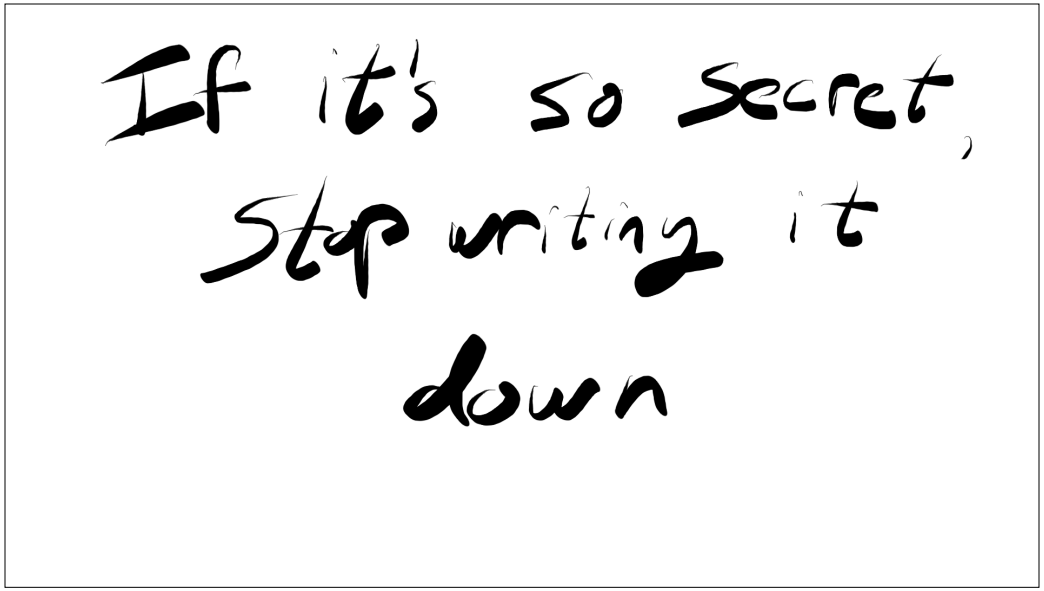


And strings are available. And here I get a disassembly with a lot of helpful comments and annotations.



And I can see all the calling structure to help you find your way around.

and even this kind of pseudo-ObjC decompilation. Here you can see it loading an URLRequest in a webview. This is really powerful against ObjC. I don't want you to think that Swift can't be reverse engineered. It totally can be. But ObjC is a reverse engineer's dream. But that includes anywhere that your Swift calls Cocoa, or relies on the ObjC runtime. Which in most apps is a lot of places. Objective-C message passing includes a lot of strings, and prevents a lot of optimizations. And aggressive optimization is definitely the most effective form of code obfuscation you can get for free.



So what should you do to reduce your risks to these simplest tools, the most automated tools? The first thing you should do is avoid sensitive strings. If you can keep them out of your app entirely, that's best. But sometimes you need to store something locally. There are three main versions of that.

```
let key = "0rlsRU5vjPvHUad70ysqX67k4DN50/sNYRD5zqnCLbg="
```

```
echo 0rlsRU5vjPvHUad70ysqX67k4DN50/sNYRD5zqnCLbg= | base64 -D | xxd -i
```

```
let key = Data([
    0×d2, 0×b9, 0×6c, 0×45, 0×4e, 0×6f, 0×8c, 0×fb, 0×c7, 0×51, 0×a7, 0×7b,
    0×d3, 0×2b, 0×2a, 0×5f, 0×ae, 0×e4, 0×e0, 0×33, 0×79, 0×d3, 0×fb, 0×0d,
    0×61, 0×10, 0×f9, 0×ce, 0×a9, 0×c2, 0×2d, 0×b8
    ])
```

First, there are Base64 encoded strings. You should never store sensitive information in Base64 because there's never a reason to. Base64 is just a way of encoding raw data. So just store the raw data. Say you have this key. And of course it's going to show up in string output like a sore thumb.

As a start, decode it back to raw bytes, and then run it through one my favorite tools, xxd. The -i option makes it dump its input a a comma-separated list of hex values. It's designed for C, but Swift is identical, so you can just cut and paste.

This doesn't show up in string lists. But….

```
$ otool -d HackMe
HackMe:
Contents of (__DATA,__data) section
0000000100009538  00000000 00000000 00000000 00000000
0000000100009548  00000000 00000000 00000000 00000000
0000000100009558  00000000 00000000 00000000 00000000
0000000100009568  00000020 00000000 00000040 00000000
0000000100009578  456cb9d2 fb8c6f4e 7ba751c7 5f2a2bd3
0000000100009588  33e0e4ae 0dfbd379 cef91061 b82dc2a9
0000000100009598  00000000 00000000 00000000 00000000
00000001000095a8  00000000 00000000 00000001 00000000
00000001000095b8  00006540 00000001 00000000 00000000
00000001000095c8  00000000 00000000 00000000 00000000
00000001000095d8  00000000 00000000 00000000 00000000
00000001000095e8  00000000 00000000 00000000 00000000
00000001000095f8  00000000 00000000 00008408 00000001
0000000100009608  00000008 00000000 00000000 00000000
0000000100009618  00000000 00000000 00000000 00000000
0000000100009628  00000000 00000000 00000000 00000000
0000000100009638  00000000 00000000 00000000 00000000
0000000100009648  00000000 00000000 00000000 00000000
```

It does show up pretty blatantly in the data section. This is better than the strings section, and it's a little harder to find things here. But you'll notice it's surrounded by zeros. Those are mostly placeholders for lazily initialized global data and padding. So it's better than a string, but it's still a problem. We'll do better later.

```
let englishKey = "Ha!U'llNevar(Find)Me!I’mSoSekret"
```

How about if you really do have an ASCII string? What about this?

There’s no good reason for a string like this. I’m looking at the length, and it’s exactly 32 characters. So, I’m betting this is an AES-256 key.

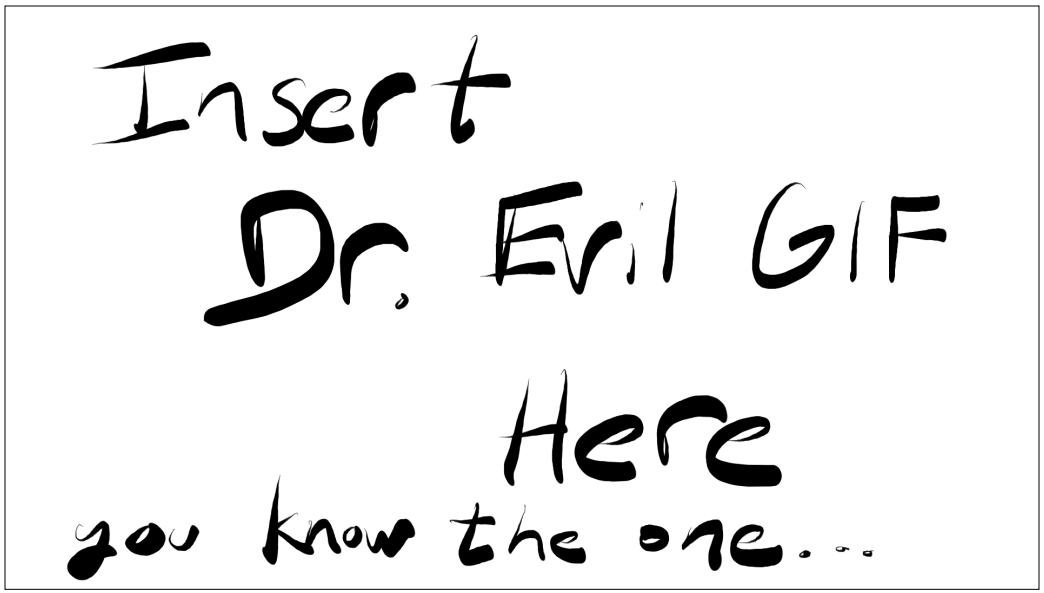
```
head -c32 /dev/random | xxd -i
```

```
let aesKey = Data([
    0x4e, 0x5a, 0x88, 0xec, 0x22, 0x2b, 0xcc, 0x7a, 0x78, 0xfe, 0x96, 0x35,
    0xc2, 0x94, 0x44, 0xac, 0x4c, 0xad, 0x96, 0xab, 0x34, 0x20, 0xa7, 0x92,
    0xe0, 0x1c, 0xc3, 0xf7, 0xfd, 0xea, 0xba, 0xa4
    ])
```

If you are using an AES key, it should be random. Totally random. Not a string of random letters. 32 totally random numbers, each ranging from 0 to 255. If you want to create an AES-256 key, this is how you should do it.

Read 32 random bytes out of /dev/random, and pipe them into xxd -i. And then store it as Data.

Now your key is a lot stronger, and you don’t have to worry about strings. When I say a lot stronger, I mean 100 trillion times stronger.



There are about 96 possible characters you can easily type on the keyboard, so if you type them totally randomly, every possible thing you could type with perfectly equal likelihood, then there are 96^32 possible keys. That's roughly 10^63. Doing it this way with 32 random bytes rather than 32 random characters, you get 256^32, or 2^256. That's about 10^77.

10^77 is 100 trillion times larger than 10^63. 100 trillion times strong than the best possible key you could get with characters, and again I'm assuming you choose your characters totally randomly, which you didn't. So that's the rule. AES keys should never ever be something you can type. And if you absolutely must put them in your code, store them as Data. Ideally do not put them in your code at all.

The other nice thing about this is that it will greatly reduce the number of people who will be able to memorize the key just by looking at the code briefly. Again, this isn't a strong security technique. Anyone with access to the code who wants to steal the key can it, and there are actually quite a lot of people who can very quickly memory 32 random numers. But it's nicer if it's not so obvious.

```
TYYLWKMUAPZHLYXJMPUW
```

That leaves API keys, which often look like this

They really are strings, and there really isn't anything you can do to change that.
First, if this is a third-party API key, I recommend you don't let apps directly connect to third party services if you can help. it Proxy all requests through your server. You gain a lot of control this way. It gives you a way to control access to your backend services and shut down attacks. If nothing else, it gives you visibility when someone is attacking your system, which may be a lot harder if the app is talking to some third party service directly.
If you can't do that, I suggest fetching API tokens from your server and not storing them. This at least gives you an chance to change API tokens if you discover they've been leaked. And again, it moves them out of the code, so they're not subject to automatic scanning.
But maybe you have to put it in the code. Just making it a Data doesn't help, because like I said, 20 ASCII values are going to show up in strings no matter what type they.

```
                        head -c 20 /dev/random | xxd -i

let mask = Data([
    0×77, 0×a6, 0×7f, 0×09, 0×1f, 0×74, 0×2d, 0×f3, 0×b6, 0×88, 0×b6, 0×ff,
    0×05, 0×c3, 0×9f, 0×3c, 0×02, 0×3a, 0×b1, 0×ac
    ])


                         print(zip(api, mask).map(^))


             let masked = Data(
                 [35, 255, 38, 69, 72, 63, 96, 166, 247, 216,
                  236, 183, 73, 154, 199, 118, 79, 106, 228, 251]
             )

            let demasked = String(bytes: zip(masked, mask).map(^),
                                  encoding: .utf8)!
```

For the first time, we really do need real obfuscation. Here's one simple way to do it. Don't scribble this down. I'm going to give you a tool to do it later.

<build>Create a totally random mask that is the same length as the API key. Read from /dev/random, pipe through xxd, **<build>**make a Data.

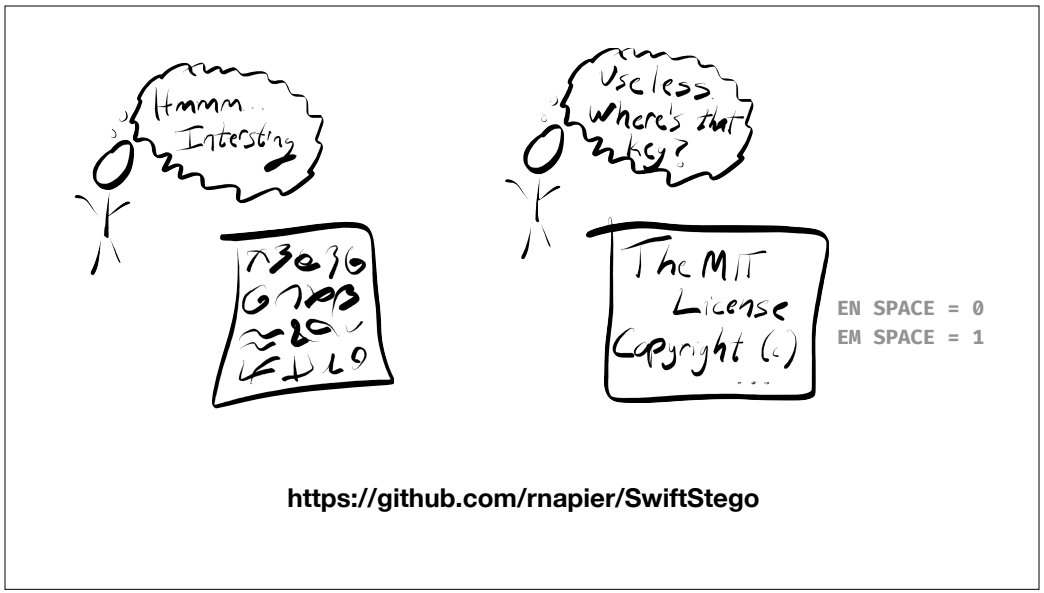
Now you need a masking function.**<build>**

All this is doing is xoring the data you want to hide with the mask.

That will give you something like.**<build>**

And you can demask that like this.**<build>**

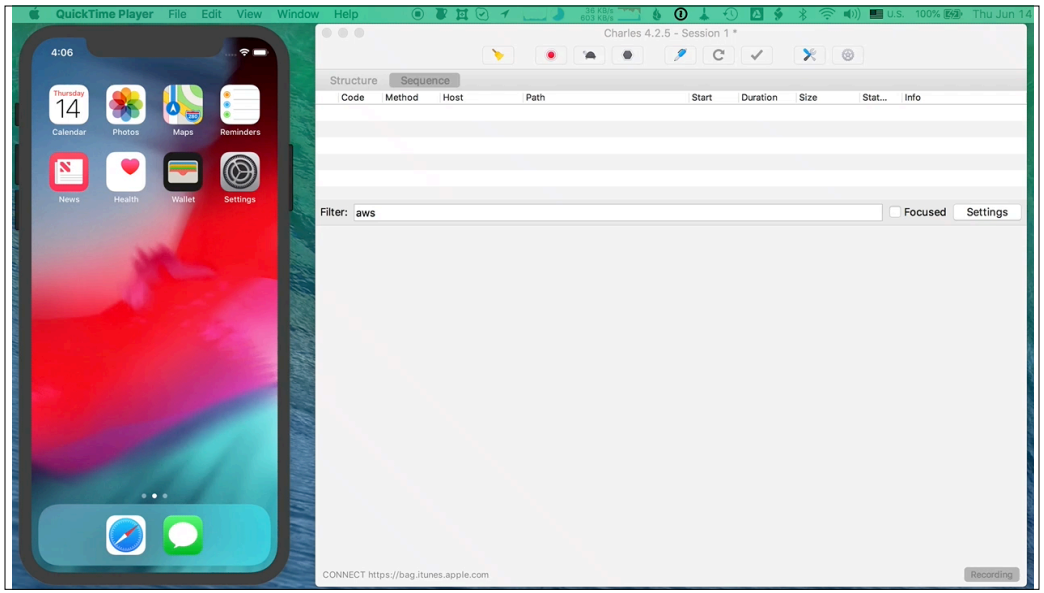
Both the mask and the masked API key will show up in your data section as random bytes because that's what they are. They only become a string when you put them together.



There are better ways to do this, but the code gets too complicated to throw on a slide. So I've started work on a new toolset, called SwiftStego. Steganography is hiding messages in other, unrelated data. The idea isn't to prevent an attacker from deciphering the secret. That's what cryptography does. It's to keep the attacker from knowing the secret exists at all. There's a real danger in these kinds of tools, since if everyone uses the same tool, then attackers will know where to look. My hope is to build some tools that are varied enough and customizable enough that they can be effective and easy to use, while each implementation is different enough that it's time consuming to build a scanner. We'll see how that works out, but watch this space.

Scanning for strings is so easy that you should expect attackers to do it to every program you ship on App Store. It's just too easy to automate, looking for well known API keys, or strings that are exactly the length of an AES key, you should just assume that anything that looks like that is going to be automatically discovered.

A step up from that is sniffing network traffic. It's not as easy to completely automate as string scanning, but it's still not very hard. I'm going to show an example using the simulator because it's easier to screen capture everything, but this works almost exactly the same using a device.



[Video - 1m]

This is using the Charles Proxy which costs about $50 and it's available on Mac, Windows, and Linux. It's my favorite for most of this kind of work, but there are a bunch of other tools and some are free like Wireshark, which I use for non-HTTP traffic. This is not hard. What can you do about it?

The reason Charles can decrypt HTTPS traffic is because it injects an extra certificate into the device's trusted key store. That's a supported feature of iOS. It's used by big organizations who have their own internal trusted certificates. It doesn't require jailbreaking or anything weird on the device. Just standard configuration. But once you install your own trusted certificate, then you can intercept requests, decrypt them, re-encrypt them to the real server, and then do it again in the other direction. This is called a Man In the Middle attack.

It's not that difficult to do and you can at least semi-automate it.

So what can you do about it? Certificate pinning. I'm not going to go into the details here. I have a whole talk on just this subject, but the idea is that you don't really need to trust every certificate that Apple trusts. Apple trusts a lot of certificates.

A Lot of Trust

You Expect...	But Also...
• Verisign	• Amazon, Cisco, Apple, ...
• Network Solutions	• US, Belgium, Germany, Netherlands, Switzerland, Turkey, …
• Thawte	• Actalis, Atos, Buypass, Certigna, Certinomis, Chambers of Commerce, E-Tugra, Echoworx, Izenpe, QuoVadis, Starfield, …
• DigiCert	
• Digital Signature Trust	

http://support.apple.com/kb/ht5012

In iOS 11, there are over 170 certificates in the root store. You probably buy your certs from one provider. Maybe two? So there's no reason to trust all the rest, and most importantly there's no reason to trust random other certs that have been added to the root store.

Certificate pinning is when you have some small number of certificates, and you just trust those for your app.

```
extension CertificateValidator: URLSessionDelegate {
    func urlSession(_ session: URLSession, didReceive challenge: URLAuthenticationChallenge,
                    completionHandler: @escaping (URLSession.AuthChallengeDisposition,
                                                  URLCredential?) -> Void) {
        let protectionSpace = challenge.protectionSpace

        if (protectionSpace.authenticationMethod == NSURLAuthenticationMethodServerTrust) {

            if let trust = protectionSpace.serverTrust {
                SecTrustSetAnchorCertificates(trust, trustedCertificates as CFArray)
                SecTrustSetAnchorCertificatesOnly(trust, true)

                var result = SecTrustResultType.invalid
                let status = SecTrustEvaluate(trust, &result)
                if status == errSecSuccess {
                    switch result {
                    case .proceed, .unspecified:
                        completionHandler(.useCredential, URLCredential(trust: trust))
                        return

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

        // Something failed. Cancel
        completionHandler(.cancelAuthenticationChallenge, nil)
    }
}
```

The code to do it is isn't very hard, but it's a little fiddly and uses this weird C API, but I've wrapped it all up in a one-file library.

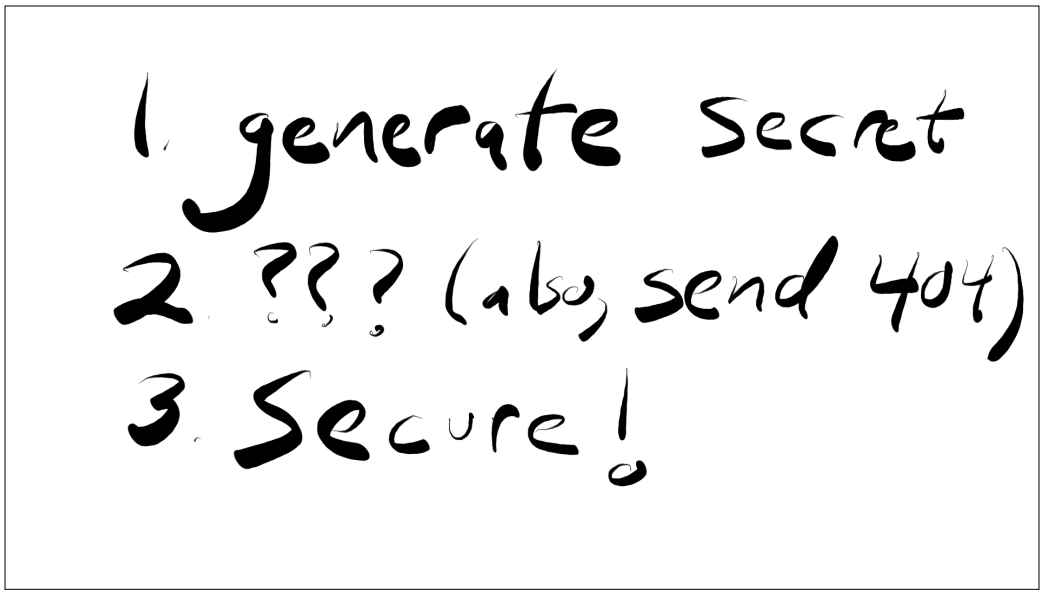
```
guard let url = Bundle.main.url(forResource: name, withExtension: "cer") else {
    preconditionFailure("\(name) not found")
}

try! validator = CertificateValidator(certificateURL: url)
session = URLSession(configuration: .default, delegate: validator, delegateQueue: nil)
```

https://github.com/rnapier/CertificateValidator

So this is something you can more or less drop into your project with the root public key you want, make it the delegate to your URLSesion, and it'll do the rest. The link is here. It'll be on the list of links for this talk.

Pinning isn't just to protect you from your users. It's a good thing to protect your users too. There's really no reason to trust so many root certificates. Any one of those 170-odd roots could be compromised, and then your users would be at risk. So pinning is just a good practice.
If you do use it for obfuscation purposes, though, don't put the certificates in plaintext on the disk. That just makes them really easy to replace on a jailbroken device. I'd treat the certificate as sensitive information and obfuscate it like we've talked about already. Generally encrypt it and store the key split up like an API key.

When dealing with a network service, I also recommend adding an secret key to all your API calls The goal is to hide your API from bots and to make manual investigation with Paw or Postman hard.



The simplest form of this is really simple. Generate a single, random, hard-coded token for your system. Require it for any request, authenticated or not. If it isn't there, return 404. Not 400 or 401. Don't tell the caller it's an error and they might want to investigate further. 404. Nobody's home. Force an attacker to actually reverse engineer your app. Don't let them just probe the server by hand. Make it hard.

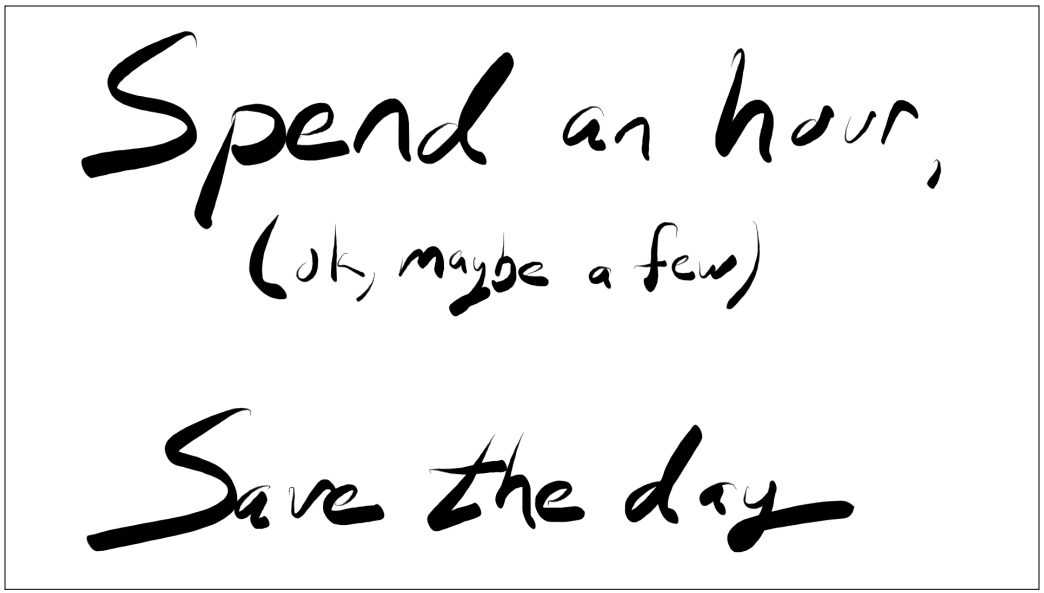
Of course you can make these tokens more complicated. You can also hash the secret together with the full URL or the full payload, so that the secret key is never the same. Of course you can use timestamps, but I'd be careful about that. It can break things if the user's clock is wrong.

People will tell you that there's no point using AES encryption over HTTPS, but it can be surprisingly effective.

True story. I was hired to develop a client for this high security API. I had the full source code for a working Android client, and my job was to build the same system for iOS. All the requests and responses were AES encrypted with a per-session key. So it's really just obfuscation, since once you connect, you can fetch the key.

But here's the thing. If you got anything wrong in the request, it didn't send you an error. It sent you a garbage response, that looked just like a normal response, but you couldn't decrypt it because it was gibberish. So you didn't know if you had made a mistake in your decryption logic, your encryption logic, or in the request itself.

It took me most of a week to reverse engineer this thing. I had the full source code for a working implementation, and I'm pretty good at building security protocols and have done a lot of it. And it still took probably 20 hours to figure this thing out. Don't let anyone tell you obfuscation can't work.



Ok, so what if the attacker is going to spend some time on your app. They're going to do more than the basics. They're going to run it on a jailbroken device, and they're going to use a serious debugger like IDA Pro. They're going to print out the values of your keys. They're going to modify your app so where you have an if statement that checks if something is valid, they'll just patch it to skip the check or jump if not equal rather than jump if equal. That's generally a one-byte change. So what do you do? For most of you, not all, but for most of you, the answer is "nothing." You've lost at that point. Your team is very unlikely to have the expertise to slow that attacker down by more than a few minutes, and you probably shouldn't spend the effort or add the complexity that it would take to do much more. I hate to be depressing, but that's just the way it is.

But here's what I recommend for most teams. Attack your app yourself using the tools we've discussed. Spend a few hours on it, maybe a day. Knowing what your secrets are and where they're hidden, see if you can find them without searching for the exact secret.

Check out your network traffic in Charles. Are there unencrypted request? Is the certificate pinning working or can you circumvent it it.

Build your app for release and archive it. Open it up in Hopper and poke around. What do you see? Is it obvious where to disable the security checks. Are there keys just lying around in the strings section?

```
xcrun simctl launch booted net.robnapier.HackMe --wait-for-debugger
```

And finally, build your app in Release mode and run it on the simulator. Use Hopper to debug it. Why the simulator? Because Hopper can only debug Intel executables. But you can launch an app in the Simulator and attach to it in Hopper.

xcrun simctl launch booted net.robnapier.HackMe --wait-for-debugger

See what you can find. If you find things right away, then that's probably something that automated scripts are going to find, too. The fact that you don't find anything doesn't mean it's well hidden, but your lack of experience is somewhat balanced by the fact that you already know the answers. So it balances out a little bit.

What about obfuscation tools? Here's the thing. Trying to obfuscate Objective-C lives on a continuum between pointless and buggy. The more you obfuscate, the more you're going to inject bugs that rely on things having specific names. The more you don't at least change the names of things, the more pointless the effort is.

Obfuscating Swift that interacts with Cocoa have basically the same problems. You can't change the names of delegate methods. It's really tricky and bug-prone to change the names of things that rely on in key-value coding, like Interface Builder and KVO and Core Data and Core Animation.

Even if you can change the names of things, the truth is that names aren't that critical to reverse engineering.

```
func a(b: URL, c: @escaping (Data?) → Void) {
    URLSession.shared.dataTask(with: b) { (d, e, f) in
        if let g = f {
            print(g)
            c(nil)
        } else if let data = d {
            c(data)
        } else {
            c(nil)
        }
    }.resume()
}
```

So say I give you this code. I'm sure there's no way you could figure out what it's doing, right? This isn't quite fair. Even a basic obfuscater would write it more like this.

```
func o1(_ o7: o2, o3: @escaping (o4) → o5) {
    o20(o6(o7) {
        if let o8 = $2 {
            o9(o8)
            o3(nil)
        } else if let o10 = $0 {
            o3(o10)
        } else {
            o3(nil)
        }
    })()
}
```

And that is a little harder to understand. But frankly when you pull this into Hopper, it is less obfuscated than it is in the source code.

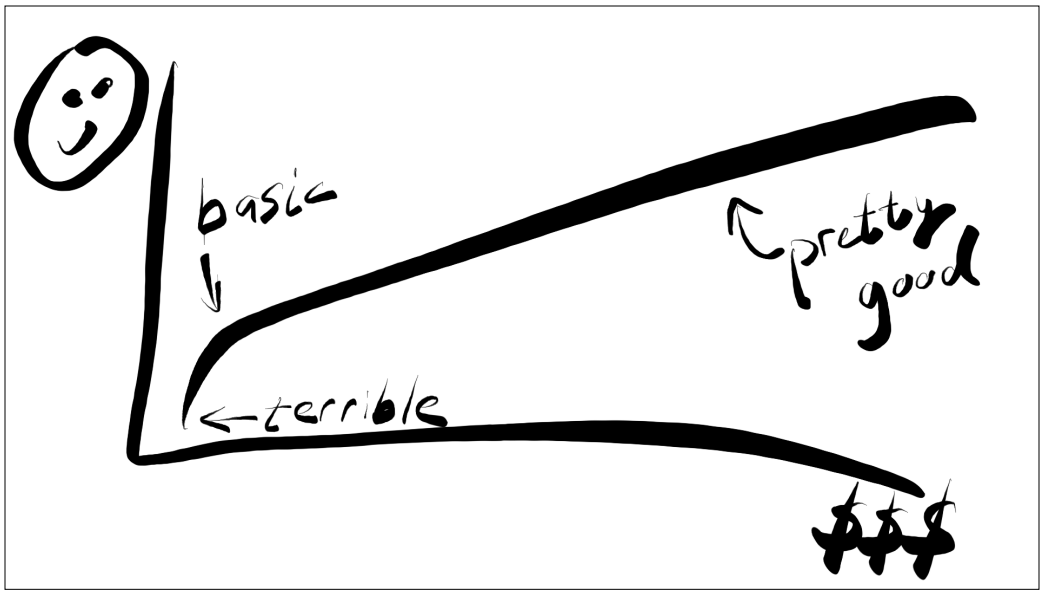
```
                            func o1(_ o7: o2, o3: @escaping (o4) → o5) {
                                o20(o6(o7) {
                                    if let o8 = $2 {
                                        o9(o8)
                                        o3(nil)
                                    } else if let o10 = $0 {
                                        o3(o10)
                                    } else {
                                        o3(nil)
                                    }
                                })()
                            }
r14 =
__T04main3o20yycSo18URLSessionDataTaskCcvp(__T04main2o6So18URLSessionDataTaskC10Foundation3URLV_yAE
0D0VSg_So11URLResponseCSgs5Error_pSgtctcvp(r12, function signature specialization <Arg[3] =
[Constant Propagated Function : closure #2 (Swift.Optional<Foundation.Data>) → () in main]> of
closure #1 (Foundation.Data?, __ObjC.URLResponse?, Swift.Error?) → () in main.o1(o7:
Foundation.URL, o3: (Foundation.Data?) → ()) → (), 0×0));


      o20 : (__ObjC.URLSessionDataTask) → () → ()
      o6 : (Foundation.URL, (Foundation.Data?, __ObjC.URLResponse?, Swift.Error?) → ()) →
      __ObjC.URLSessionDataTask
```

That o20 line looks like this.

And yes, that looks intimidating, but mostly because it's name mangled. If you run it through swift-demangle, you'll find that o20 is some void method on URLSessionDataTask, and o6 is a method that accepts an URL, and closure with data, response an error, and returns a URLSessionDataTask.

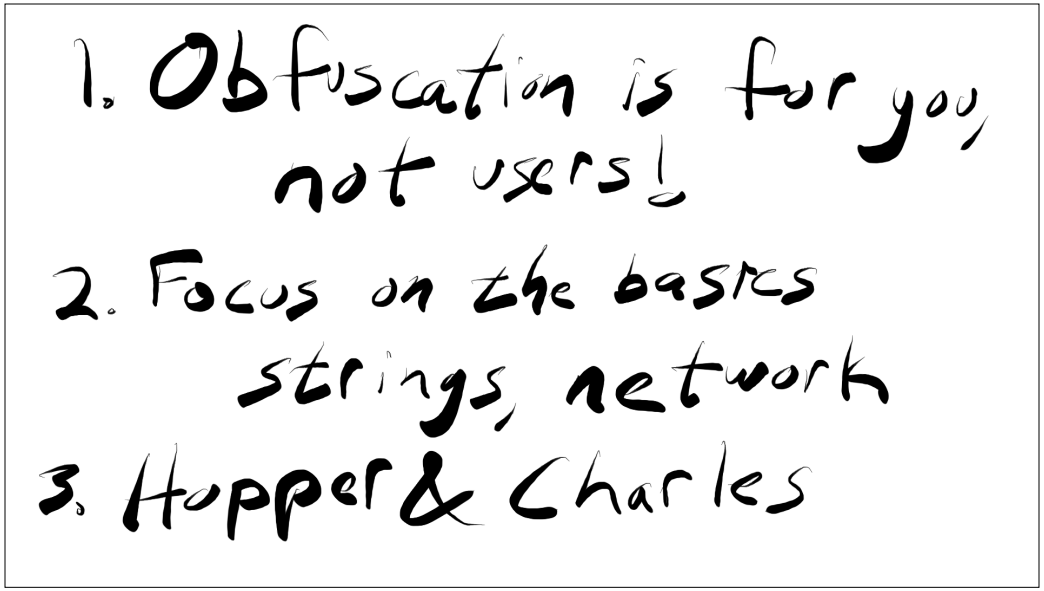
o20 : (__ObjC.URLSessionDataTask) -> () -> ()
o6 : (Foundation.URL, (Foundation.Data?, __ObjC.URLResponse?, Swift.Error?) -> ()) -> __ObjC.URLSessionDataTask

And there aren't a lot of methods that take an URL and a closure and return a data task. So we rename o6 as dataTask(), and that means o20 is almost certain resume, so we rename that one too. And pretty quickly, throughout the code, the obfuscation goes away. The parts the attacker cares about involve certain types, not certain names. They're looking for the network routines and the Keychain routines, and encryption functions. And the types involved are really distinctive. Things like URLSessionDataTask. And you can't hide the types because Swift needs them. This code actually tries to hide the types behind typealiases, but that doesn't matter in the final executable. I did a lot of work here for basically nothing.
So simple renaming tools really, in my opinion, don't do a lot for Swift. You get the same story in Java, .NET, PHP, JavaScript.



And that's really what it's all about. Costs and benefits. Security has a logarithmic payback. Just getting the basics right will take you a long way, but most don't get the basics right. Getting from the basics to well-designed security often requires expert help. Getting from best practice to best in class requires a large, ongoing investment. I'm hoping to help move teams from here [(poor)] to here [(basic)].

Today's talk has been about the outcast of security, obfuscation. There's a reason it's the outcast. Cryptography is orders of magnitude better. Obfuscation is fragile and requires ongoing maintenance. But some problems don't lend themselves to cryptography. Some problems don't have anywhere safe to store their secret, and so we're left with hiding and hoping.



I hope you go back, after the conference, and do the following.

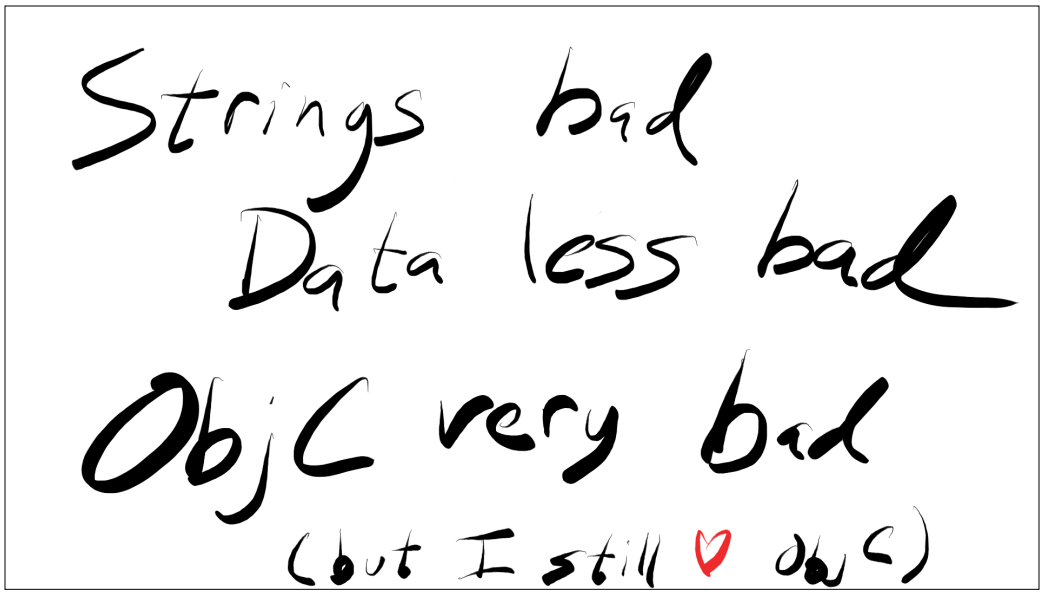
If you're using obfuscation to protect customer data, you've got to stop that. You have to use cryptography, and there has to be a secret outside the system.

But if you're protecting your own information from people who can download your app, then make sure you're not leaving your secrets lying around on the table.

Start with the absolute basics. Compile you app into an release-build IPA, and run `strings` on it. Look for any keys.

Spend $100 on Hopper and open your app up. Spend half a day trying to find things.

Spend $50 on Charles and see what your traffic looks like.



Never store sensitive data as strings. If the sensitive information in its purest form is an ASCII string, then split it with xor.
Do not store secrets in Objective-C. It's a reverse-engineer's dream. Pure Swift, C, C++ will generally be better once they're optimized. It's incredibly unlikely that they'll be worse.
Don't spend more time on this than it's really worth to you, and don't let DRM get in the way of paying customers.


Secrets
&
Lies
https://github.com/rnapier/secrets