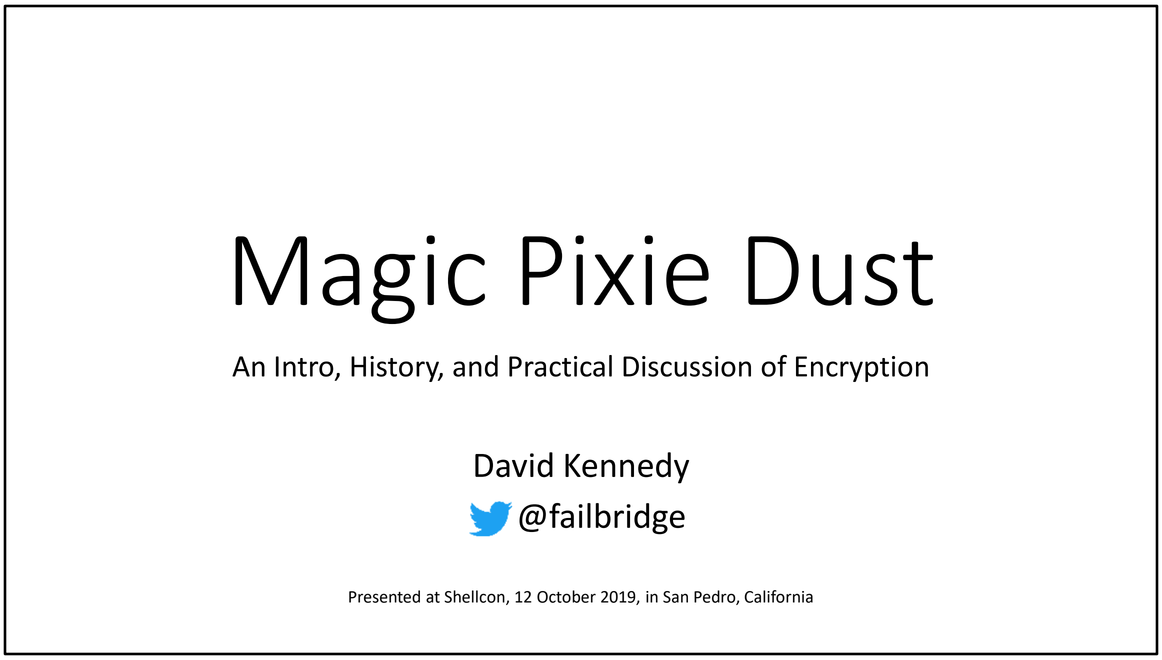


Magic Pixie Dust

An Intro, History, and Practical Discussion of Encryption

David Kennedy

@failbridge

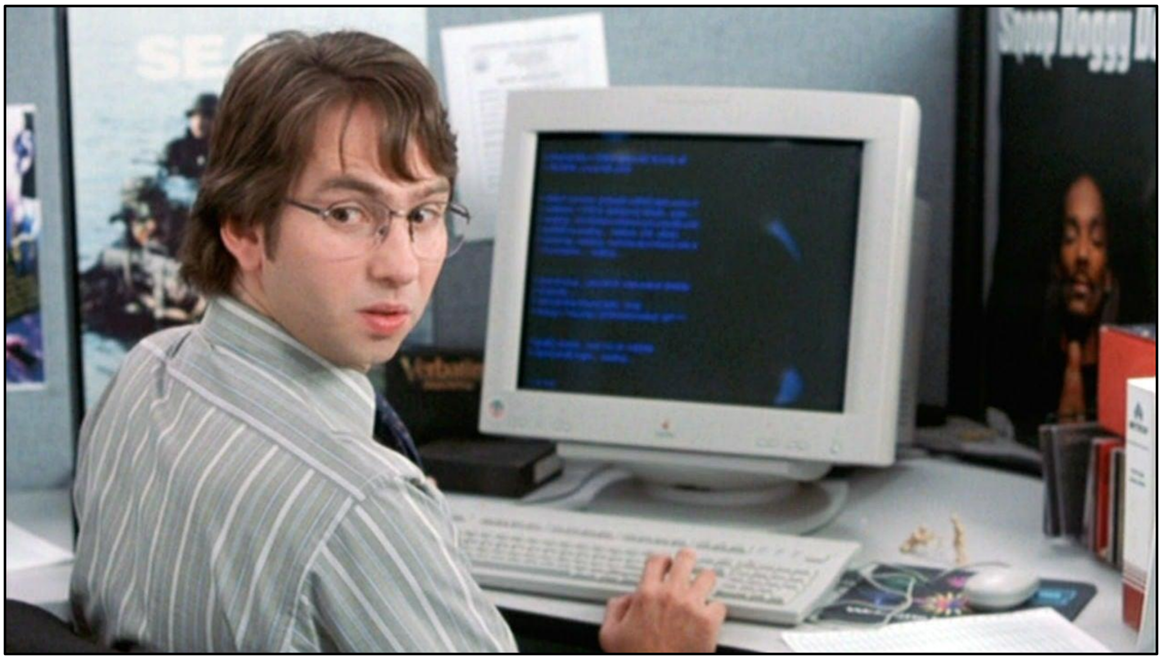
Presented at Shellcon, 12 October 2019, in San Pedro, California

Good morning. Everyone Enjoying Shellcon?

Welcome to: Magie Pixie Dust -- An Intro, History, and Practical Discussion of Encryption

My name is David Kennedy, you can find me on the twitters as failbridge. I mostly lurk, I don’t really tweet, but if you want to contact me, reach out to me there. I haven’t posted these slides anywhere yet, so if you want a copy – go to the twitters and we’ll figure it out.

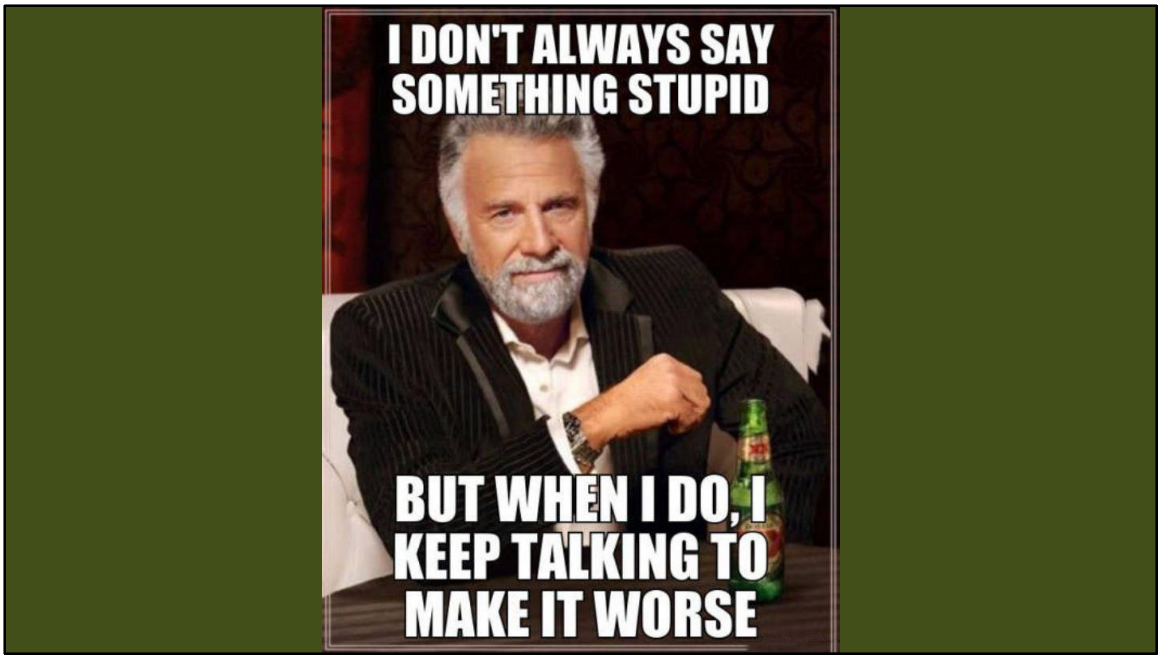
I am not a cool InfoSec person like most of you here today….



I’m just a Software Engineer. Specifically, a .NET software engineer – I mostly write in C# -- judge me if you must. So - this talk isn’t about how to attack encryption – it’s more from the perspective of me, as a developer – it’s what I’ve learned while trying not to f*** it up, but I hope that’s still useful.

Cryptography. It’s not magic pixie dust. You can’t just sprinkle encryption on your data and expect that it’s okay. But we will be going to never never land because we’ll go over some things that you should never never do.

I promise that’s my last dad joke, but we will talk about some things you should avoid.


I DON'T ALWAYS SAY SOMETHING STUPID
BUT WHEN I DO, I KEEP TALKING TO MAKE IT WORSE

The origins of this talk go back to a conversation that I had with a co-worker. I said something dumb. Well, what I said wasn't dumb – it's that I said something like it was fact without first having done my research. Here's how it went.

We were discussing the conspiracy theory that the moon landing was faked. I offered, what I thought, was the most compelling argument for why it was, definitely….[pause for dramatic effect] …. Not faked.

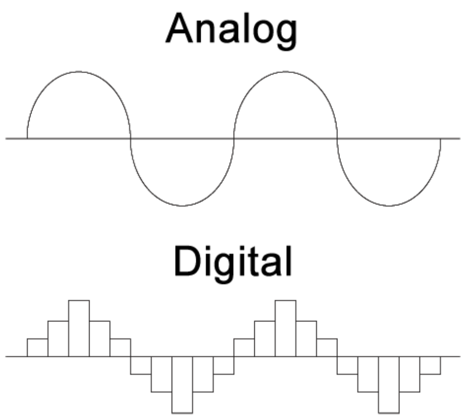
I said that all you'd really need to do (while the landing was taking place) to verify that "yes, they're on the moon", would be to point a radio antenna at the Moon, hear the signal, and triangulate it. Surely someone with a vested interest in confirming that we really were on the moon (Russia) would have done this?

SOVIET
BEAR

And here’s what I said without first doing my research: I said “and you couldn’t encrypt voice radio transmissions back then either”.

I said this because, at the time, when I thought of encryption I thought of the “fancy math” encryption – the type that we use in today’s encryption. Modern Day Encryption, like AES, depends upon having a discrete signal and radio transmissions in 1969 were analog – there just wasn’t the computational power needed to convert analog signals to binary on the fly. If your encryption algorithm requires discrete input, then you can’t use that algorithm on an analog signal.

My co-worker said “That’s not true, they COULD encrypt analog signals in 1969”.



It’s true that the Apollo radio transmissions were analog.

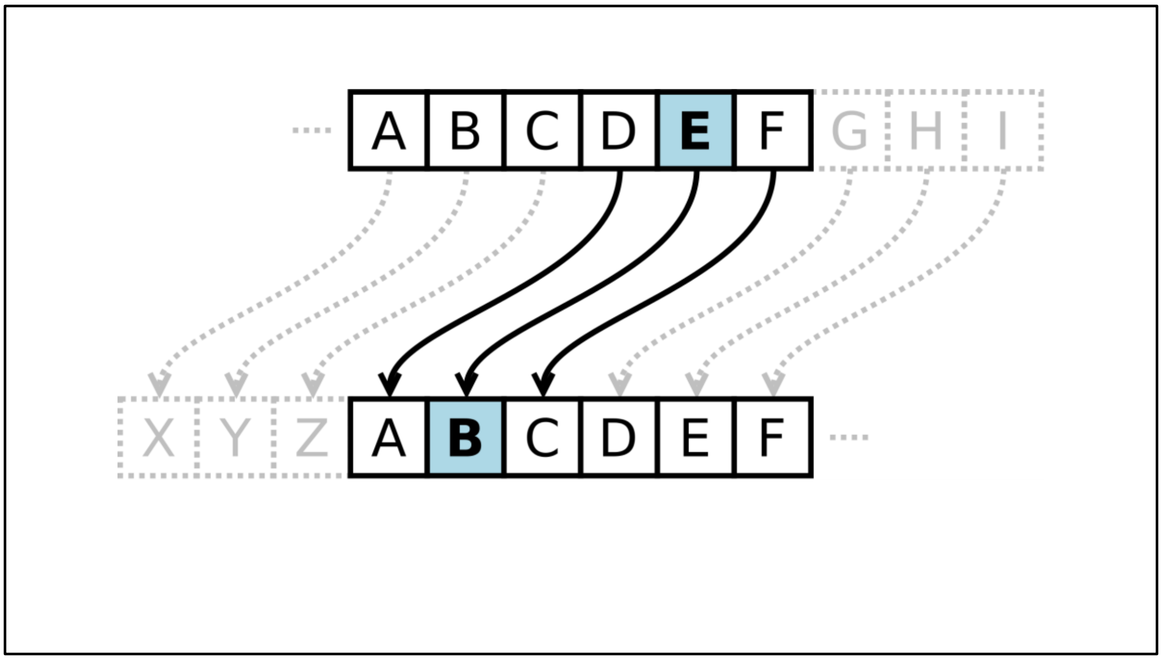
It’s also true that when we, in a general sense, talk about cryptography, ESPECIALLY in the context of computer security, we’re talking about algorithms that operate on discrete input – not analog.

But computers are a product of the modern age. So what if we’re talking about cryptography that is not of the modern age? Let’s go back a couple of thousand years. Discuss some basics, and circle back to the Moon landing.

Show of hands: Who knows what ROT13 is?

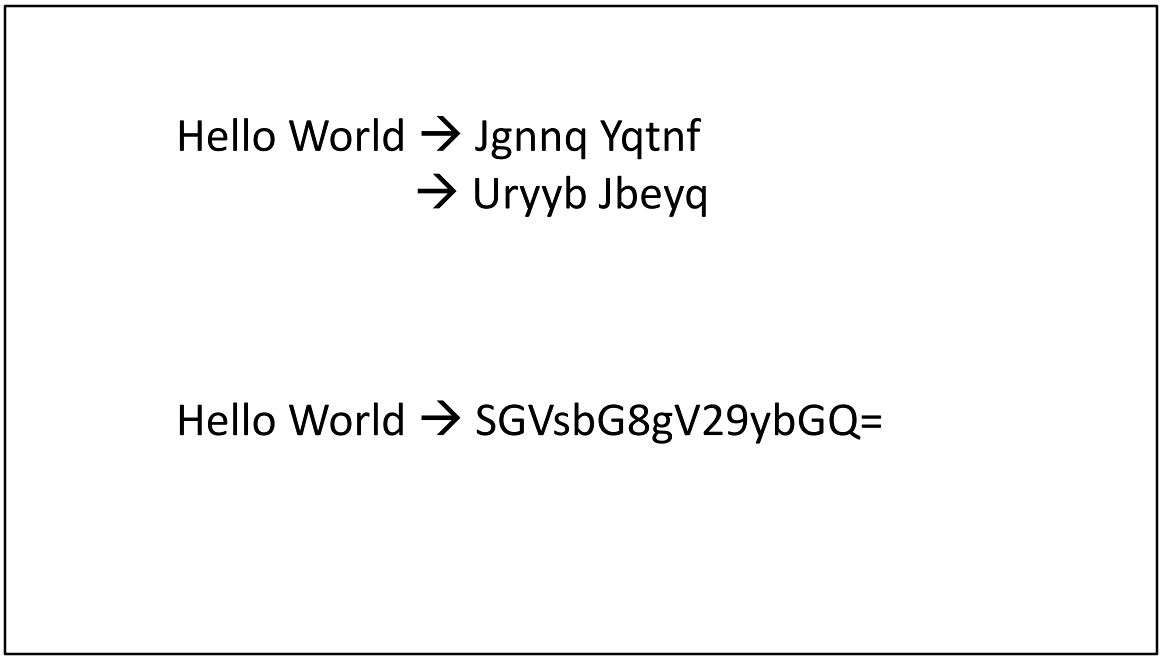
It’s also known as the “Caesar Cipher” – that’s a picture of a Caesar Salad…. But it’s called the Caesar Cipher because….

….because Julius Caesar used it in his private correspondence.



The Caesar Cipher is what's known as a substitution cipher because you substitute one value for another. Substitution Ciphers are a whole class of encryption, but the this one is sometimes called a "shift" cipher because the way you substitute is by shifting the alphabet back and forth. In encryption, we also talk about Keys. In this case, the key would 3 because you shift all the input letters to the left by 3 spaces. To decrypt, you'd shift right by the same amount.

Rot 13 (which is short for "Rotate 13") is shifted 13 letters. Because English has 26 letters, and 13 is half of that, by "encrypting" a second time you get the decrypted value.


Hello World → Jgnnq Yqtnf

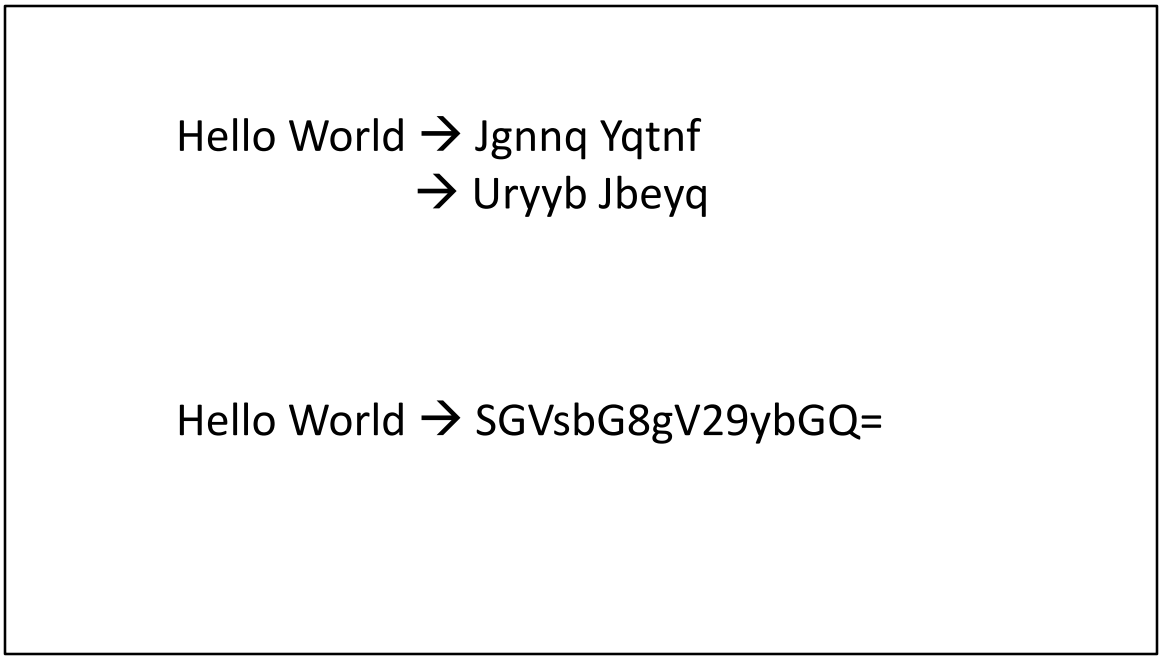
→ Uryyb Jbeyq

Hello World → SGVsbG8gV29ybGQ=

So here is “Hello World” encrypted with the Caesar Cipher with a key of 1, and then again with a key of 13.

[Click to show Base64 at bottom]

Does anyone know what this is?



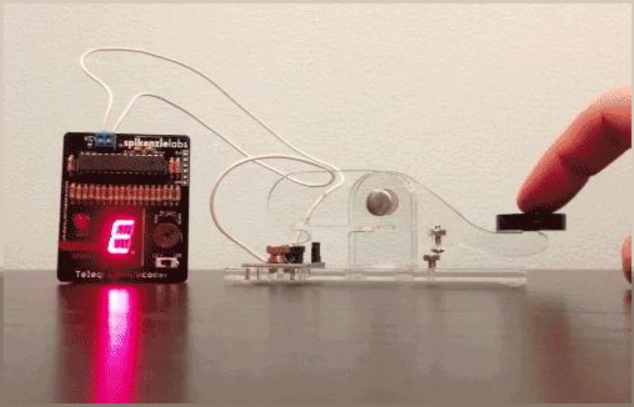
Hello World → Jgnnq Yqtnf
→ Uryyb Jbeyq

Hello World → SGVsbG8gV29ybGQ=

Conceivably, you could create base 63 – just use every character in base 64 except 1 – maybe you don't use uppercase G…

If you have just pen & paper, base 64 (or whatever base) is considerably more difficult than the Caesar Cipher. So why do we call one encryption and the other encoding?

The answer is intent. It depends on what you intend the transformation algorithm to do. Caesar intended to keep his correspondence secret. Base64 was intended to be a common, well-known, way to map binary data to ASCII.



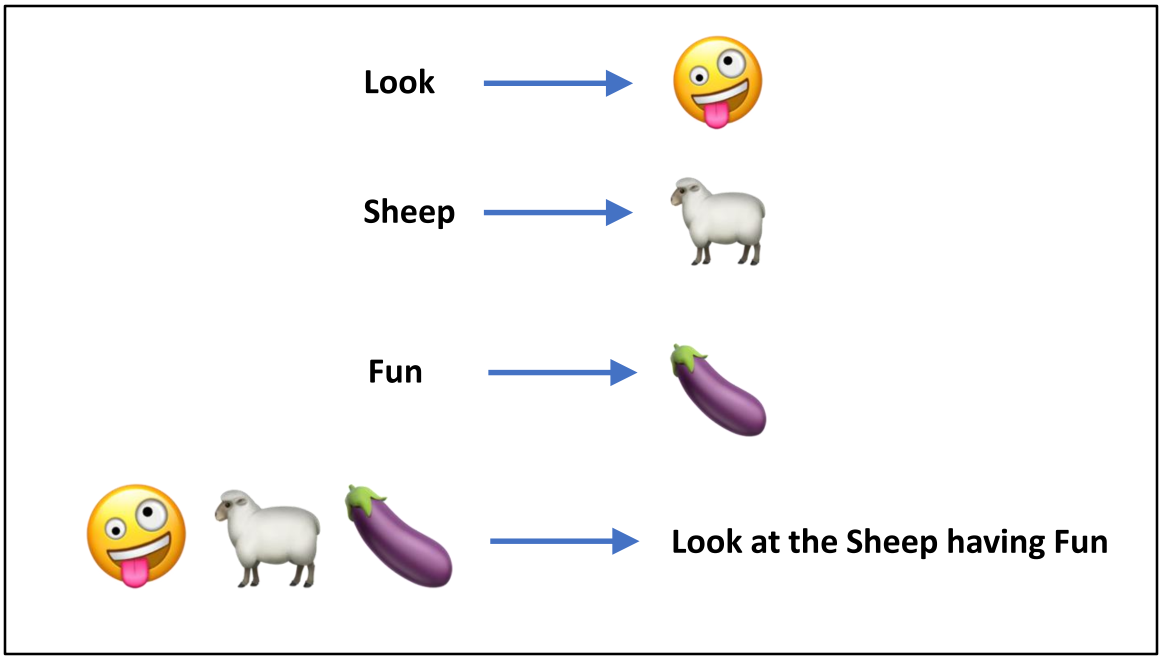
A	● ▬
B	▬ ● ● ●
C	▬ ● ▬ ●
D	▬ ● ●
E	●
F	● ● ▬ ●
G	▬ ▬ ●
H	● ● ● ●
I	● ●
J	● ▬ ▬ ▬
K	▬ ● ▬
L	● ▬ ● ●

If Morse Code had been created with the intent of taking plain-text messages and converting them into something “strange” to keep them secret… then we would call that encryption. But we don’t, we simply call it encoding.

Gif Source: https://giphy.com/gifs/shed-calculator-labs-4AUH1t6ccRfhe

Encryption is the process of encoding a message or information in such a way that only authorized parties can access it and those who are not authorized cannot

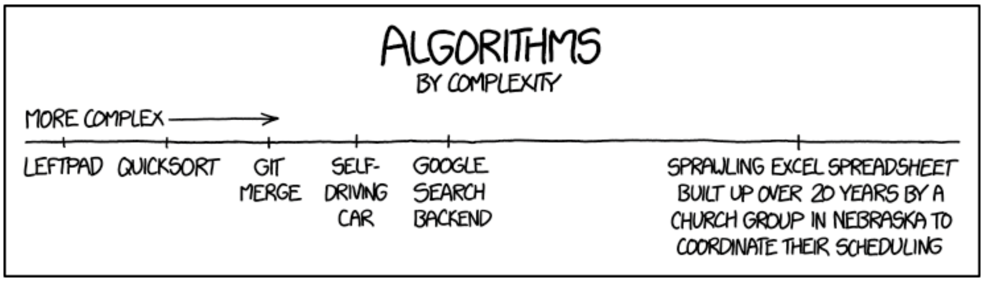
Here is the definition of Encryption, straight from Wikipedia.



And Encoding is just mapping one set of symbols to another set – secret or public.

For example, Let’s do this with Emojis. Suppose we map the word Look to this, Sheep to Sheep, and Eggplant means Fun.

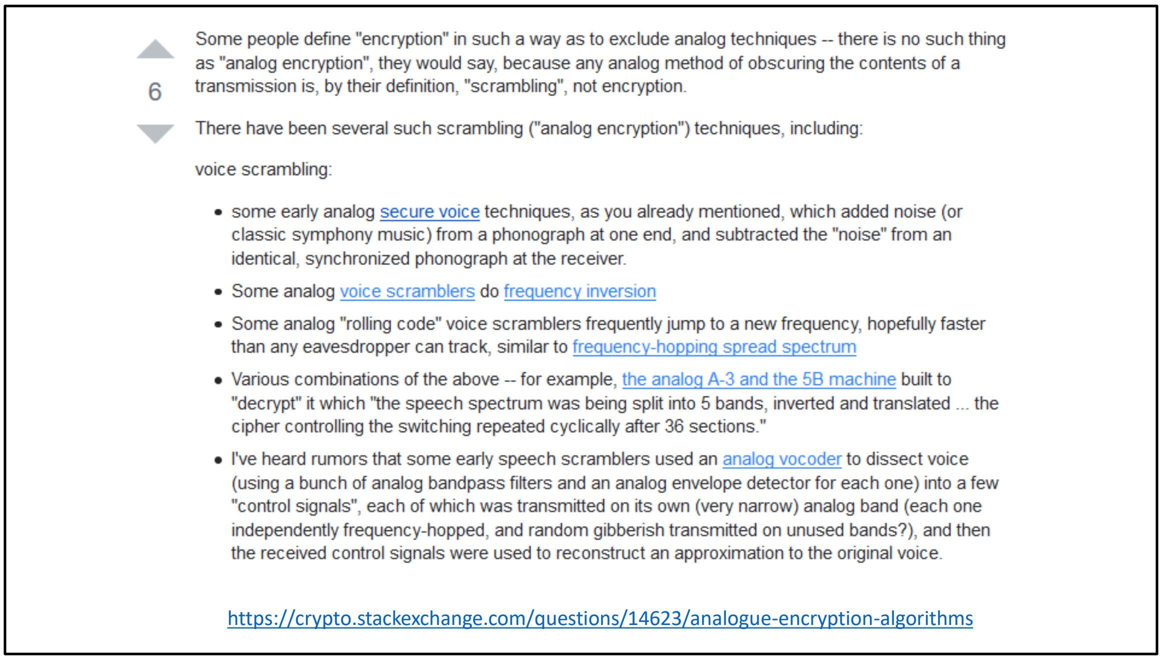
Then we can start building sentences with our encoding scheme……So zany-faze sheep eggplant means “look at the sheep having fun”. I can’t imagine what else that might mean….


ALGORITHMS
BY COMPLEXITY
MORE COMPLEX
LEFTPAD
QUICKSORT
GIT MERGE
SELF-DRIVING CAR
GOOGLE SEARCH BACKEND
SPRAWLING EXCEL SPREADSHEET BUILT UP OVER 20 YEARS BY A CHURCH GROUP IN NEBRASKA TO COORDINATE THEIR SCHEDULING

Source: https://www.xkcd.com/1667/

The Cipher is the algorithm that's used to perform the mapping. Some are more complex than others.

So….. So far, we've covered some pretty basic stuff, and I want to get to some more fun stuff (show eggplant) – talk about the more complex algorithms, the types of ones we use today, like AES, but first I want to circle back to the story about the moon landing and the idea of encrypting analog signals.



6

Some people define "encryption" in such a way as to exclude analog techniques -- there is no such thing as "analog encryption", they would say, because any analog method of obscuring the contents of a transmission is, by their definition, "scrambling", not encryption.

There have been several such scrambling ("analog encryption") techniques, including:

voice scrambling:

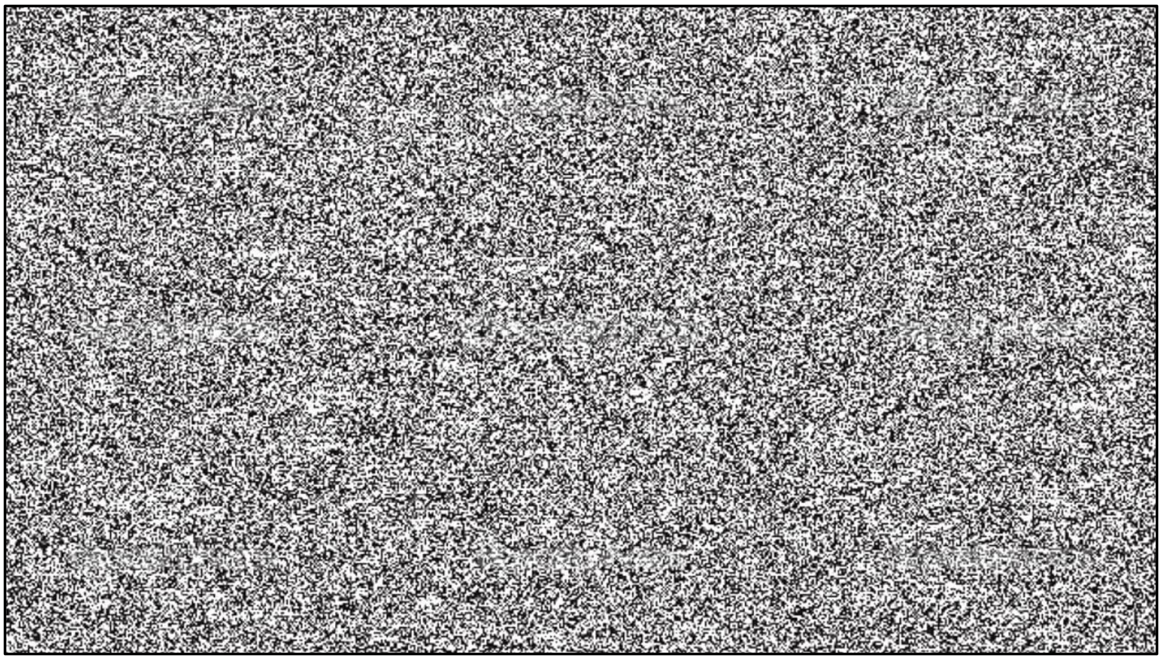
- some early analog secure voice techniques, as you already mentioned, which added noise (or classic symphony music) from a phonograph at one end, and subtracted the "noise" from an identical, synchronized phonograph at the receiver.
- Some analog voice scramblers do frequency inversion
- Some analog "rolling code" voice scramblers frequently jump to a new frequency, hopefully faster than any eavesdropper can track, similar to frequency-hopping spread spectrum
- Various combinations of the above -- for example, the analog A-3 and the 5B machine built to "decrypt" it which "the speech spectrum was being split into 5 bands, inverted and translated ... the cipher controlling the switching repeated cyclically after 36 sections."
- I've heard rumors that some early speech scramblers used an analog vocoder to dissect voice (using a bunch of analog bandpass filters and an analog envelope detector for each one) into a few "control signals", each of which was transmitted on its own (very narrow) analog band (each one independently frequency-hopped, and random gibberish transmitted on unused bands?), and then the received control signals were used to reconstruct an approximation to the original voice.

https://crypto.stackexchange.com/questions/14623/analogue-encryption-algorithms

I promise, this is the “only wall of text” in this whole presentation. In doing my research to figure out if and how analog signals, especially voice, could be encrypted I came across this answer on Stack Exchange Cryptography. I pasted the screenshot up here because this user put it better than I ever could, and I didn’t want to just paste it into the script I have in the notes. (aka, I got lazy on this slide).

If you want to dig into this further for yourself (there are several good links in there), here’s the link to the Stack Overflow page. Reach out to me on twitter if you want the slides.

[Read first paragraph]

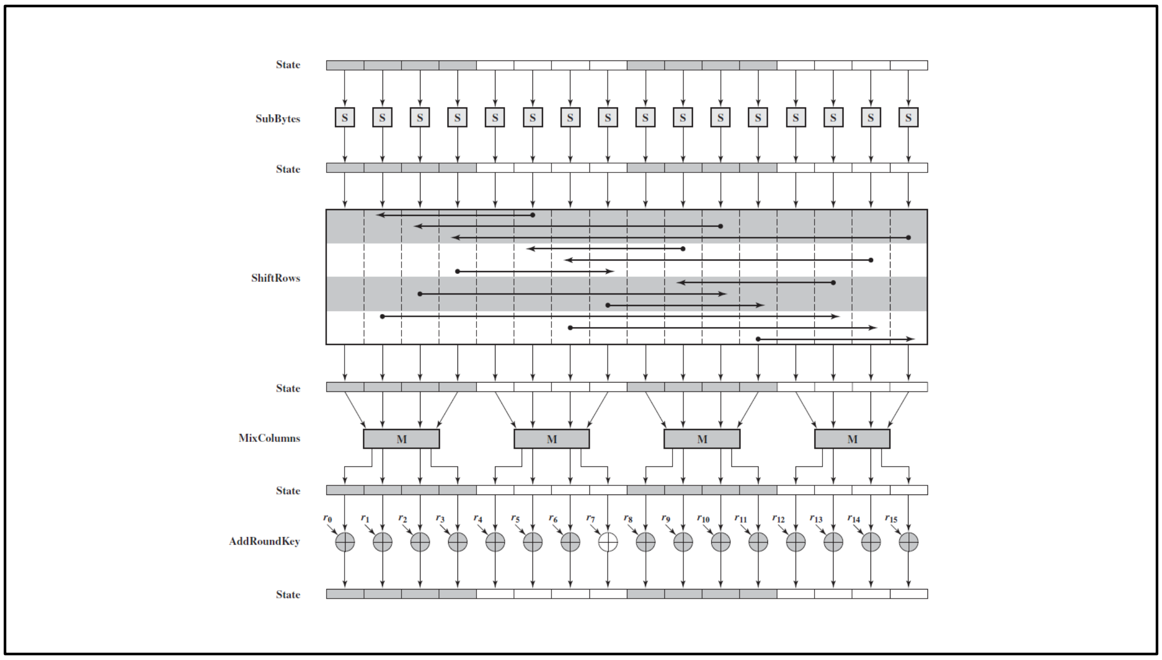


One method to secure analog signals is adding pre-recorded white noise to the transmission. On the receiving end, audio filters will filter out the white noise by using the same pre-recording. In this case, the cipher is the fact that you add white noise and the key is the pre-shared white-noise recording.

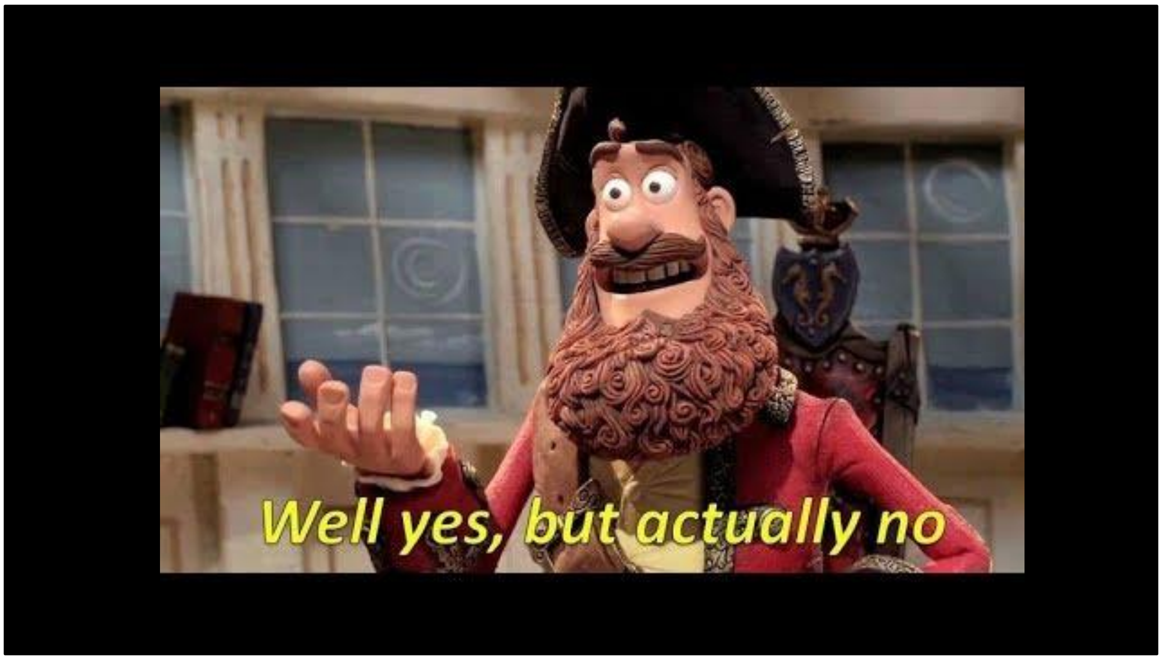
Another technique would be frequency shifting, or you could call it frequency modulation. If both the sender and receiver know what frequency shifts occur at what time-offsets, then this can be effective. That knowledge of the what frequency shifts occur and at what time periods is the encryption key.

I mentioned frequency modulation – that’s also called FM, which is the same thing as everyday FM radio. Again, the difference between encryption and encoding really is intent. So, you can say that obfuscation is encryption…. But you should never never say something like “oh, the data, or the code, is obfuscated, so we’re fine….”. But enough about the analog encryption techniques.

Let’s get back to the fun stuff – let’s start talking about stuff that is more relevant to our everyday lives. Modern Encryption.


State
SubBytes
State
ShiftRows
State
MixColumns
State
AddRoundKey
State
S
M
r_0 r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12} r_{13} r_{14} r_{15}

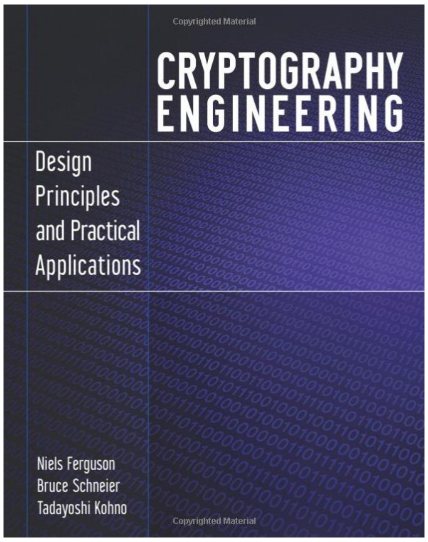
We know that the Caesar Cipher is easy to break. If there's only 26 letters in the alphabet, you can brute-force it in 26 tries or fewer. It's easy to break because it's so simple. So we might be tempted to say that the more complicated the algorithm, the better.

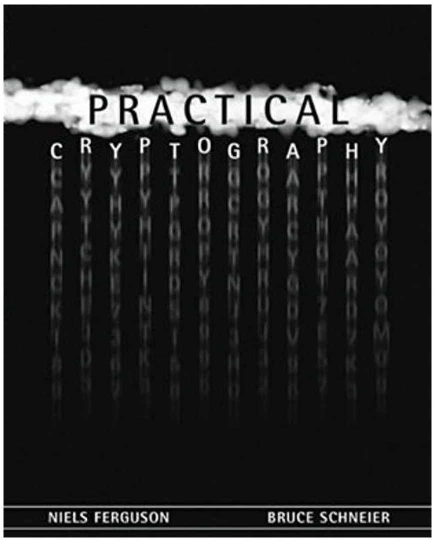


The more complex an algorithm, the slower it is. Even if you are using the correct key.

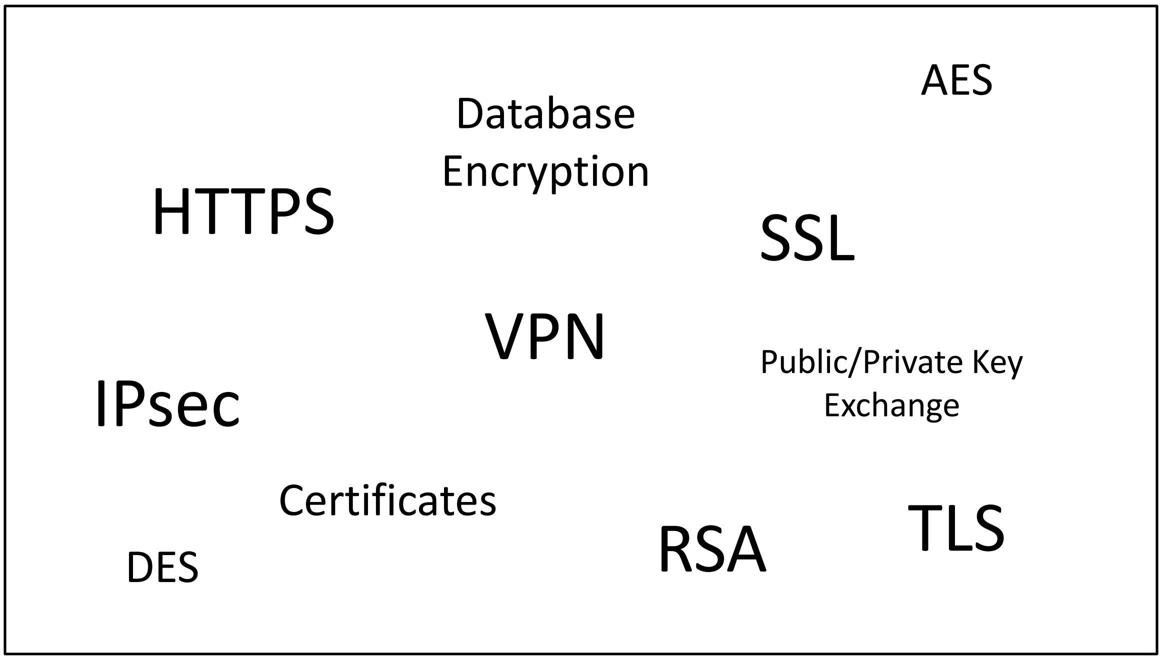
A super complex algorithm may be fine if you have a super computer. But if you're dealing with very low-power scenarios, then the same algorithm may not be the best choice. There's a balance to be had between complexity (that is, difficulty), and speed. A good algorithm is easy to encrypt and decrypt with the right key, but difficult if you don't.

If you'd like a really good, more formal, and academic discussion of what is "good" encryption,





Then I'd recommend Cryptography Engineering by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. I think the whole thing is good and talks about a lot more than just formal definitions of "good encryption", really. And while you're looking at books, I think that you can safely ignore Practical Cryptography. It's good too, but Crypto Engineering is by 2 of the same authors and is basically "Practical Crypto" second edition.



AES

Database Encryption

HTTPS

SSL

VPN

Public/Private Key Exchange

IPsec

Certificates

TLS

RSA

DES

So, let’s talk about Encryption in a more real-world sense. You may have heard of some of these terms and know that that there’s encryption involved, but let’s dig in a little bit.

Symmetric

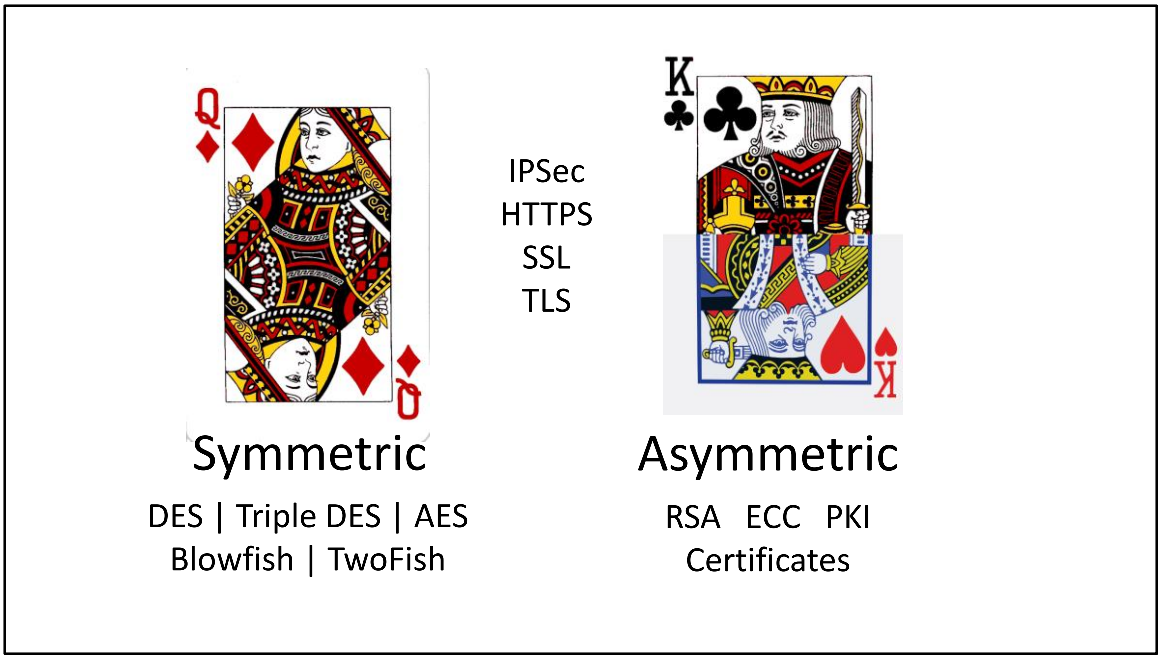
DES | Triple DES | AES
Blowfish | TwoFish

Asymmetric

I'm going to talk about two types of encryption: Symmetric and Asymmetric.

With Symmetric encryption, the same key is used for both encryption and decryption. So far, everything we've discussed – all the way from Caesar Cipher to the white-noise in analog audio – has been symmetric.

In the Symmetric world, you'll see things like DES, Triple DES, AES, Blowfish, Twofish, among others.



IPSec
HTTPS
SSL
TLS

Symmetric

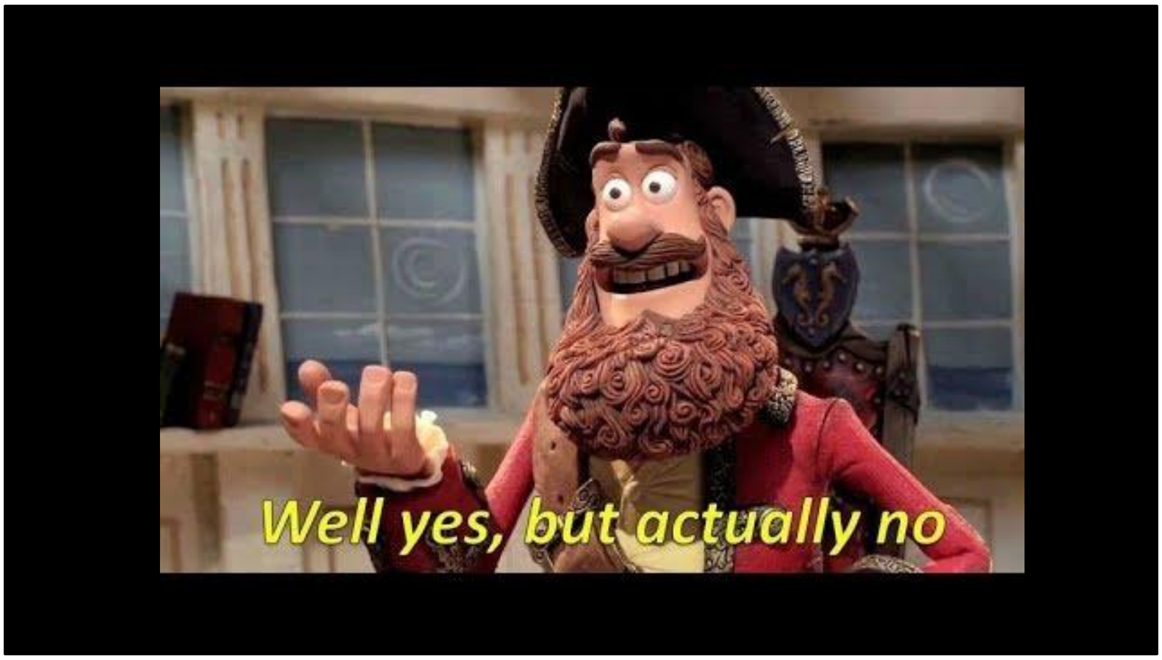
DES | Triple DES | AES
Blowfish | TwoFish

Asymmetric

RSA ECC PKI
Certificates

In the Asymmetric world you'll see things like RSA, ECC, which is Elliptic Curve Cryptography, PKI, which is Public Key Infrastructure – that's going to be your public/private key related stuff, and Certificates. Now, with the things I listed out over here on the right, there is some overlap, and I'll get to that.

And here in the middle we have things like IPSec, HTTPS, SSL, & TLS. A lot of people will say "oh, well, SSL, that's all about having an SSL Certificate, so it must be Asymmetric..



Well, yes, but actually no…. And I'll come back to that.

Symmetric

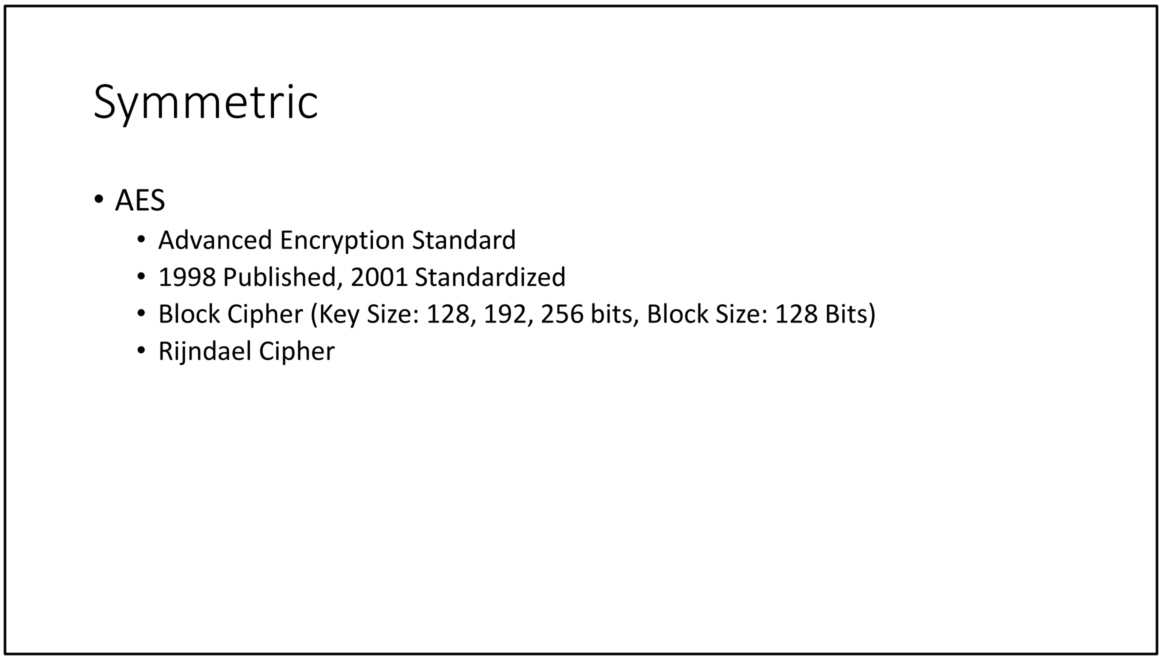
- DES
  - Data Encryption Standard
  - Mid 70s – Late 90s
  - Block Cipher (Key Size: 56 bits, Block Size: 64 Bits)

Let’s start with DES, which stands for Data Encryption Standard. In the mid 70s it was selected as a FIPS algorithm. It became the de facto standard until the late 90s when a new standards were approved. It’s a Block Cipher with a key length of only 56 bits. A few things made DES weak, but the 56 bit key length drew a lot of criticism. It was effective in the mid 70s, but became vulnerable as Moore’s law meant that computers become faster and faster.

Symmetric

- Triple DES (3DES)
  - Data Encryption Standard (times THREE!)
  - 1995
  - Block Cipher (Key Size: 168 bits, Block Size: 64 Bits)

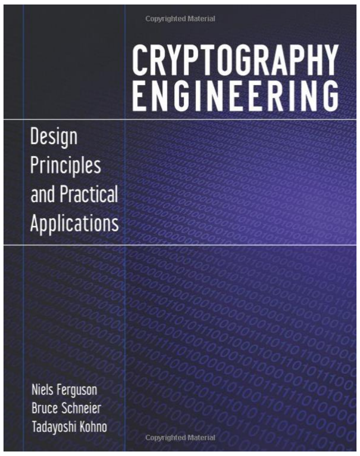
In 1995, Triple DES came along, which was basically DES times THREE! I have a picture of Triple Sec up there because I *desperately* want to invent an encryption algorithm just so I can name it Triple Sec. You'll notice the Key Size is 168 – that's 56 times 3. Well, technically, 3DES could use keys of either 56, 112, or 168 bits. I won't go into the details, but it *basically* encrypted each block of 64 bits 3 times, each time with a different key.



Symmetric

- AES
  - Advanced Encryption Standard
  - 1998 Published, 2001 Standardized
  - Block Cipher (Key Size: 128, 192, 256 bits, Block Size: 128 Bits)
  - Rijndael Cipher

AES hit the scene in 1998 and became a Standard in 2001. It’s significantly stronger than DES or 3DES. AES is not actually the name of the algorithm – it’s actually… well, I can’t pronounce that.

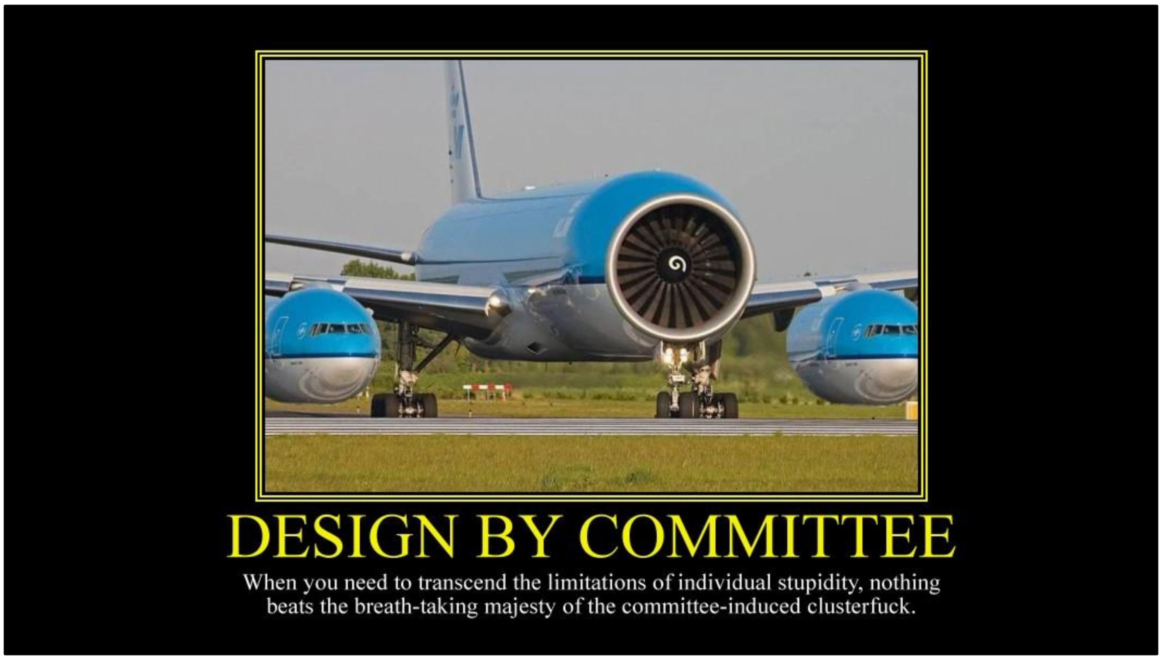


There has been some confusion about the correct pronunciation of "Rijndael." Don't worry; it's hard to pronounce unless you speak Dutch, so just relax and pronounce it any way you like, or just call it "AES".

~ Cryptography Engineers, footnote on page 54

To reference this book again…. Do we have any Dutch speakers in the room?

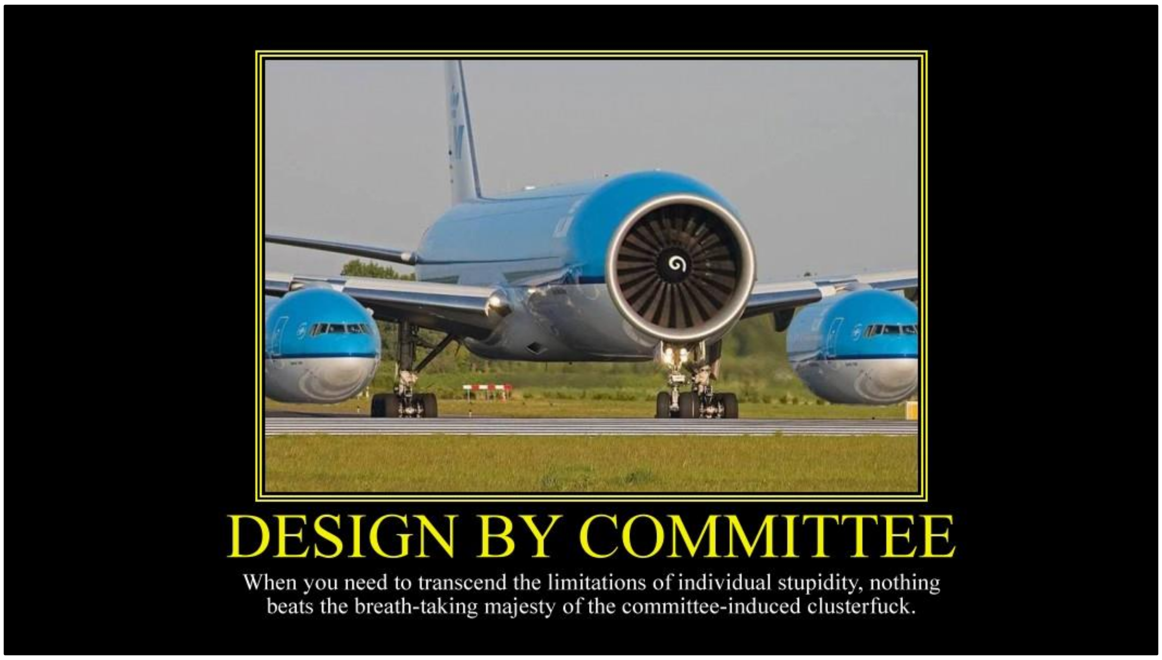
AES is still a recommended standard and provides pretty good encryption. It's not *the* strongest, but no-one will fault you for using AES. Let's say you need to implement encryption and you select an algorithm that's generally considered stronger than AES and… down the road, your application or system gets popped. In the subsequent investigation, you'll most likely really have to defend your choice to use whatever algorithm. Shoot, let's suppose that you work in a regulated industry, like banking or healthcare, and all of your stuff needs to pass a security audit before being released. No one will question you use of AES. Anything else, even if strong security, will most likely draw strong scrutiny.


DESIGN BY COMMITTEE
When you need to transcend the limitations of individual stupidity, nothing beats the breath-taking majesty of the committee-induced clusterfuck.

So AES is still considered “Pretty Good” to this day. DES kind of sucked from the beginning. That’s largely due to the way in which it became a standard. When a replacement for DES was needed, a committee did not decide to design the replacement – instead, a request for proposals was published. AES or, really… that Dutch name that I can’t pronounce, was one of many submissions. TwoFish by Bruce Schneier was another submission. Regardless of the algorithm name, whatever was to be selected was destined to be called “AES”.

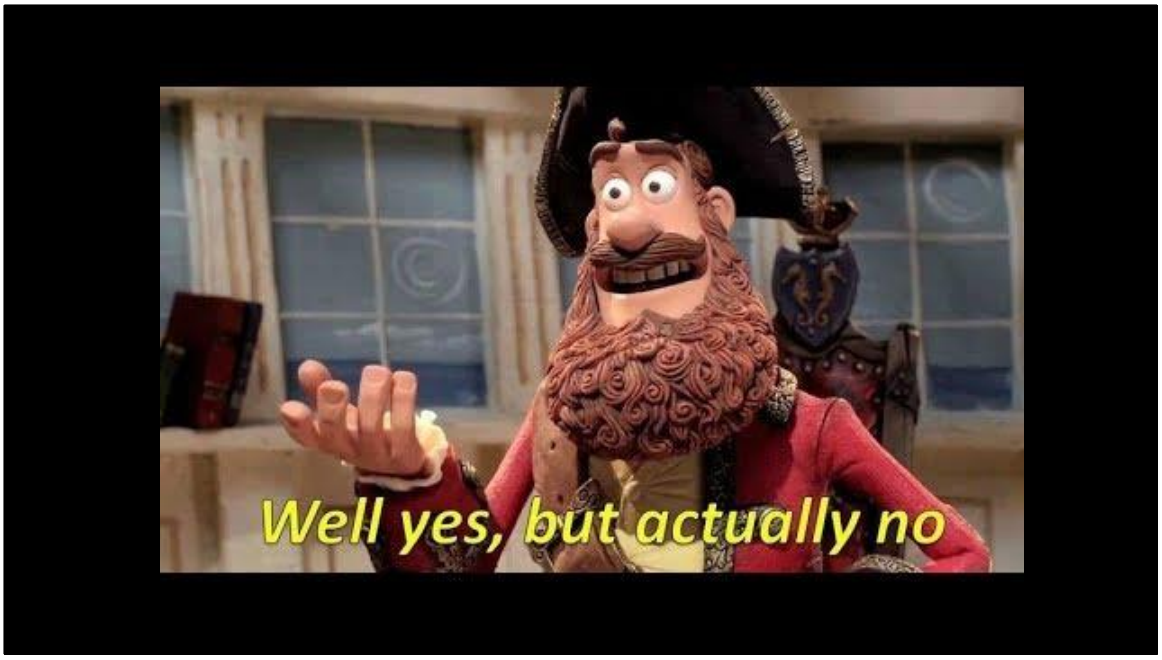
AES was selected because it was regarded as the best balance between the things that NIST thought were important – strength, speed, performance, etc…

A perfect example of design-by-committee that sucked is WEP, the wi-fi encryption that *you. Should. Never. Use*


DESIGN BY COMMITTEE
When you need to transcend the limitations of individual stupidity, nothing beats the breath-taking majesty of the committee-induced clusterfuck.

Because it was part of an open competition and not part of a closed design-by-committee, AES – as well as the other submissions – received a ton of scrutiny. As a result, the cryptography and security communities have a pretty good understanding of it any potential weaknesses. I say this because you should never trust encryption ciphers or algorithms that have not undergone public scrutiny.

So it would make sense that it’s a bad idea to “roll your own” cryptography. If you go with a well vetted standard, you should be good, right?

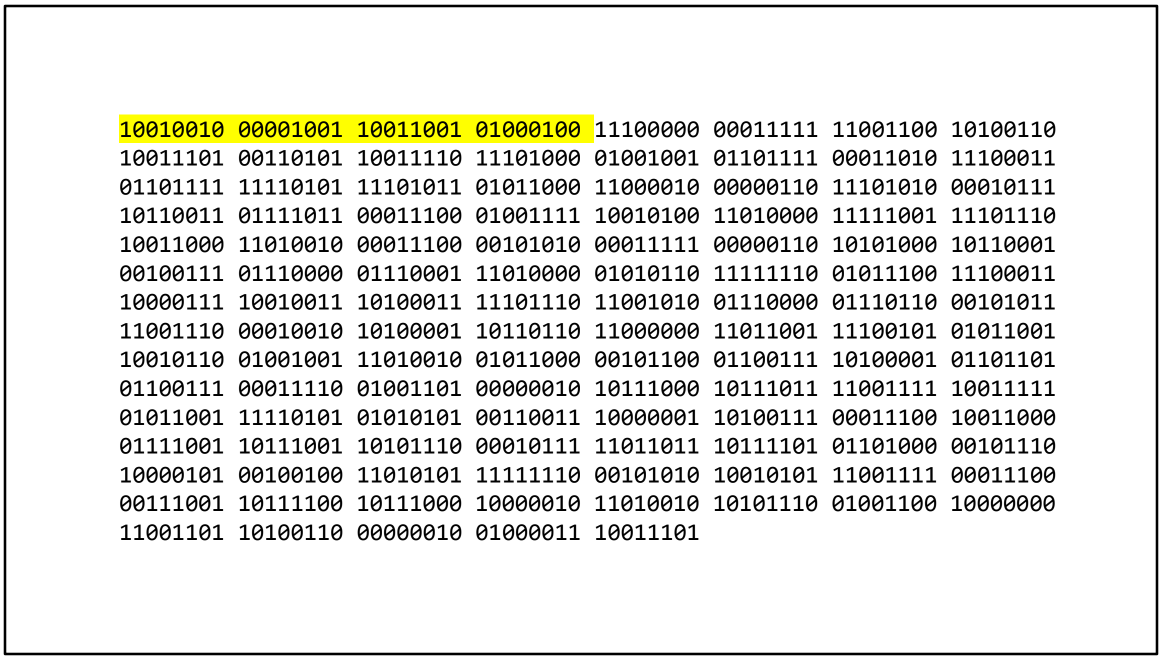


When I first needed to implement encryption in a project, I thought that “roll your own” meant inventing your own encryption cipher. Don’t invent your own cipher when there are some pretty solid ones already out there. It turns out that doing encryption the right way is much harder than that.

AES, and DES for that matter, are both what are called block ciphers. Let’s dig into that a little bit more to understand what that means, and how it fits into this concept of “don’t roll your own crypto”

```
10010010 00001001 10011001 01000100 11100000 00011111 11001100 10100110
10011101 00110101 10011110 11101000 01001001 01101111 00011010 11100011
01101111 11110101 11101011 01011000 11000010 00000110 11101010 00010111
10110011 01111011 00011100 01001111 10010100 11010000 11111001 11101110
10011000 11010010 00011100 00101010 00011111 00000110 10101000 10110001
00100111 01110000 01110001 11010000 01010110 11111110 01011100 11100011
10000111 10010011 10100011 11101110 11001010 01110000 01110110 00101011
11001110 00010010 10100001 10110110 11000000 11011001 11100101 01011001
10010110 01001001 11010010 01011000 00101100 01100111 10100001 01101101
01100111 00011110 01001101 00000010 10111000 10111011 11001111 10011111
01011001 11110101 01010101 00110011 10000001 10100111 00011100 10011000
01111001 10111001 10101110 00010111 11011011 10111101 01101000 00101110
10000101 00100100 11010101 11111110 00101010 10010101 11001111 00011100
00111001 10111100 10111000 10000010 11010010 10101110 01001100 10000000
11001101 10100110 00000010 01000011 10011101
```

If this is the binary data that you'd like encrypted, and you have a block size of, say 24 then



```
10010010 00001001 10011001 01000100 11100000 00011111 11001100 10100110
10011101 00110101 10011110 11101000 01001001 01101111 00011010 11100011
01101111 11110101 11101011 01011000 11000010 00000110 11101010 00010111
10110011 01111011 00011100 01001111 10010100 11010000 11111001 11101110
10011000 11010010 00011100 00101010 00011111 00000110 10101000 10110001
00100111 01110000 01110001 11010000 01010110 11111110 01011100 11100011
10000111 10010011 10100011 11101110 11001010 01110000 01110110 00101011
11001110 00010010 10100001 10110110 11000000 11011001 11100101 01011001
10010110 01001001 11010010 01011000 00101100 01100111 10100001 01101101
01100111 00011110 01001101 00000010 10111000 10111011 11001111 10011111
01011001 11110101 01010101 00110011 10000001 10100111 00011100 10011000
01111001 10111001 10101110 00010111 11011011 10111101 01101000 00101110
10000101 00100100 11010101 11111110 00101010 10010101 11001111 00011100
00111001 10111100 10111000 10000010 11010010 10101110 01001100 10000000
11001101 10100110 00000010 01000011 10011101
```

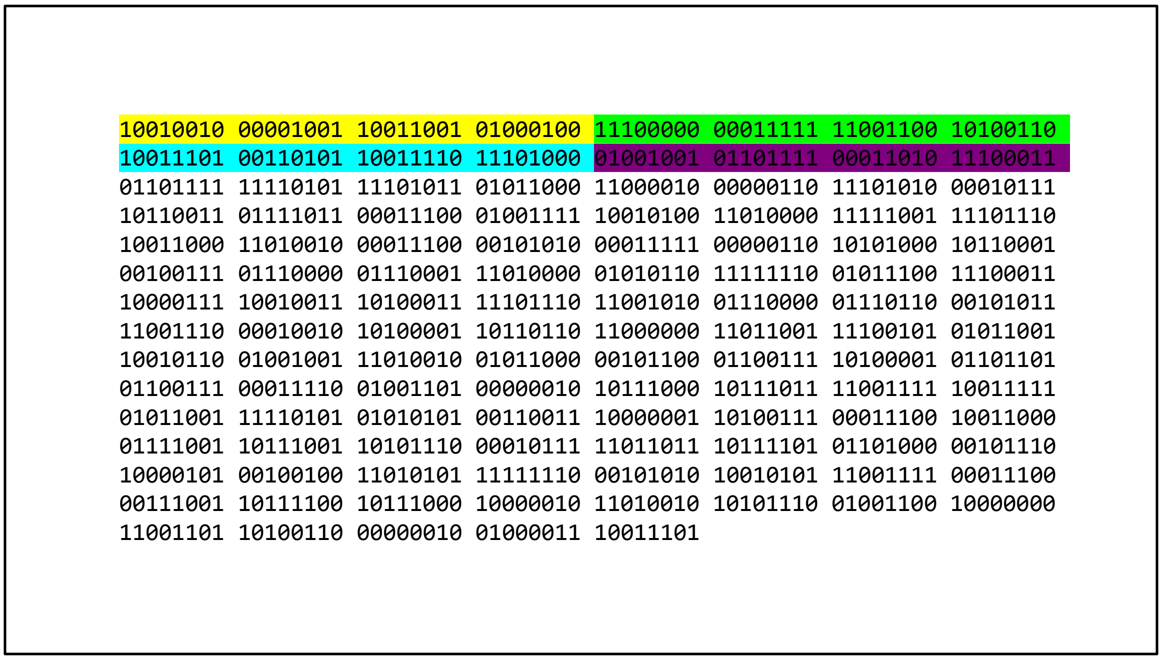
Then this is the first block, and it's encrypted with the bits from the key. In the case of DES, you have a 56 bit key that would get blended with this. In the case of AES, the key size could be 128, 192, or 256.

```
10010010 00001001 10011001 01000100 11100000 00011111 11001100 10100110
10011101 00110101 10011110 11101000 01001001 01101111 00011010 11100011
01101111 11110101 11101011 01011000 11000010 00000110 11101010 00010111
10110011 01111011 00011100 01001111 10010100 11010000 11111001 11101110
10011000 11010010 00011100 00101010 00011111 00000110 10101000 10110001
00100111 01110000 01110001 11010000 01010110 11111110 01011100 11100011
10000111 10010011 10100011 11101110 11001010 01110000 01110110 00101011
11001110 00010010 10100001 10110110 11000000 11011001 11100101 01011001
10010110 01001001 11010010 01011000 00101100 01100111 10100001 01101101
01100111 00011110 01001101 00000010 10111000 10111011 11001111 10011111
01011001 11110101 01010101 00110011 10000001 10100111 00011100 10011000
01111001 10111001 10101110 00010111 11011011 10111101 01101000 00101110
10000101 00100100 11010101 11111110 00101010 10010101 11001111 00011100
00111001 10111100 10111000 10000010 11010010 10101110 01001100 10000000
11001101 10100110 00000010 01000011 10011101
```

Then the same key would be used for the second block of 24 bits

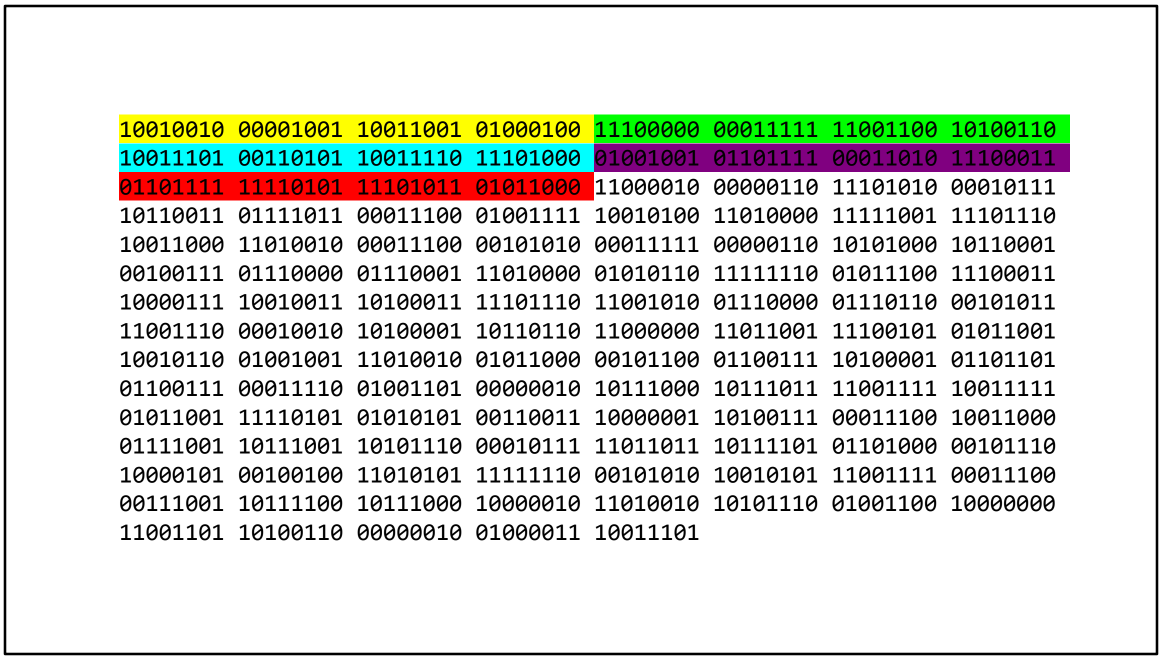
```
10010010 00001001 10011001 01000100 11100000 00011111 11001100 10100110
10011101 00110101 10011110 11101000 01001001 01101111 00011010 11100011
01101111 11110101 11101011 01011000 11000010 00000110 11101010 00010111
10110011 01111011 00011100 01001111 10010100 11010000 11111001 11101110
10011000 11010010 00011100 00101010 00011111 00000110 10101000 10110001
00100111 01110000 01110001 11010000 01010110 11111110 01011100 11100011
10000111 10010011 10100011 11101110 11001010 01110000 01110110 00101011
11001110 00010010 10100001 10110110 11000000 11011001 11100101 01011001
10010110 01001001 11010010 01011000 00101100 01100111 10100001 01101101
01100111 00011110 01001101 00000010 10111000 10111011 11001111 10011111
01011001 11110101 01010101 00110011 10000001 10100111 00011100 10011000
01111001 10111001 10101110 00010111 11011011 10111101 01101000 00101110
10000101 00100100 11010101 11111110 00101010 10010101 11001111 00011100
00111001 10111100 10111000 10000010 11010010 10101110 01001100 10000000
11001101 10100110 00000010 01000011 10011101
```

Third



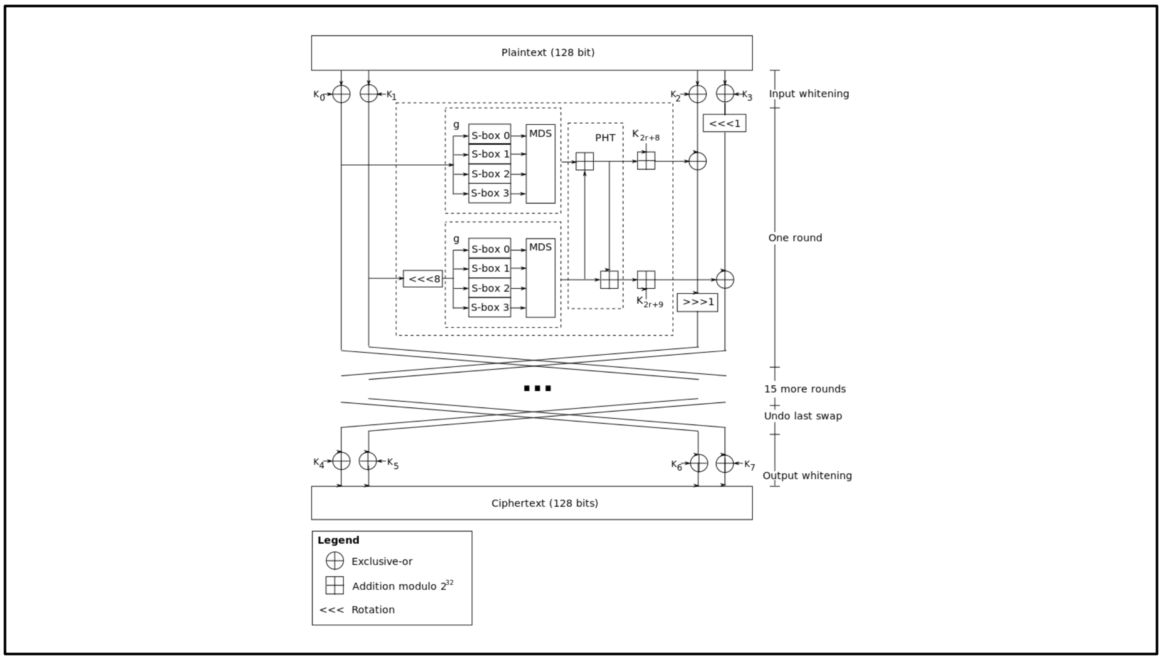
```
10010010 00001001 10011001 01000100 11100000 00011111 11001100 10100110
10011101 00110101 10011110 11101000 01001001 01101111 00011010 11100011
01101111 11110101 11101011 01011000 11000010 00000110 11101010 00010111
10110011 01111011 00011100 01001111 10010100 11010000 11111001 11101110
10011000 11010010 00011100 00101010 00011111 00000110 10101000 10110001
00100111 01110000 01110001 11010000 01010110 11111110 01011100 11100011
10000111 10010011 10100011 11101110 11001010 01110000 01110110 00101011
11001110 00010010 10100001 10110110 11000000 11011001 11100101 01011001
10010110 01001001 11010010 01011000 00101100 01100111 10100001 01101101
01100111 00011110 01001101 00000010 10111000 10111011 11001111 10011111
01011001 11110101 01010101 00110011 10000001 10100111 00011100 10011000
01111001 10111001 10101110 00010111 11011011 10111101 01101000 00101110
10000101 00100100 11010101 11111110 00101010 10010101 11001111 00011100
00111001 10111100 10111000 10000010 11010010 10101110 01001100 10000000
11001101 10100110 00000010 01000011 10011101
```

Fourth

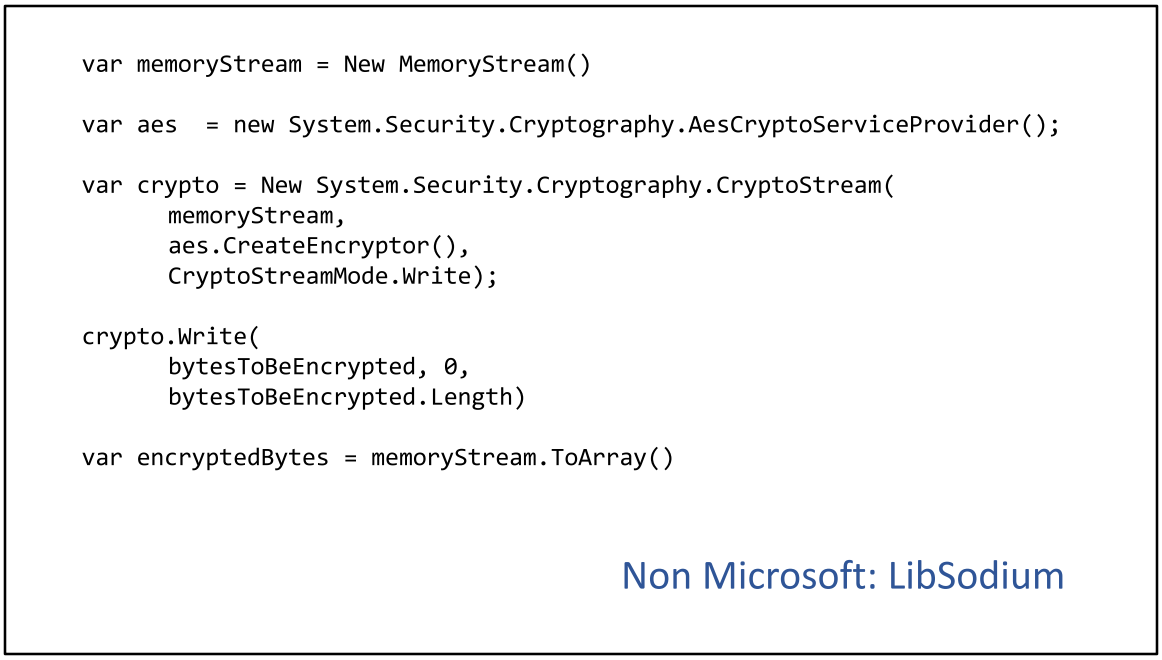


```
10010010 00001001 10011001 01000100 11100000 00011111 11001100 10100110
10011101 00110101 10011110 11101000 01001001 01101111 00011010 11100011
01101111 11110101 11101011 01011000 11000010 00000110 11101010 00010111
10110011 01111011 00011100 01001111 10010100 11010000 11111001 11101110
10011000 11010010 00011100 00101010 00011111 00000110 10101000 10110001
00100111 01110000 01110001 11010000 01010110 11111110 01011100 11100011
10000111 10010011 10100011 11101110 11001010 01110000 01110110 00101011
11001110 00010010 10100001 10110110 11000000 11011001 11100101 01011001
10010110 01001001 11010010 01011000 00101100 01100111 10100001 01101101
01100111 00011110 01001101 00000010 10111000 10111011 11001111 10011111
01011001 11110101 01010101 00110011 10000001 10100111 00011100 10011000
01111001 10111001 10101110 00010111 11011011 10111101 01101000 00101110
10000101 00100100 11010101 11111110 00101010 10010101 11001111 00011100
00111001 10111100 10111000 10000010 11010010 10101110 01001100 10000000
11001101 10100110 00000010 01000011 10011101
```

And so on. If the data isn’t evenly divisible by the block size -- see that last row? – then the data will be padded.



This is the illustration of TwoFish, a block cipher. I wanted to put AES up here, but couldn't find a diagram similar to this one. I think you get the idea – the cipher itself is a bunch of "fancy math" – some xoring and that sort of thing. In the binary I showed previously, we assumed to have a block size of 24. In this diagram, we have a block size of 128. So, this illustration is performed for each block. Sounds pretty good, right?

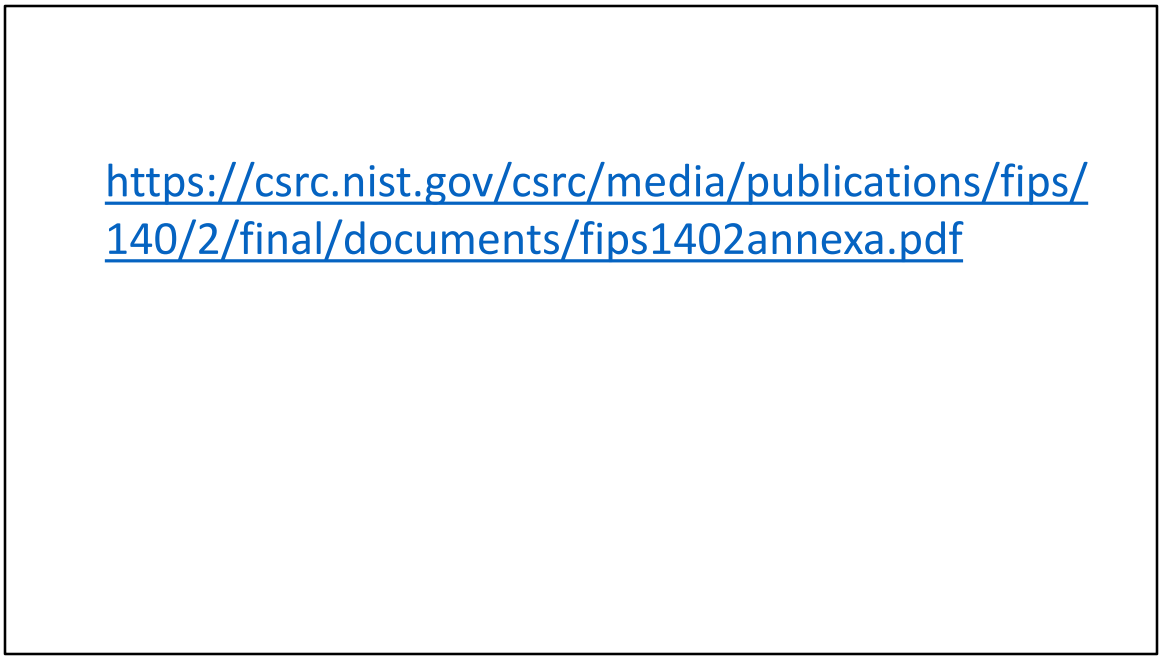


```
var memoryStream = New MemoryStream()

var aes  = new System.Security.Cryptography.AesCryptoServiceProvider();

var crypto = New System.Security.Cryptography.CryptoStream(
      memoryStream,
      aes.CreateEncryptor(),
      CryptoStreamMode.Write);

crypto.Write(
      bytesToBeEncrypted, 0,
      bytesToBeEncrypted.Length)

var encryptedBytes = memoryStream.ToArray()
```

Non Microsoft: LibSodium

You might be tempted to do something like this. Use a well known crypto library. I have some C# code here – I'm using the System.Security.Cryptography Library. If you're not a C# developer, I've heard nothing but good things about LibSodium. I've never used it though, so that's about as much as I can say about it.

Side Story – on a Windows Machine there is check-box somewhere deep down inside of Group Policy that says "use only FIPS compliant encryption". If you check that the OS does it's best to block any non FIPS approved algorithms. That means if you try to implement RJ (that dutch algorithm), windows will block execution of the application when that method tries to execute – and that's despite it being a part of the .NET framework. Similarly, SSL will not work on that machine. TLS will, but SSL will not. If you're writing an application and you want it to be deployed in a US Department of Defense setting, keep in mind that that group policy box is usually checked. I have no idea if LibSodium will work or not.



https://csrc.nist.gov/csrc/media/publications/fips/140/2/final/documents/fips1402annexa.pdf

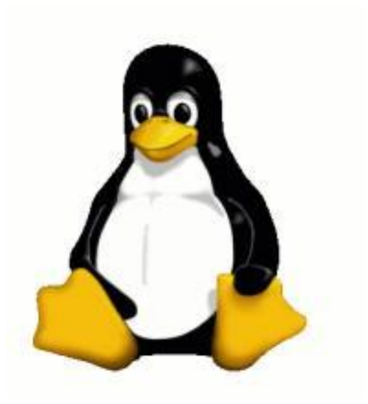
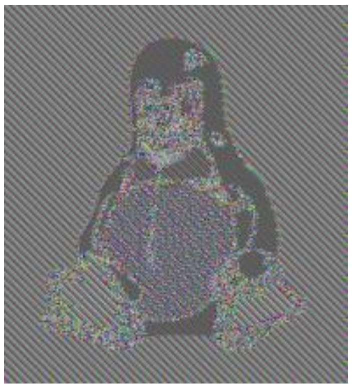
If you'd like to see a list of what those FIPS compliant algorithms are, you can see them here.

```
var memoryStream = New MemoryStream()

var aes  = new System.Security.Cryptography.AesCryptoServiceProvider();

var crypto = New System.Security.Cryptography.CryptoStream(
      memoryStream,
      aes.CreateEncryptor(),
      CryptoStreamMode.Write);

crypto.Write(
      bytesToBeEncrypted, 0,
      bytesToBeEncrypted.Length)

var encryptedBytes = memoryStream.ToArray()
```

Coming back to this – it would not be unreasonable for a developer to think “Hey, this is AES, which is good and it’s FIPS approved… so this should be solid, right?”

I mean, I’ve been guilty of this before.

This isn’t good enough because if two blocks are the same, then their encrypted values will be the same as well.

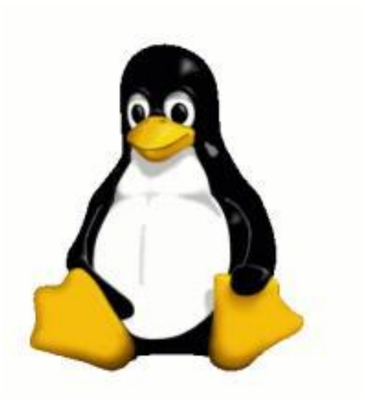
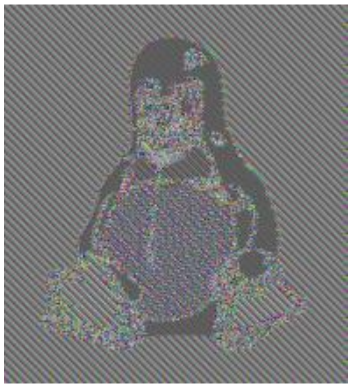



For example, if this is the input to a block cipher, then the output might look something like this.

And speaking of predictable input…. And attacker who is trying to break your encryption might not know the entirety of the message, but they might know part of it. For example, some file types have large sections of them that are all zeros.

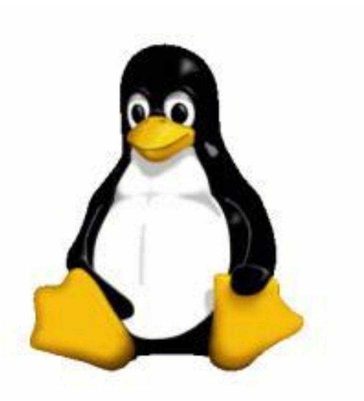
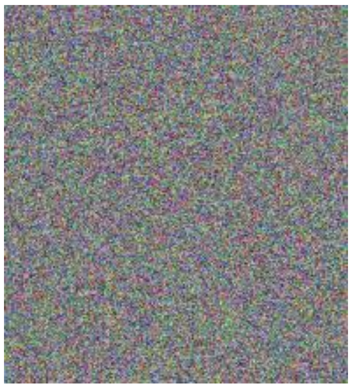
Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

If I recall correctly, what helped Alan Turing break the Nazi Encryption during World War II is that there was some predictability to most of the encrypted messages – for example, just about every message ended with “Heil Hitler”. This is a still from the movie The Imitation Game, which , if you haven’t seen it, then I highly recommend. I really enjoyed it.

This is there the concept of Block Cipher Mode comes in. In this example, ECB, which stands for Electronic Code Book, was used. There are several other modes. A good mode will give you this.

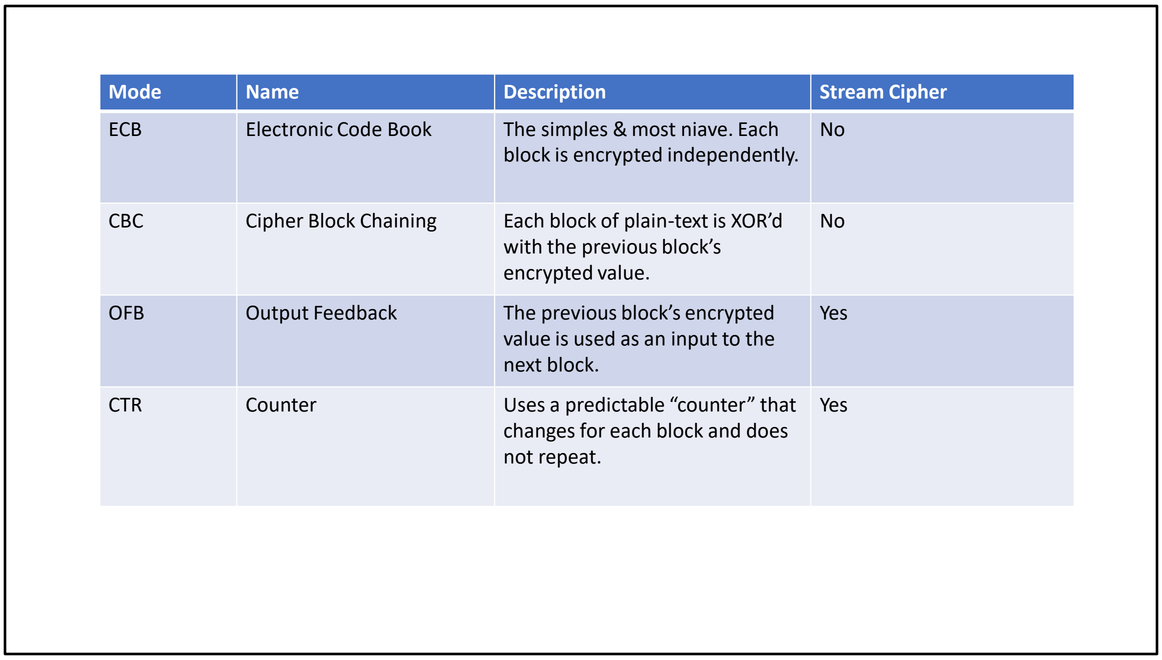
Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

```
var memoryStream = New MemoryStream()

var aes  = new System.Security.Cryptography.AesCryptoServiceProvider();

aes.Mode = CipherMode.CBC

var crypto = New System.Security.Cryptography.CryptoStream(
      memoryStream,
      aes.CreateEncryptor(),
      CryptoStreamMode.Write);

crypto.Write(
      bytesToBeEncrypted, 0,
      bytesToBeEncrypted.Length)

var encryptedBytes = memoryStream.ToArray()
```

Looking back at our C# example, you can specify the mode pretty easily by adding a line of code.

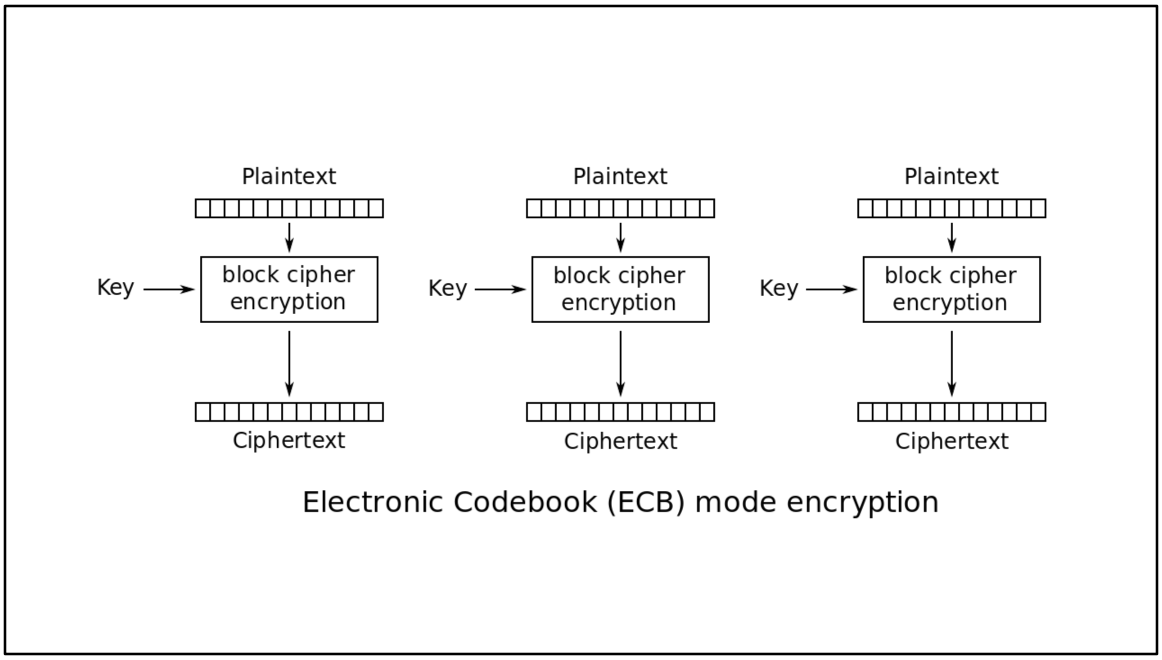


Mode	Name	Description	Stream Cipher
ECB	Electronic Code Book	The simples & most niave. Each block is encrypted independently.	No
CBC	Cipher Block Chaining	Each block of plain-text is XOR'd with the previous block's encrypted value.	No
OFB	Output Feedback	The previous block's encrypted value is used as an input to the next block.	Yes
CTR	Counter	Uses a predictable "counter" that changes for each block and does not repeat.	Yes

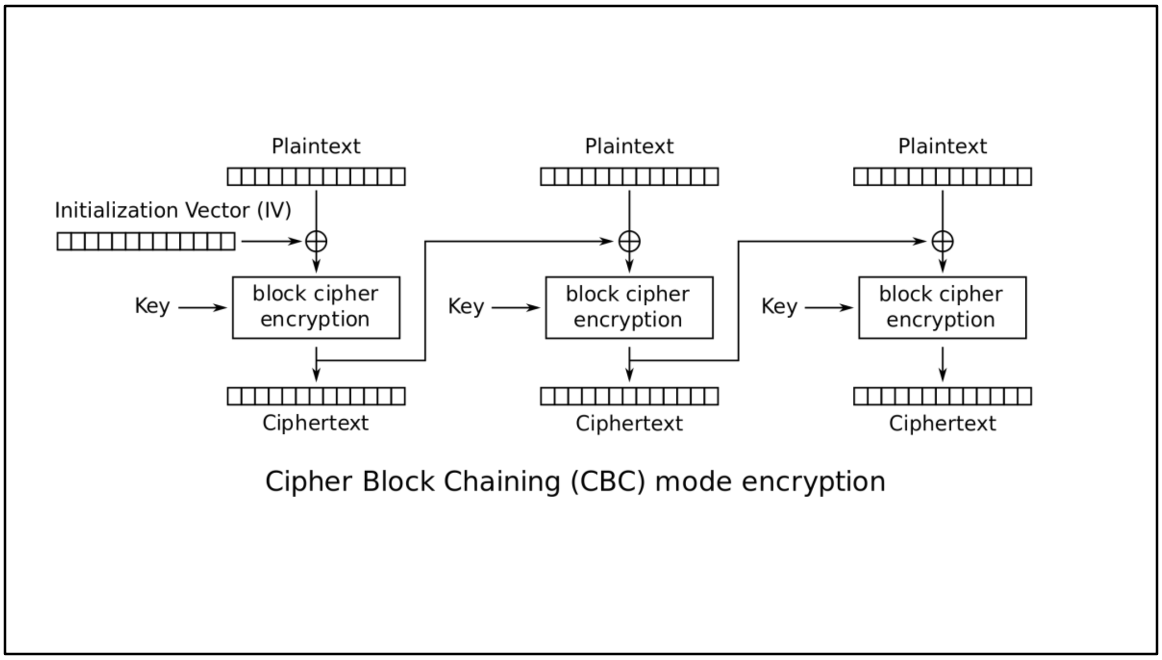
Here are 4 common Block Cipher Modes, and there are others. I'll give that a moment to sink in.

You'll notice that both OFB and CTR are considered Stream Ciphers. I'll be honest …. I don't fully understand the difference between Block Ciphers and Stream Ciphers. Some literature I read implies that they're separate, but this would imply that Block Ciphers can be in Stream Cipher mode. If we use AES in OFB mode, is it no longer a block cipher? You probably don't want to hear your present say "I don't really understand this stuff", but I hope that it drives home the point – DO NOT ROLL YOUR OWN CRYPTO.

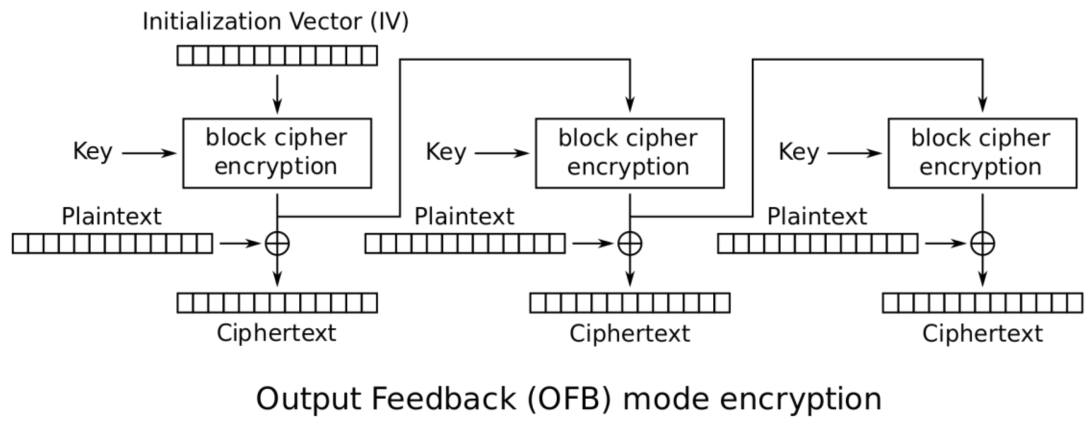
But if you must…..don't use ECB.



Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation



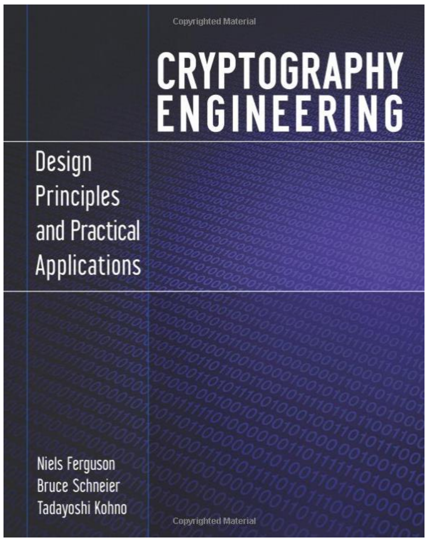
Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

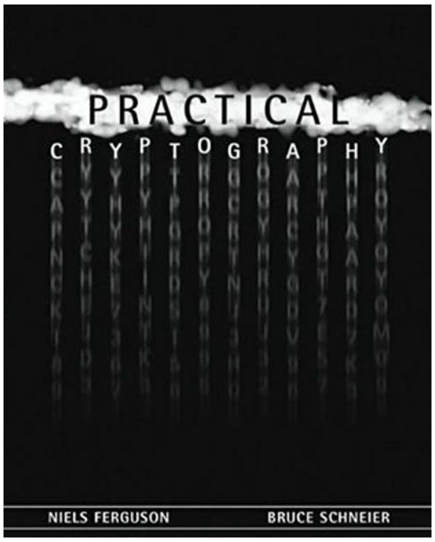


Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Choose Your Weapon!

So which mode should you use? I'll let you guess which one is ECB……

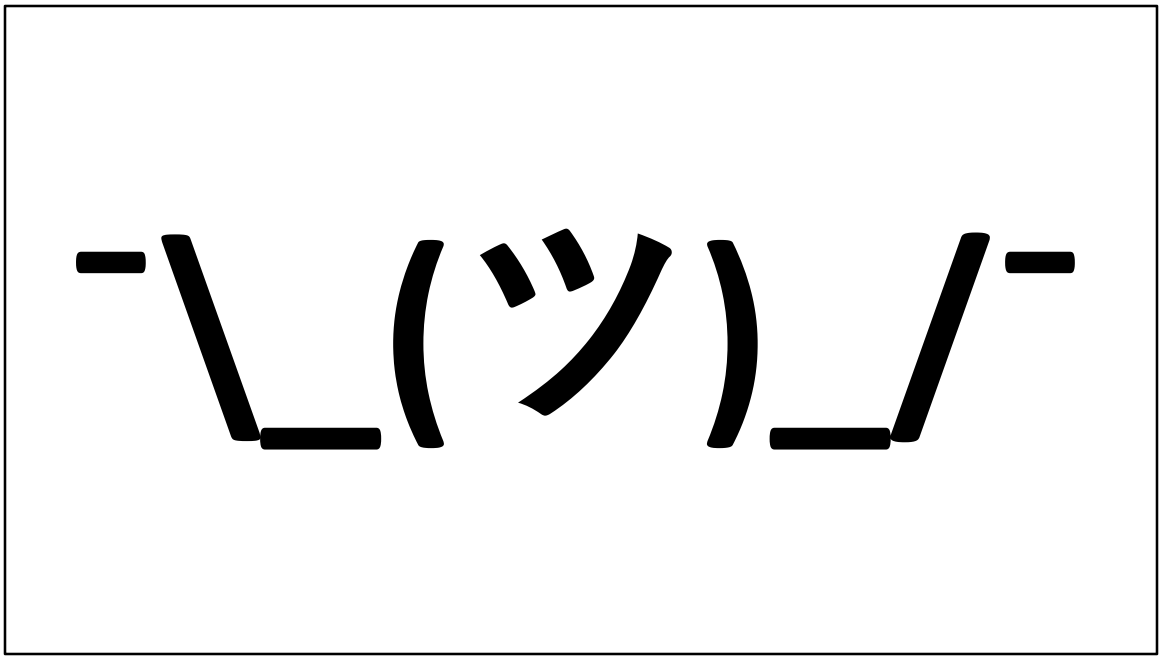




I've read both of these books, and remember – Crypto Engineering is just an updated version of Practical. They both discuss, in detail, the strengths and weakness of the 4 modes which I listed. In practical, which was published in 2003, they recommended CTR, but in Crypto, published in 2010, they recommended CBC.

I recently implemented some AES encryption for a project, and just this week it was undergoing a security audit and the CBC got me dinged because some static code analysis complained that it was a "weak" mode – susceptible to something called a Padding Oracle Attack.

So which one to use……



¯_(ツ)_/¯

So, I used to think that rolling your own crypto meant inventing your own cipher. Definitely don’t do that. But it’s so much more than that.

Even if you use the right cipher, you still must use it the right way…. And I didn't even get into initialization vectors (which is basically the same thing as a salt), or key management issues.

If you need encryption, do your best to use an existing solution. If you must write code, do some research to find a reliable library that handles as much for you as possible, or consider bringing in a consultant. If you work in a large organization with, you may have a product security team – reach out to them for guidance.

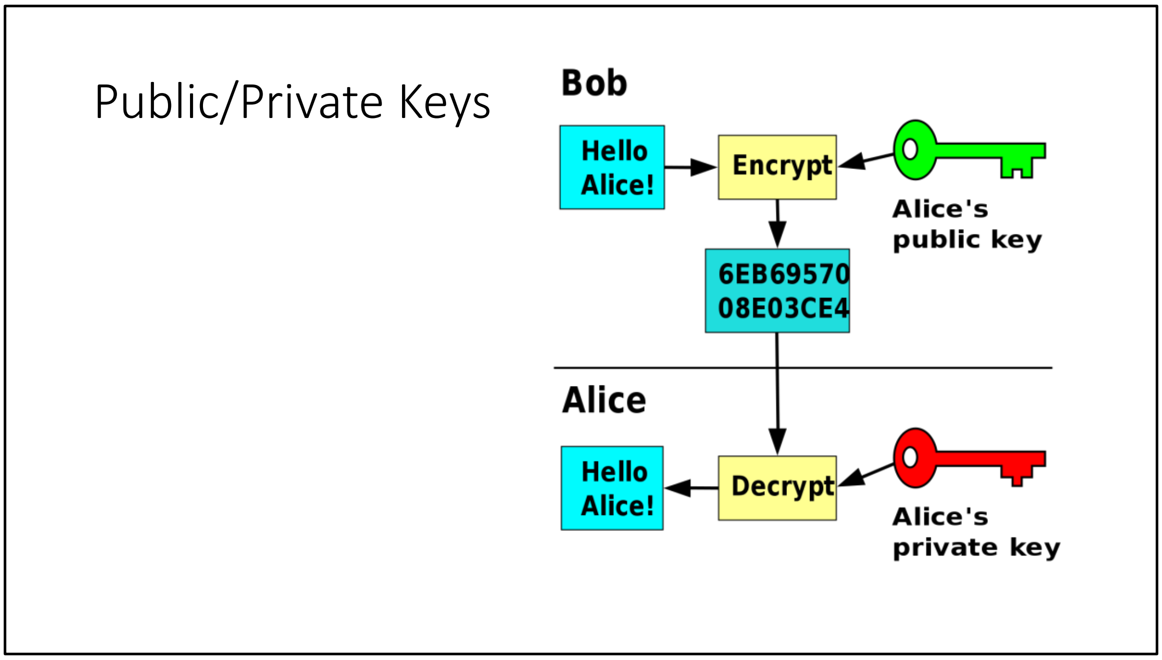
Symmetric

DES | Triple DES | AES
Blowfish | TwoFish

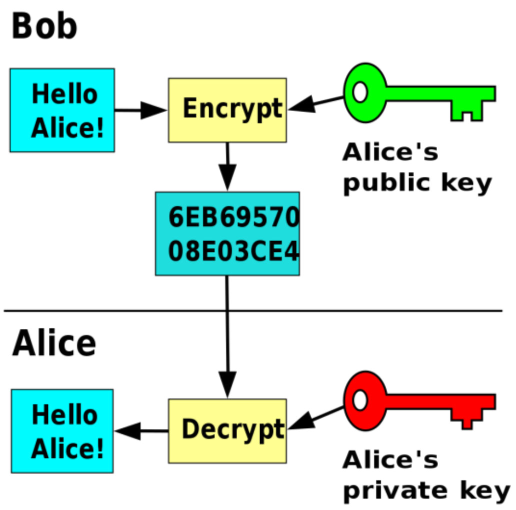
Asymmetric

RSA | ECC | PKI | PGP
Certificates | X.509

Okay, so…. That was all discussing Symmetric encryption. Let's dig into Asymmetric. Remember, symmetric uses the same key to encrypt and decrypt. Asymmetric uses one key to encrypt and another to decrypt.

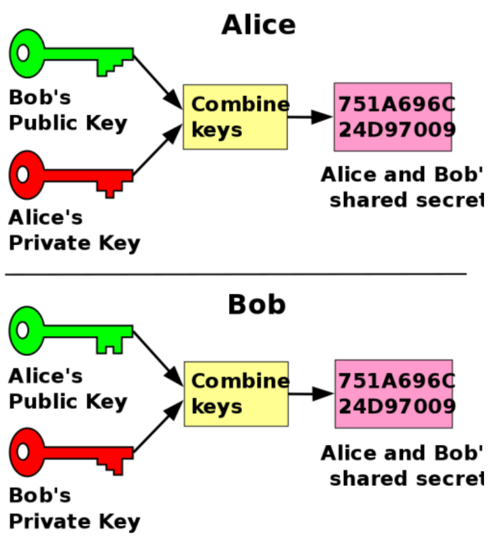


Public/Private Keys

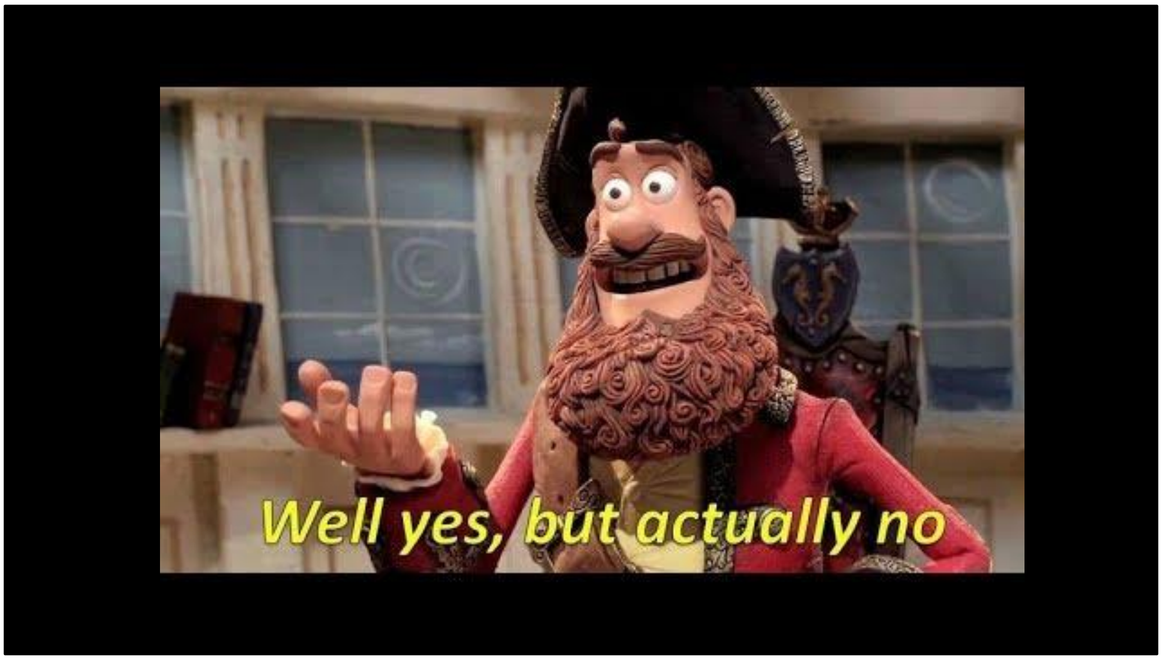


In Public Private key encryption, the idea is that everyone has two keys – one which they share with everybody and one which they keep only to themselves. So, if you want to send me a message that only I can read, you ask me for my public key. I give it you, and you use that to encrypt the message you'd like to send me. That message can then only be decrypted with the private key that I keep secret. This might sound like a great solution, but it's not so simple. Asymmetric encryption such as this is significantly slower than Symmetric. There are also message size limits. For example, with RSA encryption, a key of 2048 bits can encrypt, at most, only 245 bytes.

Public/Private Key Exchange

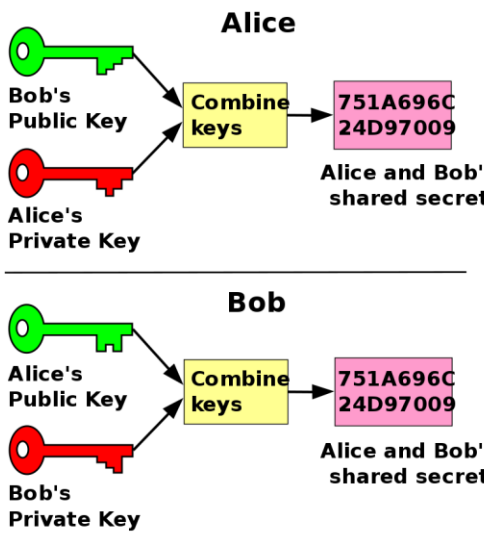


Which brings us to Public Private key exchange. What happens here is that the asymmetric cryptography is used to agree upon a shared symmetric encryption key and algorithm. Perhaps Alice and Bob both agree to communicate from here on out using AES and a 256 bit key. This is how HTTPS, SSL, TLS, all work. Same with your VPN or some system using IPSec.

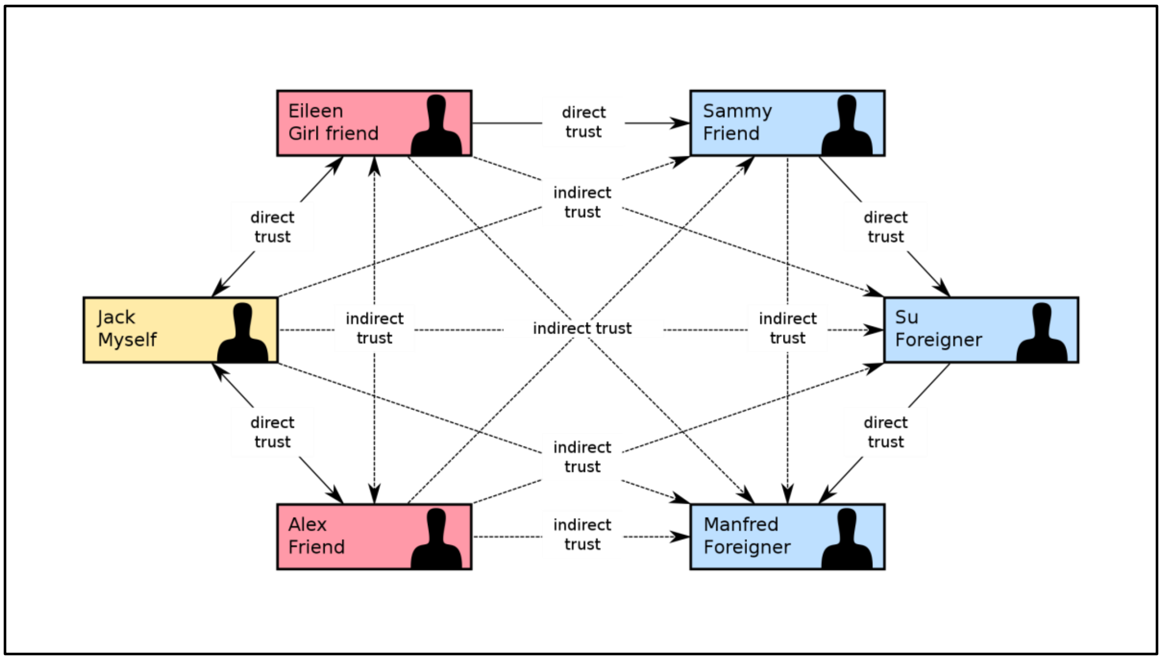


So when people ask if these things are symmetric or asymmetric…. They're actually both. The communication starts off as asymmetric as part of a handshake and to establish symmetric.

Public/Private Key Exchange



So, suppose that I'm Alice in this scenario. With this key exchange, I can be sure that my communication with whoever claims to be Bob is secure. But how do I know that the person claiming to be Bob really is Bob? If I'm logging into my Google over HTTPS, can I be sure that the server on the other end is actually Google and not just someone saying "yep, I'm Google?"

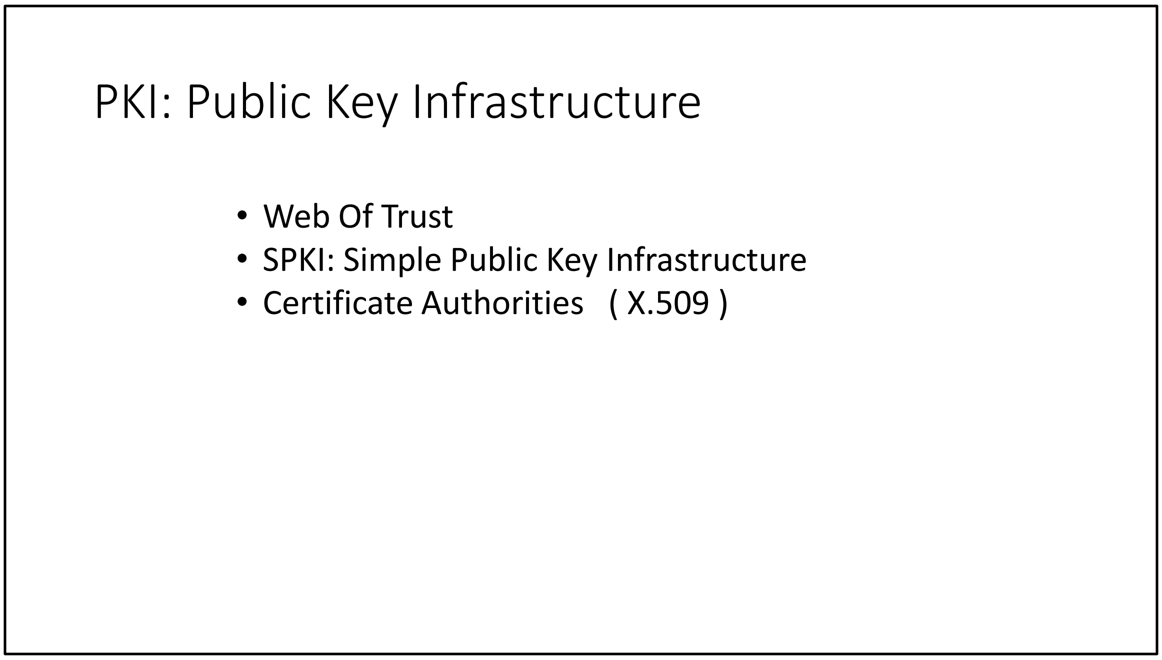


There is where a web of trust comes into play. If you know me and trust that I who I say I am, and your friend trusts you, then they can trust my claim of identity by proxy of you. This can be done by way of signing people's public keys – which is sometimes done at "key signing parties". But this doesn't scale well….

HEY, I JUST GOT HOME FROM THE PARTY
THE ONE WITH THE IRC FOLKS?
YEAH.
HOW WAS IT?
GOT TOO DRUNK. I SCREWED UP, BAD.
WHAT HAPPENED?
THERE WAS A GIRL. NO IDEA WHO SHE WAS. DON'T EVEN KNOW HER NAME. I WAS TOO DRUNK TO CARE.
AND WHAT, YOU SLEPT WITH HER?
NO.
I SIGNED HER PUBLIC KEY.
SHIT, MAN.

https://xkcd.com/364/

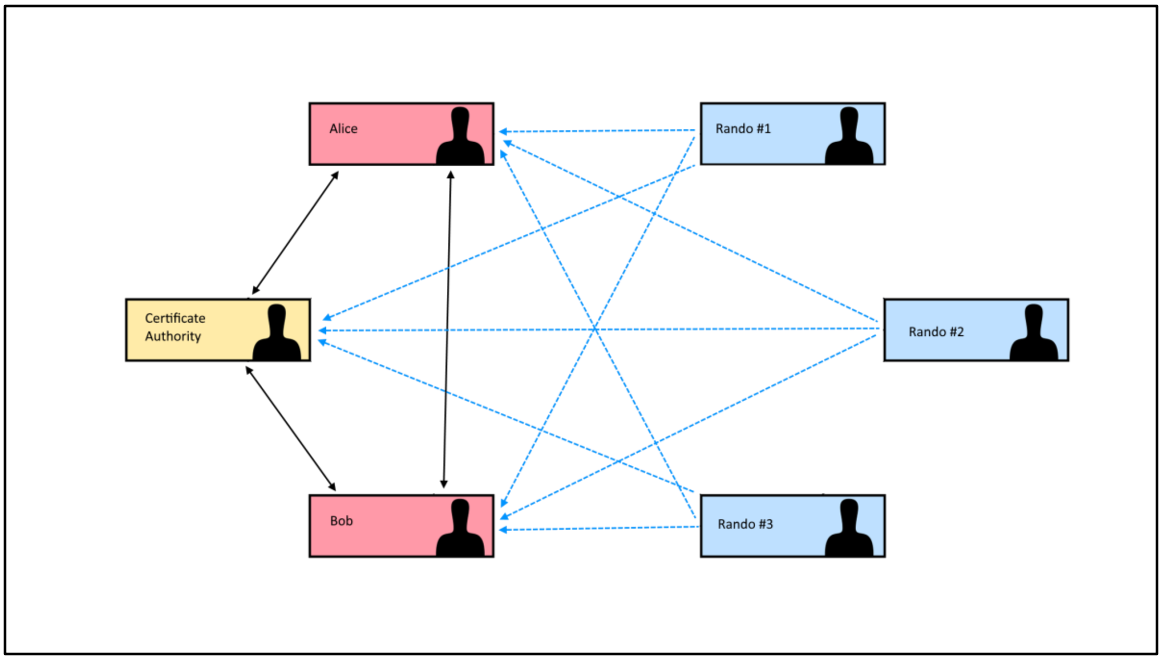
And not everyone is as ….careful… in signing public keys. I may know you and trust you to be who you say you are…. But can I trust you to be selective in who *you* trust.



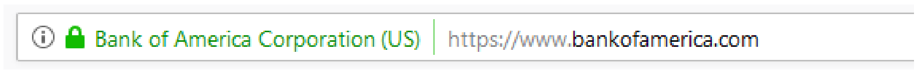
PKI: Public Key Infrastructure

- Web Of Trust
- SPKI: Simple Public Key Infrastructure
- Certificate Authorities (X.509)

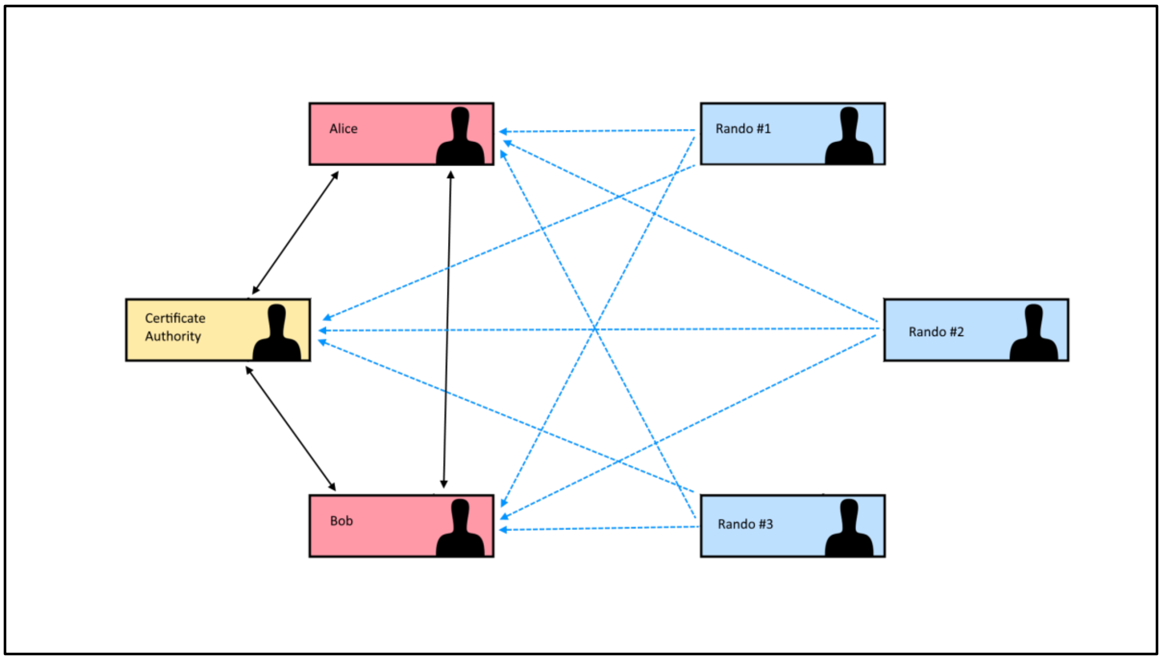
Which brings us to public key infrastructure. There are 3 main types of PKI. We briefly spoke about the Web of Trust, which is what we see with PGP encryption. There's SPKI, but with limited time left I'd like to discuss Certificate Authorities. Certificate Authorities are, in a broad sense, very similar to the Web of Trust, except with very specific individuals being designated as a trusted third party.



A Certificate Authority is trusted by everyone in the web of trust, but not everyone is trusted by everyone else. In this graph, the black lines with arrows on both ends indicate mutual trust. The blue lines with arrows at one end indicate one-way trust. Rando #1 trusts the Certificate Authority and Alice, but the CA and Alice don't trust Rando #1. You may ask, why do we all trust the CA? How was that trust established in the first place? Let's think about Windows, or your cell phone – when you get these devices, they come pre-packaged with a Public Certificate from the Certificate Authority. The Certificate Authority then issues (usually sells) Certificates to parties like Alice and Bob who can prove they are who they claim to be. This is "proof" is usually done by non cryptographic means. I've had to go to a notary before to get an SSL Cert. I co-founded a startup a while back, and we decided to get an EV Cert – which is extended validation.

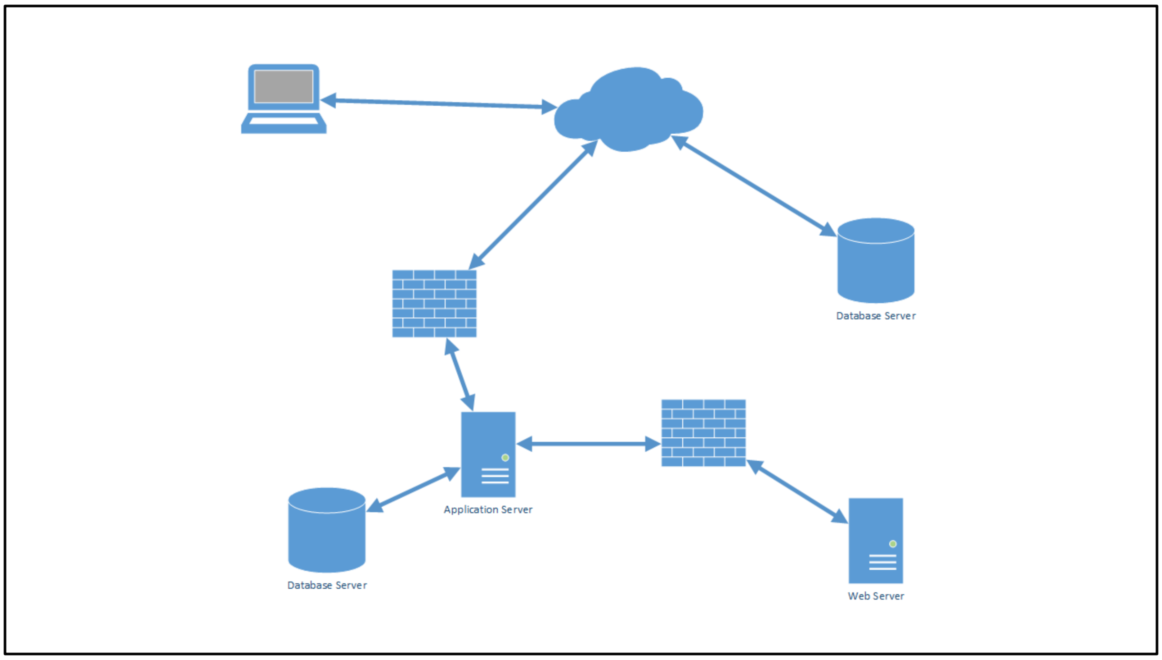

Bank of America Corporation (US) | https://www.bankofamerica.com

I co-founded a startup a while back, and we decided to get an EV Cert. EV stands for Extended Validation. With these certificates, you see the business name in the URL bar of your browser, not just the green lock. We had to get our company's articles of incorporation, or some other business filings from the state in which we were incorporated, and then get our corporate lawyer to draft up a letter with the exact verbiage requested by the Certificate Authority and have him sign it, as well as have him provide his Attorney License number.

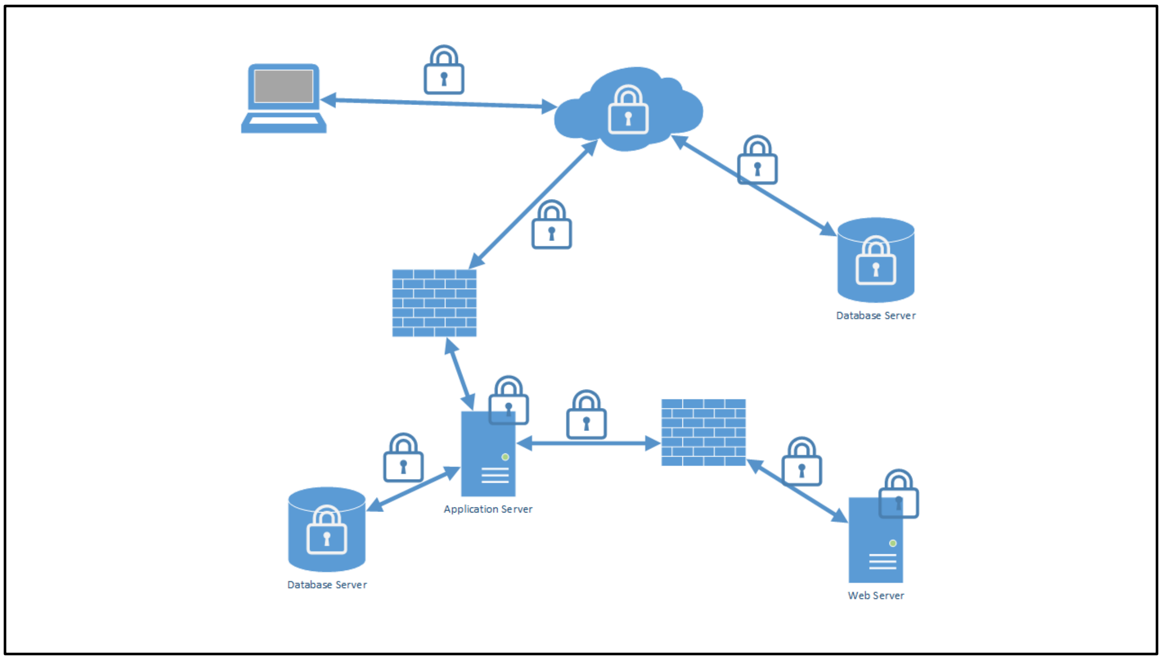


When the Certificate Authority is satisfied with the proof of identity, they will issue a certificate to Alice of Bob with the Certificate Authority's signature on it. Because all these Rando's trust the Certificate Authority, and they see that Alice and Bob have certificates with the CA's signature, they therefore trust the identity of Alice and Bob.

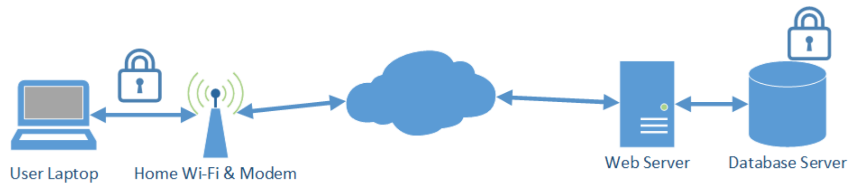
I’d like to change gears a little bit. So far, we’ve gone over the implementation details of encryption. Let’s talk about how to integrate it.



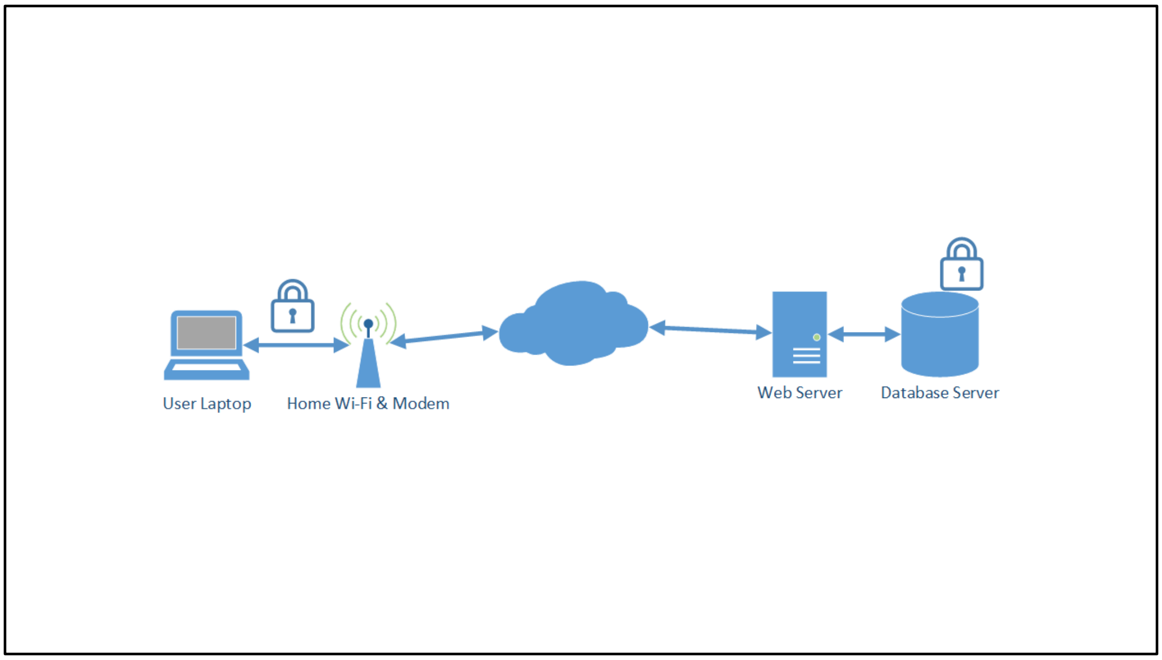
Let’s pretend we have a network diagram like this. It’s got a cloud. Okay, good… gotta have a cloud. We have a couple of data base servers, a web server, and application server, some firewalls, and what looks like a user's laptop. By the way, I kind of hate network diagrams like this – they don’t really tell me a whole lot. But anyway… let’s suppose someone says “Can you put a lock icon everywhere there is encryption?”



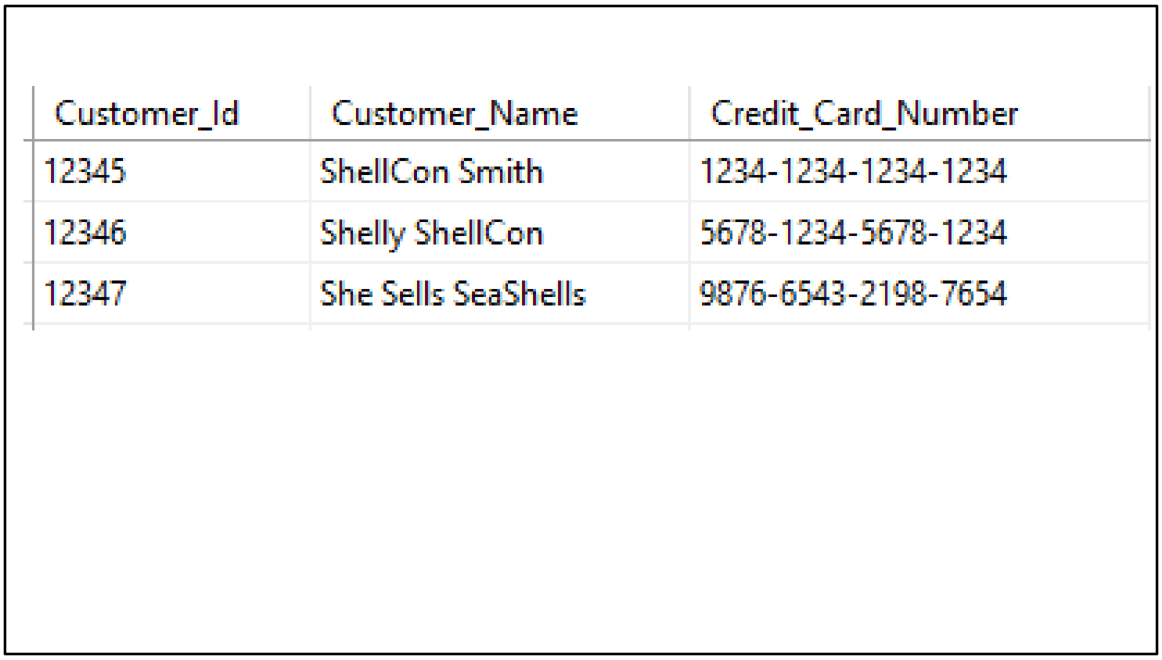
I've literally seen network diagrams that look like this. This is just disgusting to me – it doesn't help us understand the actual security posture of this system or network.



Let’s look at a simplified network. This is a user, like you or I with a laptop connected to a Wi-Fi access point. We’re visiting some website online, and the website had a SQL database with user info in it. Maybe there’s a lock on the database because we have a check list that says “you must encrypt user data at rest”. At rest means “on disk”. Okay, done. We good, right?

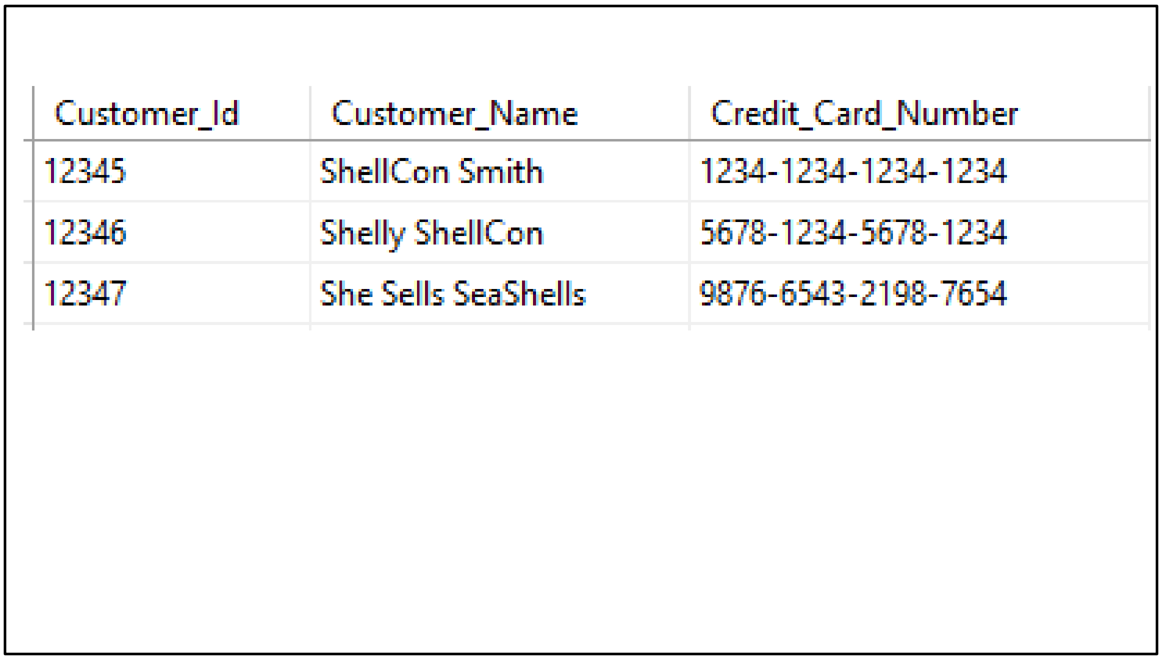


From this diagram, can we tell anything about that encryption? Maybe it’s full disk encryption? Maybe it’s Transparent Database Encryption? Or Column Level Encryption? It could be a combination. From this diagram, we have no idea, and thus it really says nothing about what we’re protecting ourselves against. Let’s suppose that it is disk level encryption.



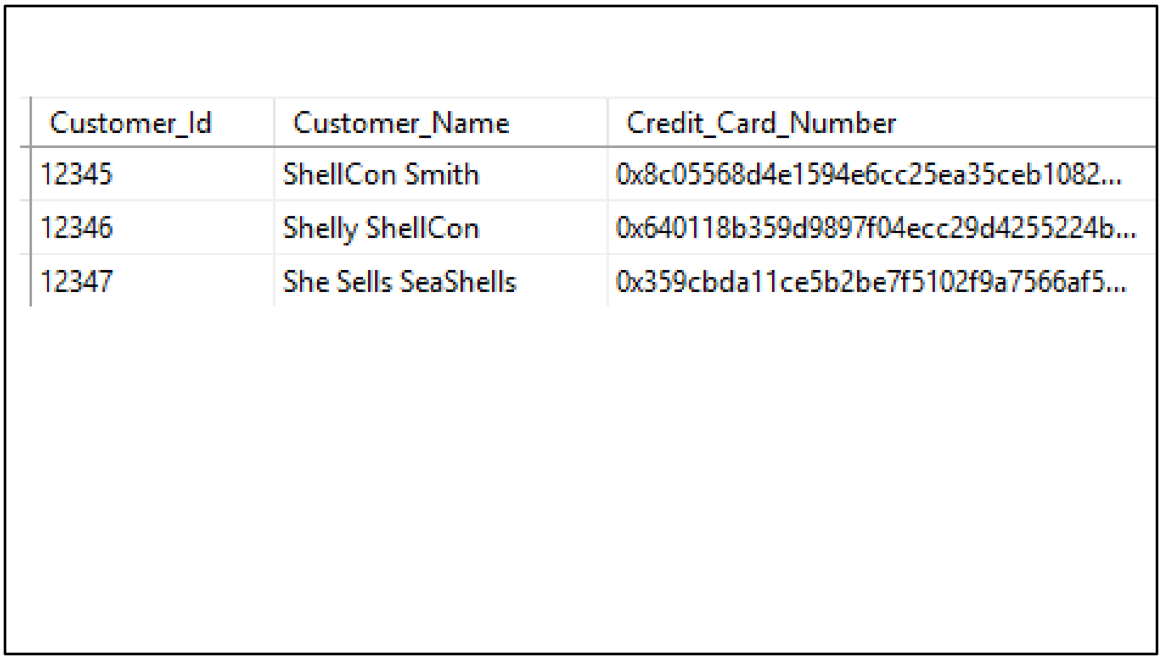
Customer_Id	Customer_Name	Credit_Card_Number
12345	ShellCon Smith	1234-1234-1234-1234
12346	Shelly ShellCon	5678-1234-5678-1234
12347	She Sells SeaShells	9876-6543-2198-7654

If you have Disk Level encryption turned on, then all of your data on disk is encrypted. But if you open your SQL manager and run select * from customers, you'll see this. This is what someone logged onto the machine will see, so your disk encryption doesn't protect you in that case. This is certainly what your application, which is running in your web server, will see. This means that if your web app is vulnerable to SQL injection attacks, then your disk encryption won't protect you either. The disk encryption protects you from someone stealing your hard drives and copying data off them.



Customer_Id	Customer_Name	Credit_Card_Number
12345	ShellCon Smith	1234-1234-1234-1234
12346	Shelly ShellCon	5678-1234-5678-1234
12347	She Sells SeaShells	9876-6543-2198-7654

Let’s suppose you don’t have disk encryption, but you are running Microsoft SQL Server Enterprise, which offers Transparent Data Encryption. We don’t typically think of databases as files, but and the end of the day, they are. Transparent Data Encryption will encrypt your whole database file on disk, but it will be transparent to you as an authenticated SQL user – again, this includes your web app, which will see this plaintext.



Customer_Id	Customer_Name	Credit_Card_Number
12345	ShellCon Smith	0x8c05568d4e1594e6cc25ea35ceb1082...
12346	Shelly ShellCon	0x640118b359d9897f04ecc29d4255224b...
12347	She Sells SeaShells	0x359cbda11ce5b2be7f5102f9a7566af5...

Let’s suppose that you have Column Level Encryption enabled and have encrypted the credit card column. If you open your SQL Browser, you’ll see something like this if you select * -- you can see the credit card numbers are encrypted. This also means that an attacker running arbitrary SQL commands via SQL Injection will also get this.

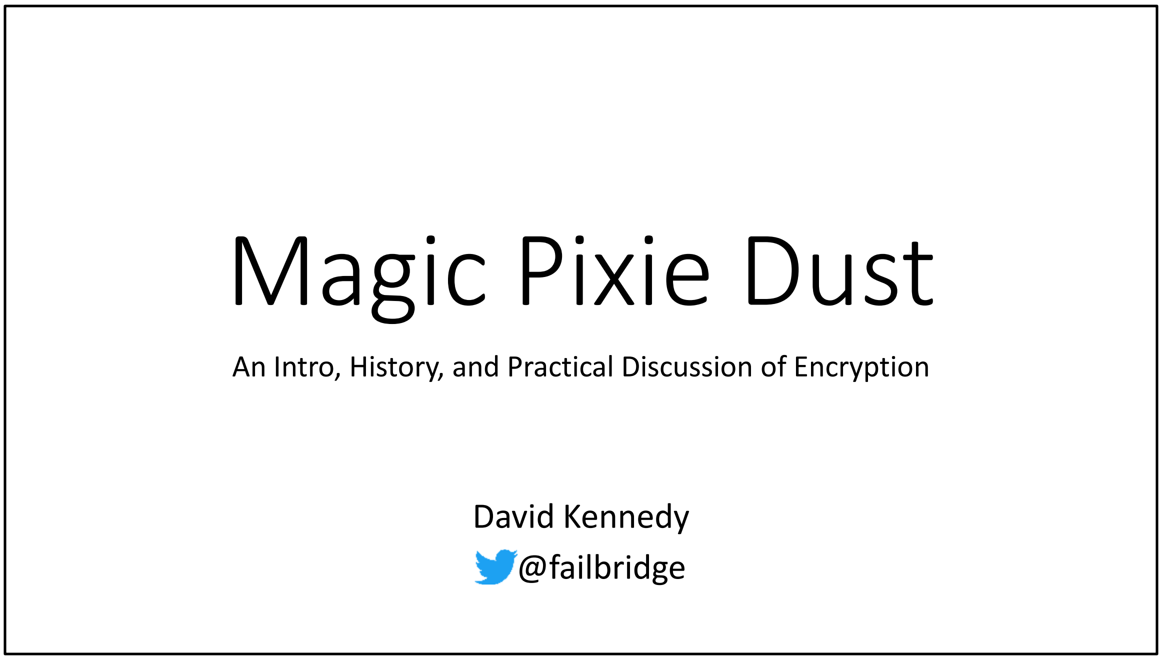
Topics not discussed….

- More on Block Ciphers
  - Initialization Vectors
  - Key Management
  - Message Authentication Codes (MAC)
  - Side-channel attack details
- Elliptic Curve Cryptography
- Hashing

Conclusion

- Encryption is not magic pixie dust
  - Don’t roll-your-own
    - Use well established libraries, services, or experts/consultants
  - Just because data’s encrypted doesn’t mean it’s safe
    - Understand your threat model
    - Use the correct encryption methods, and use them correctly

I'M DAVID PUMPKINS.
ANY QUESTIONS???



Magic Pixie Dust

An Intro, History, and Practical Discussion of Encryption

David Kennedy
@failbridge