SHARKFEST '13

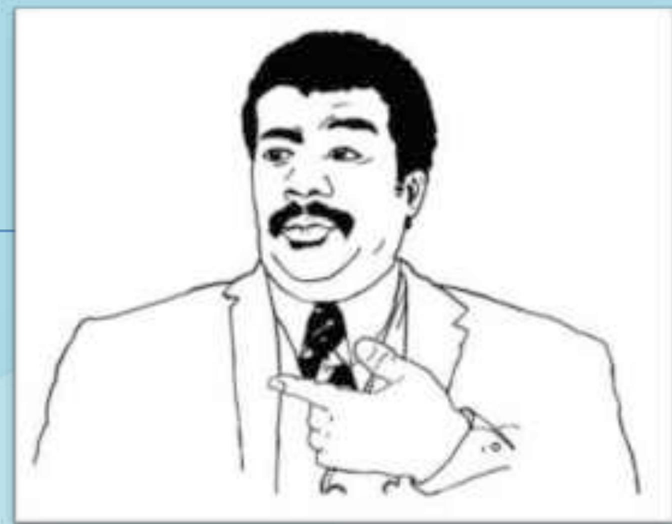
Wireshark Developer and User Conference

# Why is crypto so hard to get right?

Ron Bowes, Leviathan Security Group

Sharkfest 2013

# About me



- Ron Bowes
  - @iagox86
  - http://www.skullsecurity.org
  - ron.bowes@leviathansecurity.com
- Security consultant for Leviathan Security Group
- Founder/president of SkullSpace, Winnipeg's hackerspace
- Rockclimber
  - The best way to improve your self confidence is to hang 1000ft in the air – from an anchor you built yourself!
    - This is probably more common in California, but I'm from the prairies!

# Quick agenda

- History of crypto attacks
- A bunch of examples, with proofs of concept
  - Key re-use
  - Hash length extension
  - Padding oracle
- Some proposed solutions

# Why am I doing this?

- In my opinion, crypto is one of the most important technologies in the modern world, if implemented correctly
- Crypto implementation is hard
- I decided to teach myself attacks by writing tools
  - Before I knew it, I had enough to make an interesting talk!

# Why am I doing this?

- The four stages of competence:
  - **Unconscious incompetence** - When you don't know how bad you are or what you don't know.
  - **Conscious incompetence -** When you know how bad you are and know what steps you need to take to get better.
  - **Conscious competence** - When you're good and you know it (this is fun!)
  - **Unconscious competence** - When you're so good you don't know it anymore.
- I went to help people go from **Unconscious incompetence** to **Conscious incompetence**!
- Source: http://happybearsoftware.com/you-are-dangerously-bad-at-cryptography.html

# Why am I doing this?

- One more quote from the page, then I'll move on:

  **"Cryptography is perilous because you get no feedback when you mess up**. For the average developer, one block of random base 64 encoded bytes is as good as any other."

  "You can get good at programming by accident. If your code doesn't compile, doesn't do what you intended it to or has easily obvervable [sic] bugs, you get immediate feedback, you fix it and you make it better next time."

  "*You cannot get good at cryptography by accident*."

# Why am I doing this?

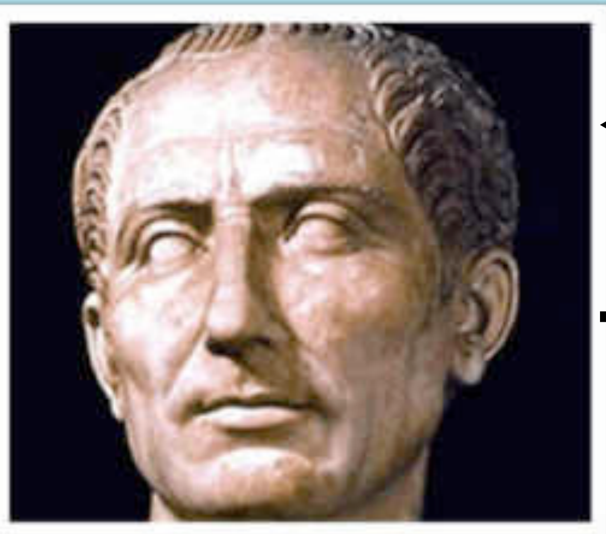
- Another great talk: "If You're Typing The Letters A-E-S Into Your Code, You're Doing It Wrong"
  - http://www.cs.berkeley.edu/~daw/teaching/cs261-f12/misc/if.html
  - Hey look, it's berkeley!
  - Same information, same conclusion, as both the previous post and this talk!
  - A fun, interesting read! One of my first forays into crypto.

The somewhat accurate

# HISTORY OF CRYPTO

# c. 75 BC: Caesar cipher

- Shift cipher
- 25 possible encodings (26, if you count ‘0’)
- Trivially bruteforced



J uijol nz gsjfoet bsf uszjoh up ljmm nf!

mpm

# Caesar – World War II: No developments

File: 1359396104253.jpg-(17 KB, 220x293, 220px-EnigmaMachineLabeled.jpg)

Anonymous (ID: kv30XfOr) 01/28/13(Mon)13:01:44 No.454378465

Hey 4chan, I need help writing a paper! Were there any important developments in cryptography between the Caesar Cipher and the Enigma Machine?

>> Anonymous (ID: nijl5aT4) 01/28/13(Mon)13:02:33 No.454378583

No.

The End.

You're welcome.

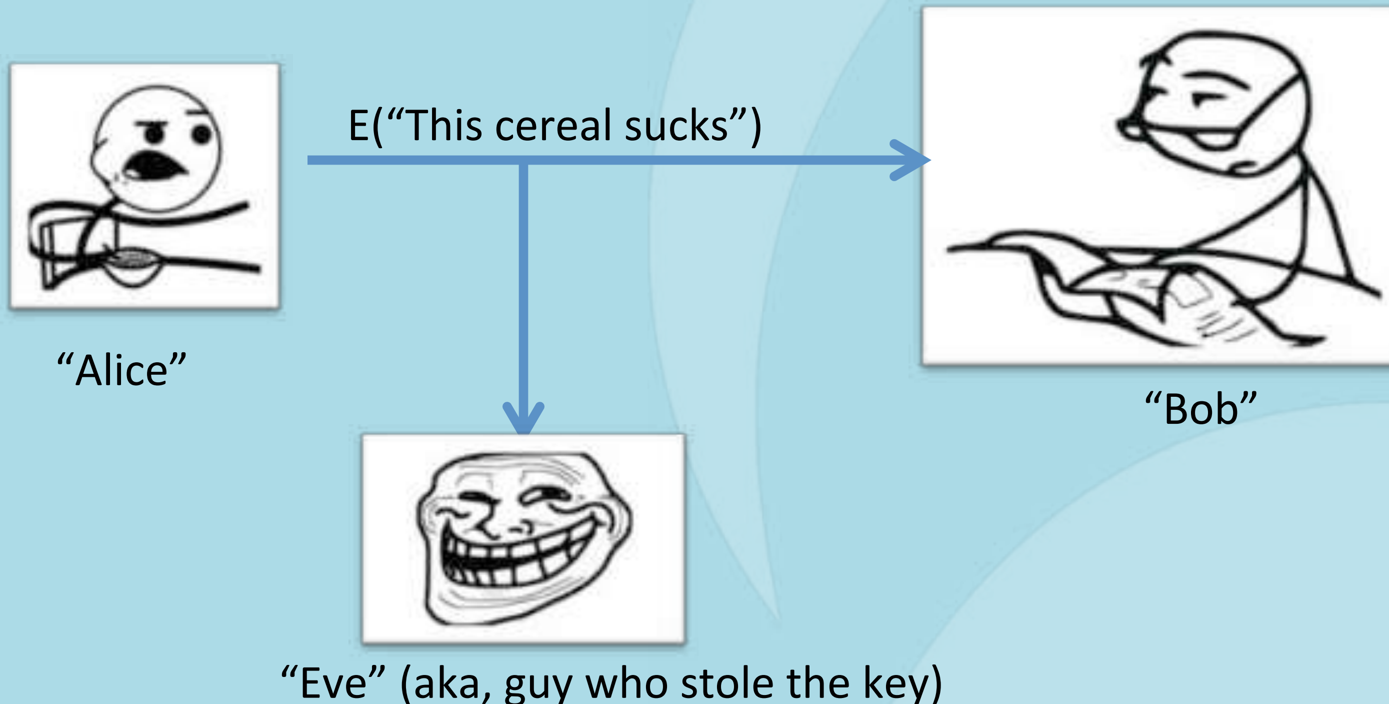
[Return] [Catalog] [Top] [Update] [ Auto]

# World War II: Enigma Machine

# Let’s get more modern

# 1970s: DES was invented!

- A symmetric-key block cipher
- Message could be decrypted by the intended recipient and everybody who's stolen the key



# Still 1970s: Along came DH and RSA

- Now both parties have to exchange keys with "Eve" (or each other) before they can communicate

# 1990s: Certification Authorities

- Now you can see if any of 100s of companies thinks the "Bob" is actually "Bob"

"Eve"

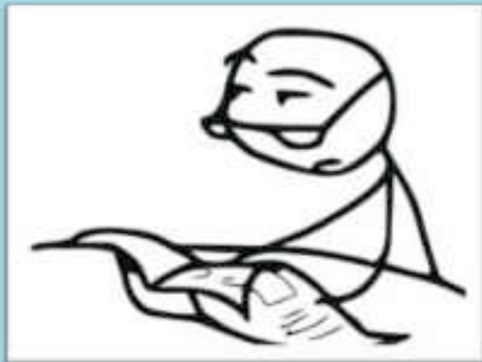
$K_{eve}$

"Alice"

Is this $K_{bob}$?

Sure, whatever

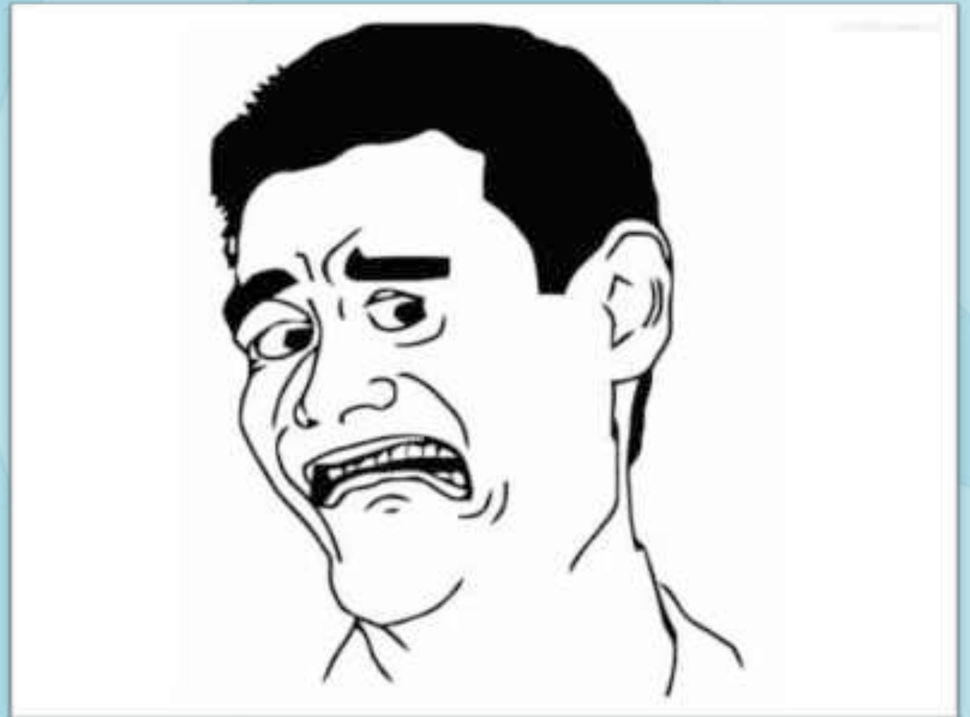
?



"Bob"

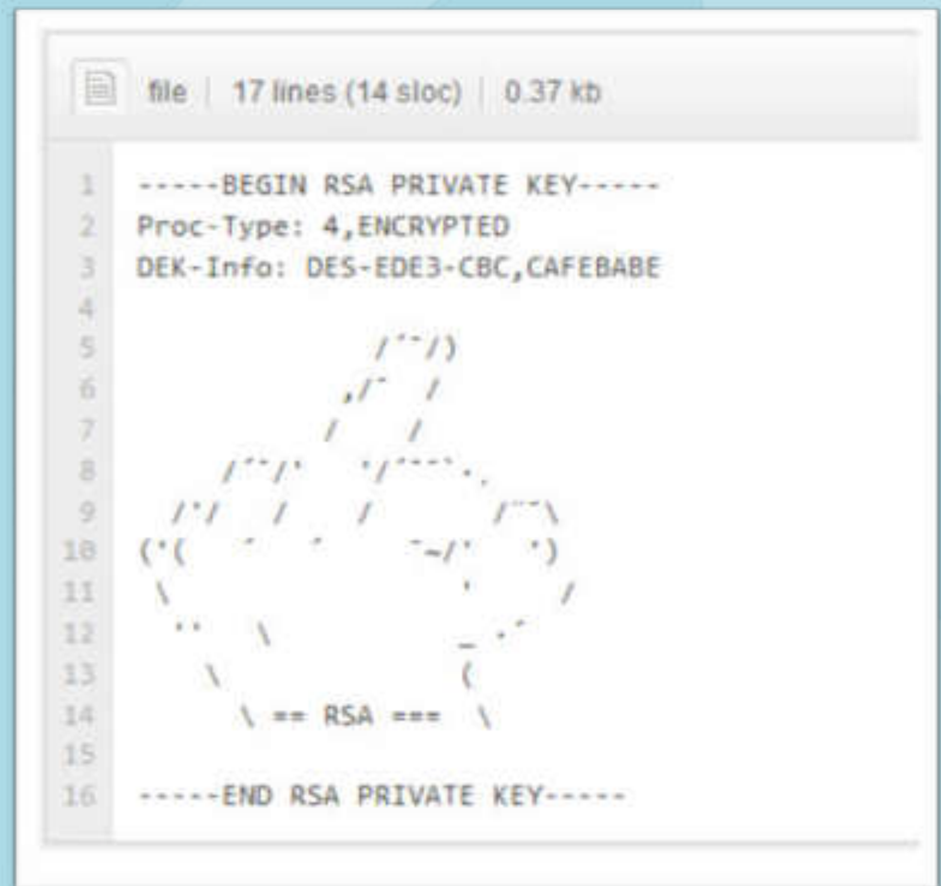
# 1990s: WEP

- While we're talking about Goatse…
- RC4 w/ 24-bit IV
- Using RC4 all kinds of wrong led to total compromise



# 2008: github (and other "Web 2.0" stuff)

- A new place for people to post private keys, passwords, and other confidential data



# Point?

- These days, encryption is rarely broken directly
- It’s broken by…
  - Implementation error (developer mistakes)
  - Operator error (end-user mistakes)
    - Document, key, codebook theft/leakage
  - Stupidity (aka, CAs)
  - Side-channel attacks
- The rest of this talk will be about indirect ways to break state-of-the-art crypto!

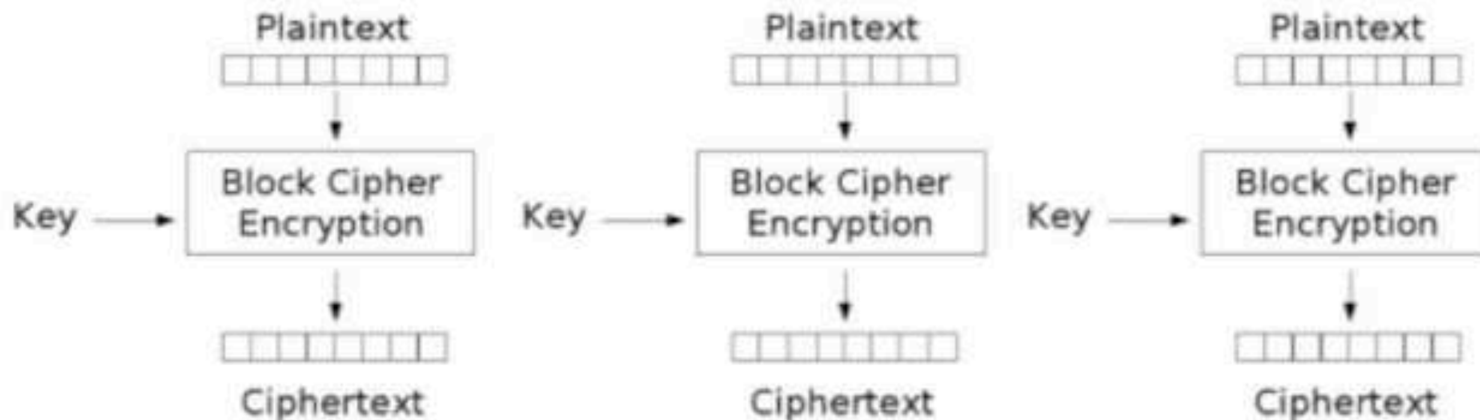
# IMPORTANT CONCEPTS

# Encryption

- The act of obscuring data using a secret key, such that only the intended recipient – and anybody else who manages to steal the key – can read it

# Encryption: Block cipher

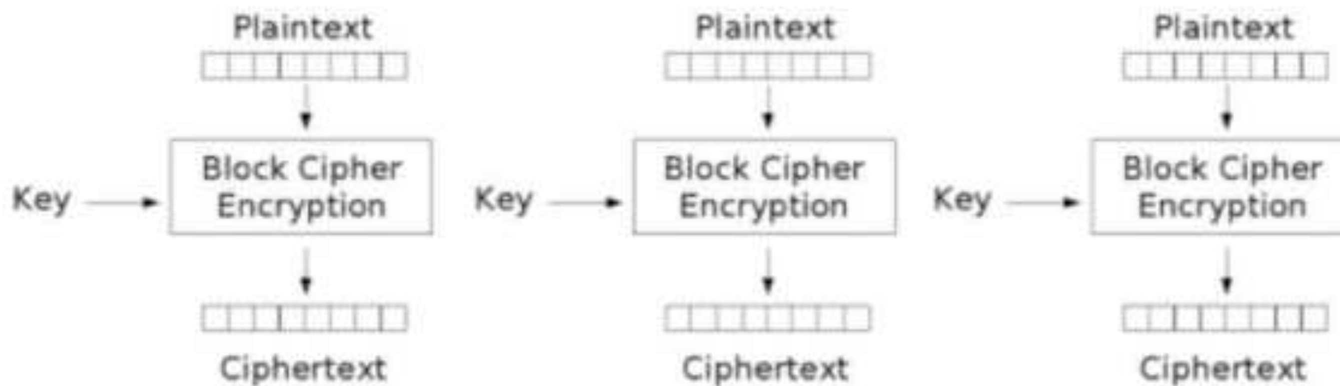
- Plaintext is broken into 8- or 16-byte blocks, each is encrypted individually
- Various "modes of operation" can be used to ensure that the ciphertext isn't repeated




Electronic Codebook (ECB) mode encryption

# Encryption: Block cipher modes of operation – ECB

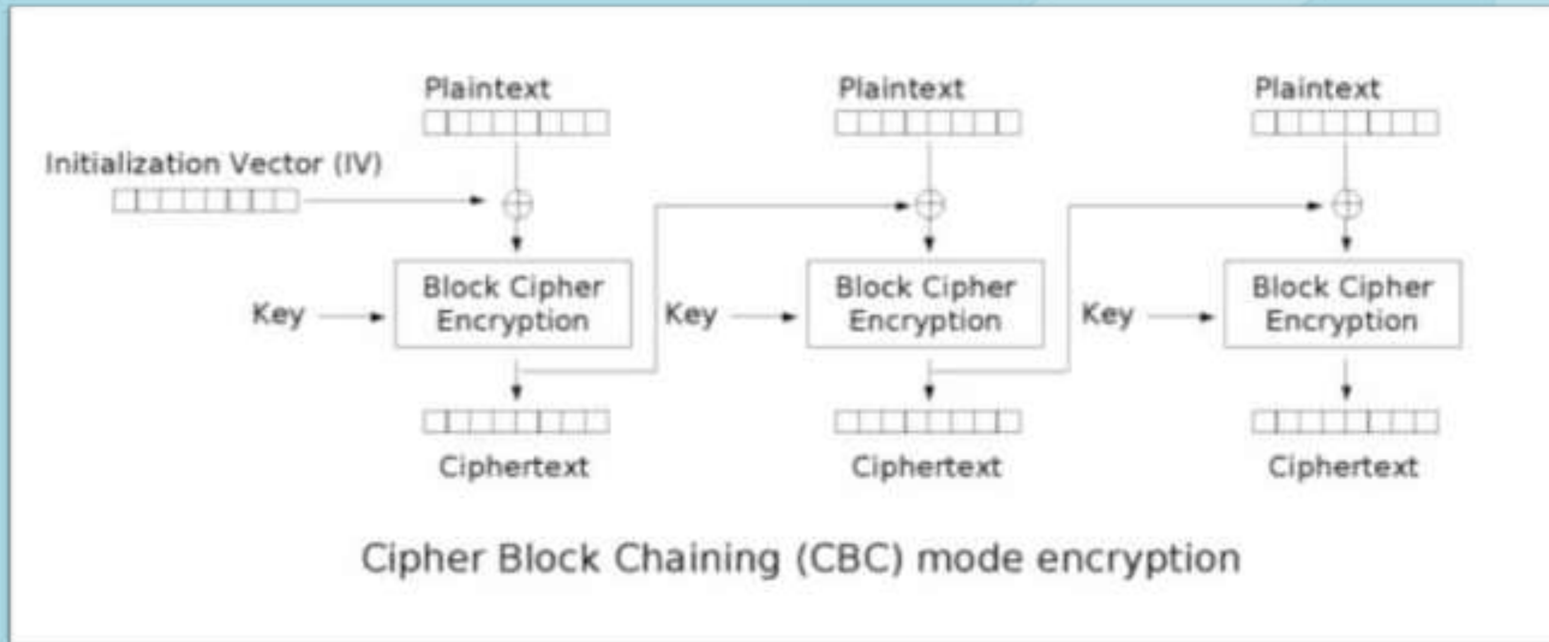
- "Electronic codebook" mode encrypts each block individually:




- This leads to problems like the famous "ECB Tux" image:

# Encryption: Block cipher modes of operation – CBC

- "Cipherblock Chaining" feeds the output from each block into the input of the next:



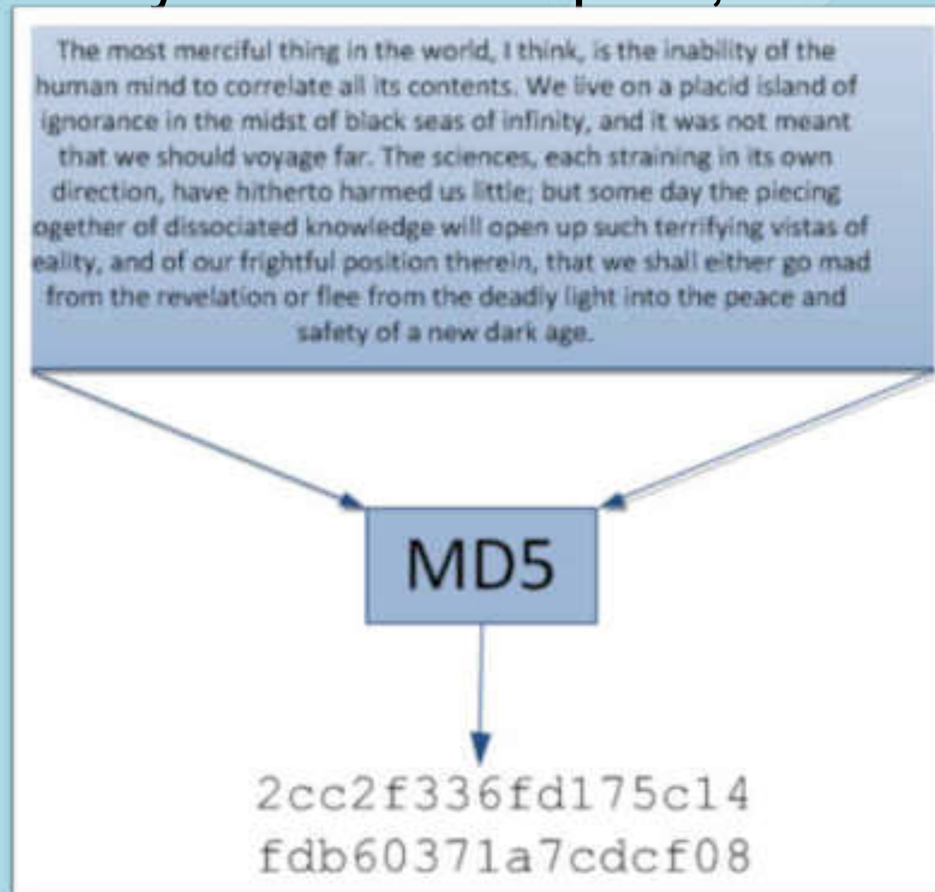

- This is much better than CBC, but also has some serious problems
- We'll talk about this in detail when we talk about padding oracles

# Initialization vectors: IVs

- The ‘input’ into an encryption function
- Designed so that the same data encrypted with the same key doesn’t generate the same ciphertext
- We’ll see why that’s a problem

# Hashing

- Reducing a large amount of data to a small amount
- Works similarly to a block cipher, as we'll see



Now, what you all came here for…

# ATTACKS

# KEY RE-USE IN BLOCK CIPHERS

# Key re-use in block ciphers

- Using the same key/IV to encrypt two messages = fail
- This affects:
  - DES (all modes)
  - 3DES (all modes)
  - AES (all modes)
  - RC2
  - RC4
  - RC5
  - And… well, everything else I've tested

# Key re-use in block ciphers: When does this work?

- This attack works if:
  - Any normal cipher is used (block or stream)
    - Note that there are better ways to attack stream ciphers
  - The attacker controls at least [blocksize] bytes of the plaintext, preferably at the beginning
    - Note that only bytes after the attacker-controlled text can be decrypted
  - The same key and IV are used each time the encryption happens
    - Note that some ciphers – like ECB – don’t have IVs, so this attack cannot be prevented

# Key re-use in block ciphers – the setup

- Here's our "oracle":

```
1  require 'openssl'
2
3  def do_crypto(prefix)
4    c = OpenSSL::Cipher::Cipher.new("DES-ECB")
5    c.encrypt
6    c.key = "MYDESKEY"
7    return c.update("#{prefix}This is some test data") + c.final
8  end
```

- Note that we're using "DES-ECB" – this attack will work, as-is, with every block and stream cipher in almost every "mode"
- ECB is somewhat special because it can't be fixed
  - We'll talk about ECB, CBC, etc. when we talk about padding oracles

# Key re-use in block ciphers: example [1]

- Here's the output from do_crypto("A" * 16):

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | Two blocks of just "A"s (note that the ciphertext is the same) |
| $C_1$ | \x74 | \x31 | \xe1 | \xf0 | \xc6 | \x1b | \x35 | \x11 | |
| $P_2$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | |
| $C_2$ | \x74 | \x31 | \xe1 | \xf0 | \xc6 | \x1b | \x35 | \x11 | |
| $P_3$ | 'T' | 'h' | 'i' | 's' | ' ' | 'i' | 's' | ' ' | The rest of the string encrypted as-is |
| $C_3$ | \x35 | \x13 | \x7b | \x27 | \xb6 | \xf5 | \xda | \x9c | |
| $P_4$ | 's' | 'o' | 'm' | 'e' | ' ' | 't' | 'e' | 's' | |
| $C_4$ | \xb1 | \x0e | \xdf | \x42 | \x93 | \xe8 | \x17 | \x42 | |
| $P_5$ | 't' | ' ' | 'd' | 'a' | 't' | 'a' | \x02 | \x02 | |
| $C_5$ | \xe0 | \x6f | \xcf | \xc0 | \xcf | \xfe | \x87 | \x66 | |

# Key re-use in block ciphers: example [2]

- Here's the output from do_crypto("A" * 7):

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | |
| $C_1$ | \xea | \xca | \x59 | \x30 | \x3d | \x8b | \xe6 | \x0f | ← GOAL |
| $P_2$ | 'h' | 'i' | 's' | ' ' | 'i' | 's' | ' ' | 's' | |
| $C_2$ | \xf2 | \xaa | \xb1 | \xfb | \x54 | \xb4 | \xb5 | \x87 | Stuff |
| $P_3$ | 'o' | 'm' | 'e' | ' ' | 't' | 'e' | 's' | 't' | That<br>We |
| $C_3$ | \x34 | \x87 | \x06 | \x80 | \x9a | \xcc | \xad | \x43 | Don't<br>Care |
| $P_4$ | ' ' | 'd' | 'a' | 't' | 'a' | \x03 | \x03 | \x03 | About |
| $C_4$ | \xd3 | \x71 | \x2a | \xf5 | \x79 | \x10 | \x25 | \xea | … |
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | …<br>… |
| $C_1$ | \xea | \xca | \x59 | \x30 | \x3d | \x8b | \xe6 | \x0f | … |

# Key re-use in block ciphers: example [3]

Goal:

| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' |
|---|---|---|---|---|---|---|---|---|
| $C_1$ | **\xea** | **\xca** | **\x59** | **\x30** | **\x3d** | **\x8b** | **\xe6** | **\x0f** |

← We're trying to find a match for this

- It's pretty trivial to guess a single byte…
  - ['A'..'Z' + 'a'..'z'] do |c| do_crypto('AAAAAAA' + c); end

Note: I'm only showing the first block or two

| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'S' |
|---|---|---|---|---|---|---|---|---|
| $C_1$ | \x1c | \x32 | \x22 | \x39 | \xb7 | \x99 | \x73 | \x42 |
| $P_2$ | 'T' | 'h' | 'i' | 's' | ' ' | 'i' | 's' | ' ' |
| $C_2$ | \x35 | \x13 | \x7b | \x27 | \xb6 | \xf5 | \xda | \x9c |

← Nope

| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' |
|---|---|---|---|---|---|---|---|---|
| $C_1$ | **\xea** | **\xca** | **\x59** | **\x30** | **\x3d** | **\x8b** | **\xe6** | **\x0f** |

← Oh hai!

| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'U' |
|---|---|---|---|---|---|---|---|---|
| $C_1$ | \x5a | \x3c | \x17 | \x25 | \xc8 | \x0f | \x68 | \x3f |

← Nope

# Key re-use in block ciphers: example [4]

- Here's the output from do_crypto("A" * 6):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | 'h' |
| $C_1$ | **\xcb** | **\x7a** | **\x74** | **\xd0** | **\x38** | **\x45** | **\xbf** | **\x21** |
| $P_2$ | 'i' | 's' | ' ' | 'i' | 's' | ' ' | 's' | 'o' |
| $C_2$ | \xf9 | \x8e | \xcd | \xdf | \x49 | \xf0 | \x86 | \xcb |
| $P_3$ | 'm' | 'e' | ' ' | 't' | 'e' | 's' | 't' | ' ' |
| $C_3$ | \x70 | \x8c | \xc0 | \x1d | \xe5 | \xf2 | \xdc | \x01 |
| $P_4$ | 'd' | 'a' | 't' | 'a' | \x04 | \x04 | \x04 | \x04 |
| $C_4$ | \xb4 | \x74 | \xfc | \x99 | \xd9 | \xbe | \xd2 | \x70 |
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | 'h' |
| $C_1$ | \xcb | \x7a | \x74 | \xd0 | \x38 | \x45 | \xbf | \x21 |

← GOAL

Stuff
That
We
Don't
Care
About
…
…
…
…

# Key re-use in block ciphers: example [5]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | 'h' |
| $C_1$ | **\xcb** | **\x7a** | **\x74** | **\xd0** | **\x38** | **\x45** | **\xbf** | **\x21** |

← We're trying to find a match for this

- Once again, we're guessing a single byte:
  - ['A'..'Z' + 'a'..'z'] do |c| do_crypto('AAAAAAT' + c); end

Note: I'm only showing the first block or two

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | 'g' | |
| $C_1$ | \xbb | \x48 | \x96 | \xa3 | \xb9 | \xb5 | \xc4 | \x32 | ← Nope |
| $P_2$ | 'T' | 'h' | 'i' | 's' | ' ' | 'i' | 's' | ' ' | |
| $C_2$ | \x35 | \x13 | \x7b | \x27 | \xb6 | \xf5 | \xda | \x9c | |
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | 'h' | |
| $C_1$ | **\xcb** | **\x7a** | **\x74** | **\xd0** | **\x38** | **\x45** | **\xbf** | **\x21** | ← Oh hai! |
| $P_1$ | 'A' | 'A' | 'A' | 'A' | 'A' | 'A' | 'T' | 'i' | |
| $C_1$ | \x79 | \xc2 | \x04 | \x11 | \x64 | \xd0 | \xae | \xc2 | ← Nope |

# Key re-use in block ciphers: What's going on?

- We continue likewise till we've decrypted the entire packet
- What's going on?
  - We're forcing the first unknown byte to be on a block boundary, then guessing it
  - We can guess any character in 256 guesses, as long as we know all the characters before it

# Key re-use in block ciphers: A tool!

- I wrote a tool called “Prephixer” to implement this attack
  - https://www.github.com/iagox86/prephixer
- Let’s do a demo!

# Preventing key re-use in block ciphers

- Use different initialization vectors (IVs) when encrypting data
- If possible, use a different key (not always possible)
- If you're using ECB mode.... WHY ARE YOU USING ECB MODE!?

# HASH LENGTH EXTENSION ATTACKS

# Hash length extension attacks

- This is why I became interested in crypto attacks
- The basic idea: most hash algorithms (before SHA3) can “pick up where they left off”
- What’s that mean for security?
  - Let’s find out!

# Hash length extension: the setup

- This is an attack where the following happens:
- Server calculates the following value:

```
verifier = H(secret || data)
```

- Then sends that verifier and data to the user
- Later, when the user sends the data back, it uses the verifier to ensure the data hasn't changed

```
if(H(secret || new_data) != verifer)
    throw error()
else
    trusted_operation(attacker_data)
```

The “Bad” operations are in red

# How hashing works...

Let's look at an example...

```
SHA1("The most merciful thing in the world, I think, is the inability of
the human mind to correlate all its contents. We live on a placid island
of ignorance in the midst of black seas of infinity, and it was not meant
that we should voyage far. The sciences, each straining in its own
direction, have hitherto harmed us little; but some day the piecing
together of dissociated knowledge will open up such terrifying vistas of
reality, and of our frightful position therein, that we shall either go
mad from the revelation or flee from the deadly light into the peace and
safety of a new dark age.")
```

First, the string is broken into 64-character blocks (for SHA1):

```
"The most merciful thing in the world, I think, is the inability "
"of the human mind to correlate all its contents. We live on a pl"
"acid island of ignorance in the midst of black seas of infinity,"
" and it was not meant that we should voyage far. The sciences, e"
"ach straining in its own direction, have hitherto harmed us litt"
"le; but some day the piecing together of dissociated knowledge w"
"ill open up such terrifying vistas of reality, and of our fright"
"ful position therein, that we shall either go mad from the revel"
"ation or flee from the deadly light into the peace and safety of"
" a new dark age."
```

# How hashing works

- Padding is added to the last block:

```
"The most merciful thing in the world, I think, is the inability "
"of the human mind to correlate all its contents. We live on a pl"
"acid island of ignorance in the midst of black seas of infinity,"
" and it was not meant that we should voyage far. The sciences, e"
"ach straining in its own direction, have hitherto harmed us litt"
"le; but some day the piecing together of dissociated knowledge w"
"ill open up such terrifying vistas of reality, and of our fright"
"ful position therein, that we shall either go mad from the revel"
"ation or flee from the deadly light into the peace and safety of"
" a new dark age.\x80\0\0\0\0\0\0\0\0\0\0\0\0..........\0\x12\x80"
```

- The padding is equal to a 1-bit followed by a bunch of zero bits ("\x80\0\0\0\0….") followed by the length in bits
- The last eight bytes are equal to the length of the string (0x250 bytes) in bits (0x1280 bits)

# How hashing works

- Each block is hashed individually, and its output is fed into the next block

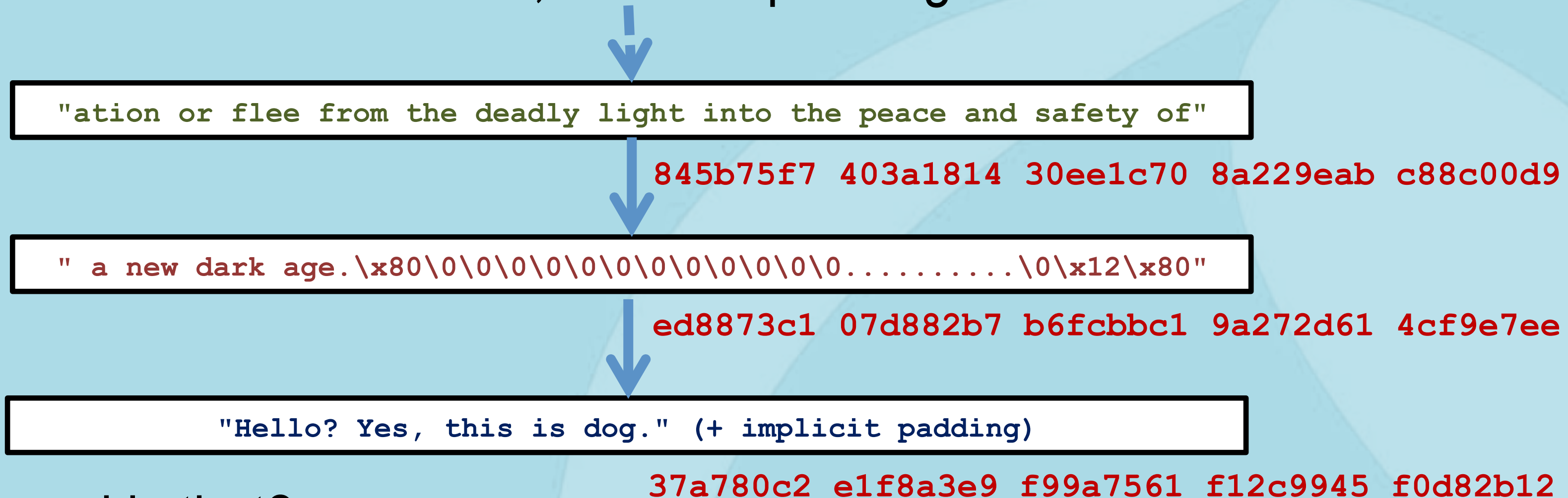
```
"The most merciful thing in the world, I think, is the inability "
    3be731c7 5880ab49 5d90c0cd df295189 b3c91449
"of the human mind to correlate all its contents. We live on a pl"
    58ec922f bdee6eec 699db098 a378f875 2b29d697
"acid island of ignorance in the midst of black seas of infinity,"
    c55bc6a3 b26d5dd0 96166093 5d73c256 7f357a83
"ation or flee from the deadly light into the peace and safety of"
    845b75f7 403a1814 30ee1c70 8a229eab c88c00d9
" a new dark age.\x80\0\0\0\0\0\0\0\0\0\0\0..........\0\x12\x80"
    ed8873c1 07d882b7 b6fcbbc1 9a272d61 4cf9e7ee
```

The output of the last block ***is our hash***

In other words, the final hash is made up of ***the entire state!***

# Hash extension: We're almost there!

- What if we add another block, after the padding?



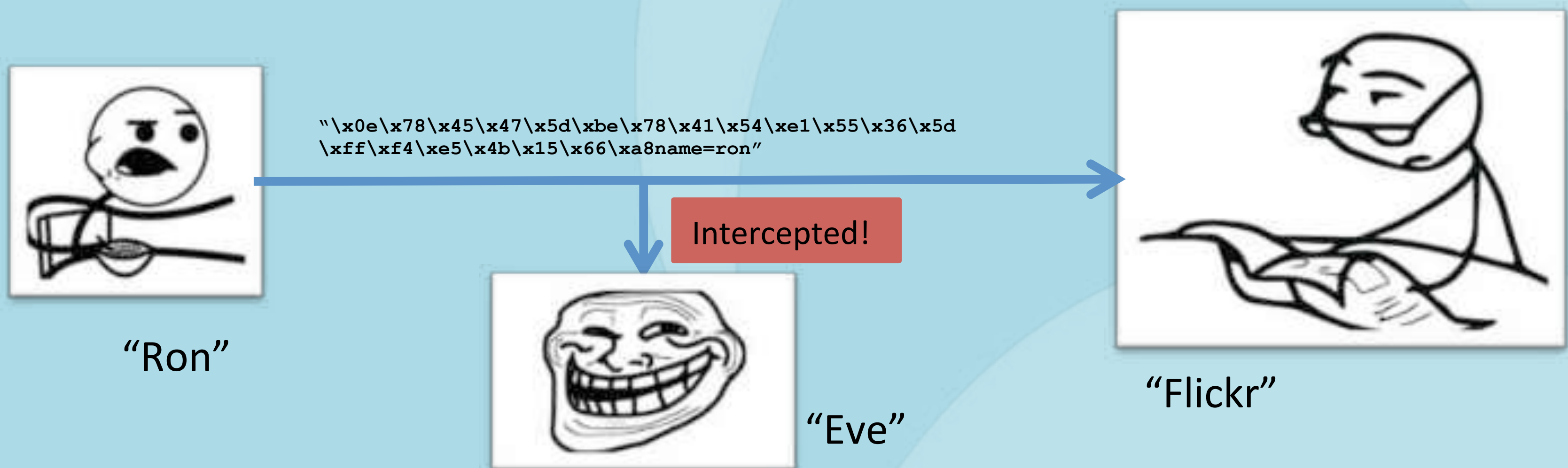

- What good is that?
- **Because 100% of the state was included in the final hash – `ed8873c1 07d882b7 b6fcbbc1 9a272d61 4cf9e7ee` – we could add more data to the plaintext and calculate a new hash *without knowing the plaintext*!**

# Hash extension: And here we are!

- Let's say that again, to make sure we're clear: we just calculated the checksum for:
  - `(original_text || padding || “Hello? Yes, this is dog.”)`
- Knowing only:
  - The output of the original hash function:
    - `ed8873c1 07d882b7 b6fcbbc1 9a272d61 4cf9e7ee`
  - And the text we wanted to add!
- We did *not* need to know original_text!

# Hash extension: Applying it

- Flickr used to have an API something like this:
  - `message = SHA1(shared_secret || commands) + commands`
- Where `shared_secret` = the user's key (aka, a password)
- Example:
  - `message = SHA1("secretkey" + "name=ron") + "name=ron"`
  - `message = "\x0e\x78\x45\x47\x5d\xbe\x78\x41\x54\xe1\x55" + "\x36\x5d\xff\xf4\xe5\x4b\x15\x66\xa8name=ron"`



# Hash extension: Applying it

- Now, Eve has a message and its associated hash. What can he use it for?
- Remember, because of padding, this is what's actually hashed:

```
"secretkeyname=ron\x80\0\0\0\0\0\0\0\0\0\0...\0\0\0\0\0\x88"
```

```
0e784547 5dbe7841 54e15536 5dfff4e5 4b1566a8
```

- But wait… as we saw before, this is the entire state of the hash! So why can't we add another block?

# Hash extension: Applying it

- Let's look at how both the evil client and the legit server calculates that hash:

"secretkeyname=ron\x80\0\0\0\0\0\0\0\0\0\0...\0\0\0\0\0\x88"

0e784547 5dbe7841 54e15536 5dfff4e5 4b1566a8

"&deletemyaccount=1" (+ implicit padding)

d281ac3c f91dbb96 0ec12c4e 0e5d73bd 91be6f10

We don't need to know the original data!

# Hash extension: evil client

```c
#include <stdio.h>
#include <openssl/sha.h>

int main(int argc, const char *argv[])
{
  SHA_CTX ctx;
  uint8_t buf[SHA_DIGEST_LENGTH];
  size_t i;

  SHA1_Init(&ctx);
  ctx.h0 = 0x0e784547;
  ctx.h1 = 0x5dbe7841;
  ctx.h2 = 0x54e15536;
  ctx.h3 = 0x5dfff4e5;
  ctx.h4 = 0x4b1566a8;
  ctx.Nl = 512;

  SHA_Update(&ctx, "&deletemyaccount=1", 18);
  SHA1_Final(buf, &ctx);

  for(i = 0; i < SHA_DIGEST_LENGTH; i++)
    printf("%02x", buf[i]);
  printf("\n");

  return 0;
}
```

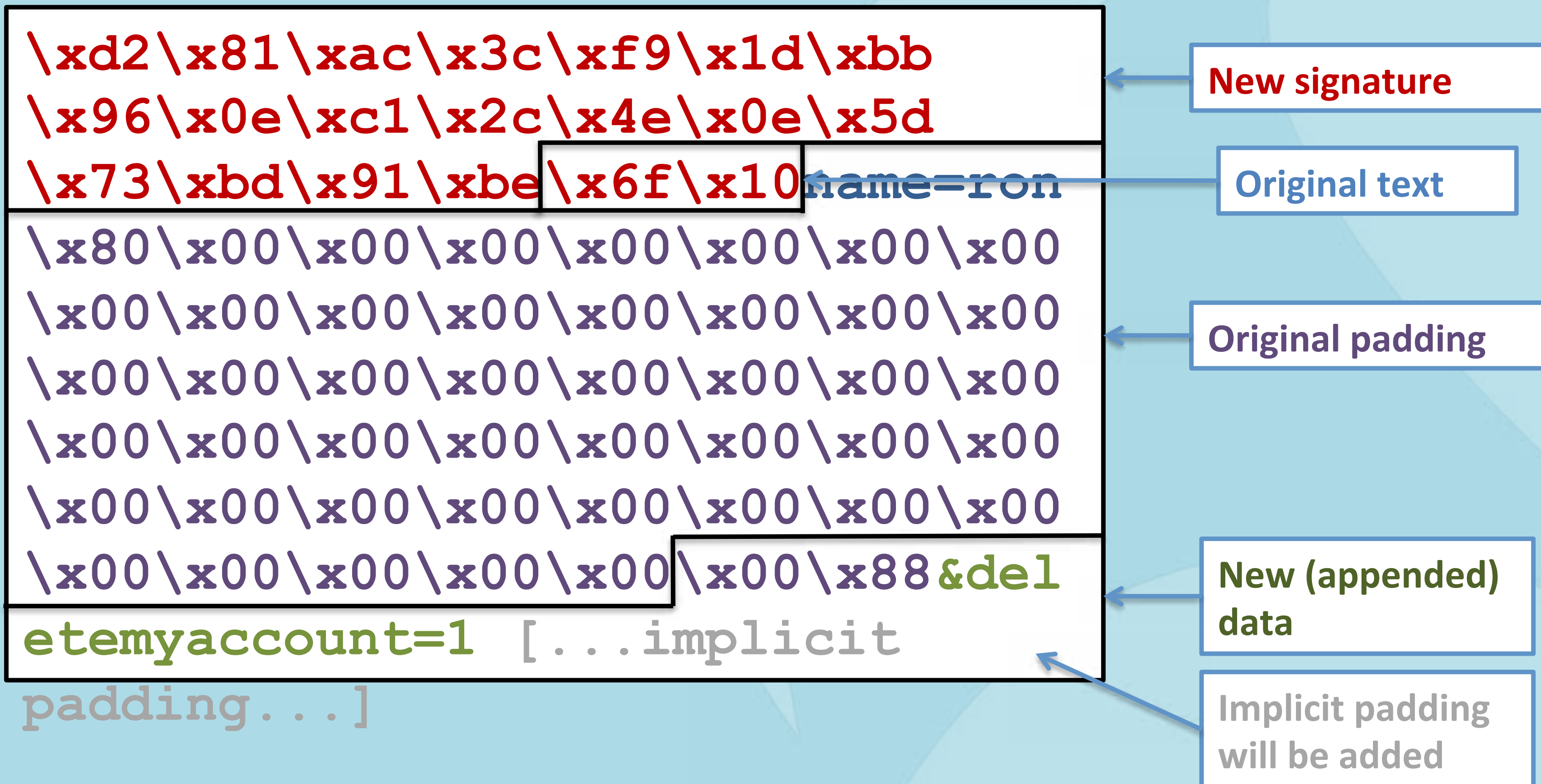
← Eve writes this program

Looks right!

```
$ gcc -o test test.c -lssl -lcrypto
$ ./test
d281ac3cf91dbb960ec12c4e0e5d73bd91be6f10
```

# Hash extension: Evil client -> legit server

- Eve then sends the following to the server:



# Hash extension: Legit server

- The legit server prepends the secret key to the data Eve sent, and calculates the hash:

```
1.9.3p194 :002 > require 'openssl'
 => true
1.9.3p194 :004 > Digest::SHA1.hexdigest("secretkey" + "name=ron
\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x88&deletemyaccount=1")
 => "d281ac3cf91dbb960ec12c4e0e5d73bd91be6f10"
```

Remember what Eve calculated!

```
$ gcc -o test test.c -lssl -lcrypto
$ ./test
d281ac3cf91dbb960ec12c4e0e5d73bd91be6f10
```

# Hash extension: A tool!

- It's amazingly difficult to write these attacks by hand
  - I never fail to mess up the number of zeroes, or forget to convert the length to bits, or screw up endianness
- Luckily, you don't have to! I wrote hash_extender to take care of that
- hash_extender supports the following hashes:
  - **MD4, MD5, RIPEMD160, SHA, SHA1, SHA256, SHA512, Whirlpool**
- The following hash types are more difficult to extend, because the state is truncated before being used:
  - **SHA224, SHA384**
- And, the following hash type is impossible to extend, by design, although time will tell:
  - **SHA3**

# That was a lot of material…

- So let’s look at some cats then do a demo

# Hash extension: summary

- If an attacker has access to a hash in the form of:
  - H(secret + “knowndata”)
- He can trivially calculate:
  - H(secret + “knowndata” + padding + anything)

# Hash extension: Defense

- HMAC.
- Next topic.

# PADDING ORACLES

# Padding oracles

- Hash extension attacks are fairly simple to understand – you just have to realize that hashes can “pick up where they left off”
- Padding oracles, on the other hand, require a bit more of a leap
- That being said, let’s do it!

# Padding oracles: Overview

- Not to be confused with the Oracle database…
- This isn't an attack against any particular algorithm, but against cipher-block chaining (CBC)
- Invented by Serge Vaudenay in the early 2000s, also called the "Vaudenay attack"
- A padding oracle attack occurs when an attacker has encrypted and unknown data that he can ask a server to secretly decrypt
  - The data is a block cipher (DES, AES, etc) in CBC mode
  - The server doesn't give indication as to what the plaintext data is
  - The returns a boolean value indicating whether the decryption succeeded (which is based on the padding)

# Padding oracles: Padding

- We already talked about padding on hashes, but this is different
- Block ciphers require the data to be padded such that it's a multiple of the blocksize
  - If the data is already a multiple, an empty block is added
- It doesn't matter what the padding is, just that it's known and unambiguous
- Let's look at the most common…

# Padding oracles: Padding

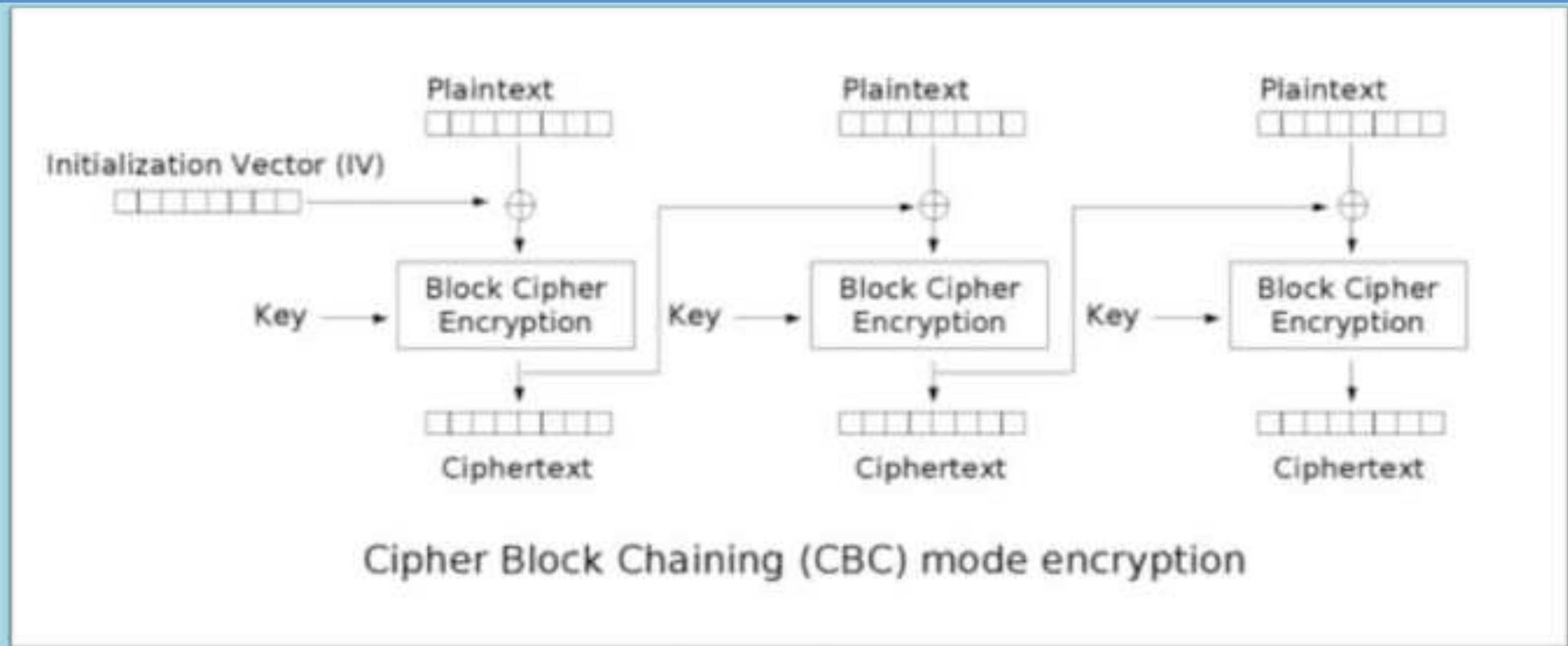
- Typically, PKCS #7 is used, which says…
  - The value of the padding = the number of bytes of padding
- Eg (assume block size = 8 [DES, for example]):

| Block 1 | | | | | | | | Block 2 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | e | l | l | o | \x03 | \x03 | \x03 | | | | | | | | |
| H | e | l | l | o | | W | o | r | l | d | \x05 | \x05 | \x05 | \x05 | \x05 |
| P | a | s | s | w | o | r | d | \x08 | \x08 | \x08 | \x08 | \x08 | \x08 | \x08 | \x08 |

# Padding oracles: CBC mode encryption

- Now that we’ve looked at padding, let’s look at how the blocks fit together
- We already talked about electronic codebook (ECB) and cipher-block chaining (CBC)
- The “padding oracle attack” is actually an attack against CBC
- Let’s see why…

# Padding oracles: CBC mode encryption



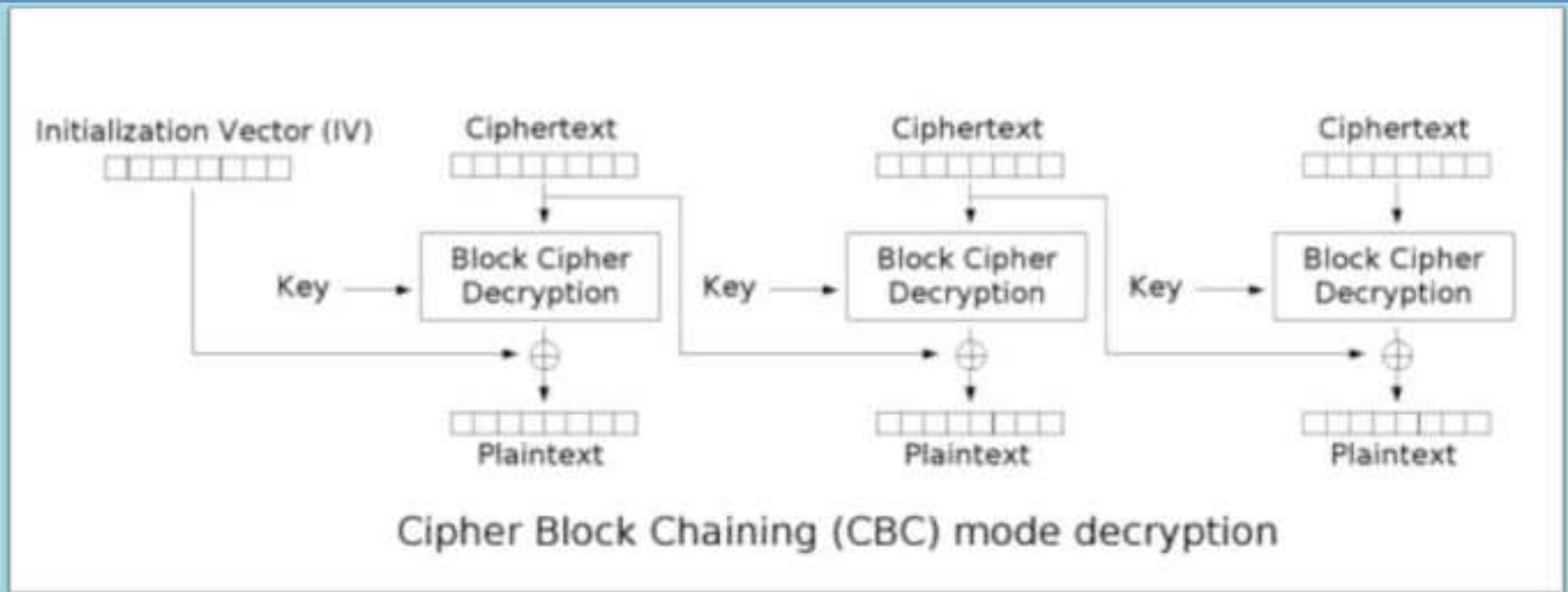
Cipher Block Chaining (CBC) mode encryption

- For any given block of plaintext, $P_n$, the corresponding ciphertext, $C_n$, can be calculated as:

$$C_n = E(P_n \oplus C_{n-1})$$

- In other words, you XOR the plaintext with the previous ciphertext, then encrypt it

# Padding oracles: CBC mode decryption




- For any given block of ciphertext, $C_n$, we can calculate the corresponding plaintext, $P_n$, as:

$$P_n = D(C_n) \oplus C_{n-1}$$

- In other words, the ciphertext is decrypted, then XORed with the previous ciphertext

# Padding oracles

- So, we have two formulas:

$$C_n = E(P_n \oplus C_{n-1})$$
$$P_n = D(C_n) \oplus C_{n-1}$$

- We can verify these make sense by encrypting and decrypting a block:

$$P_n = D(E(P_n \oplus C_{n-1})) \oplus C_{n-1}$$
$$P_n = P_n \oplus C_{n-1} \oplus C_{n-1}$$
$$P_n = P_n$$

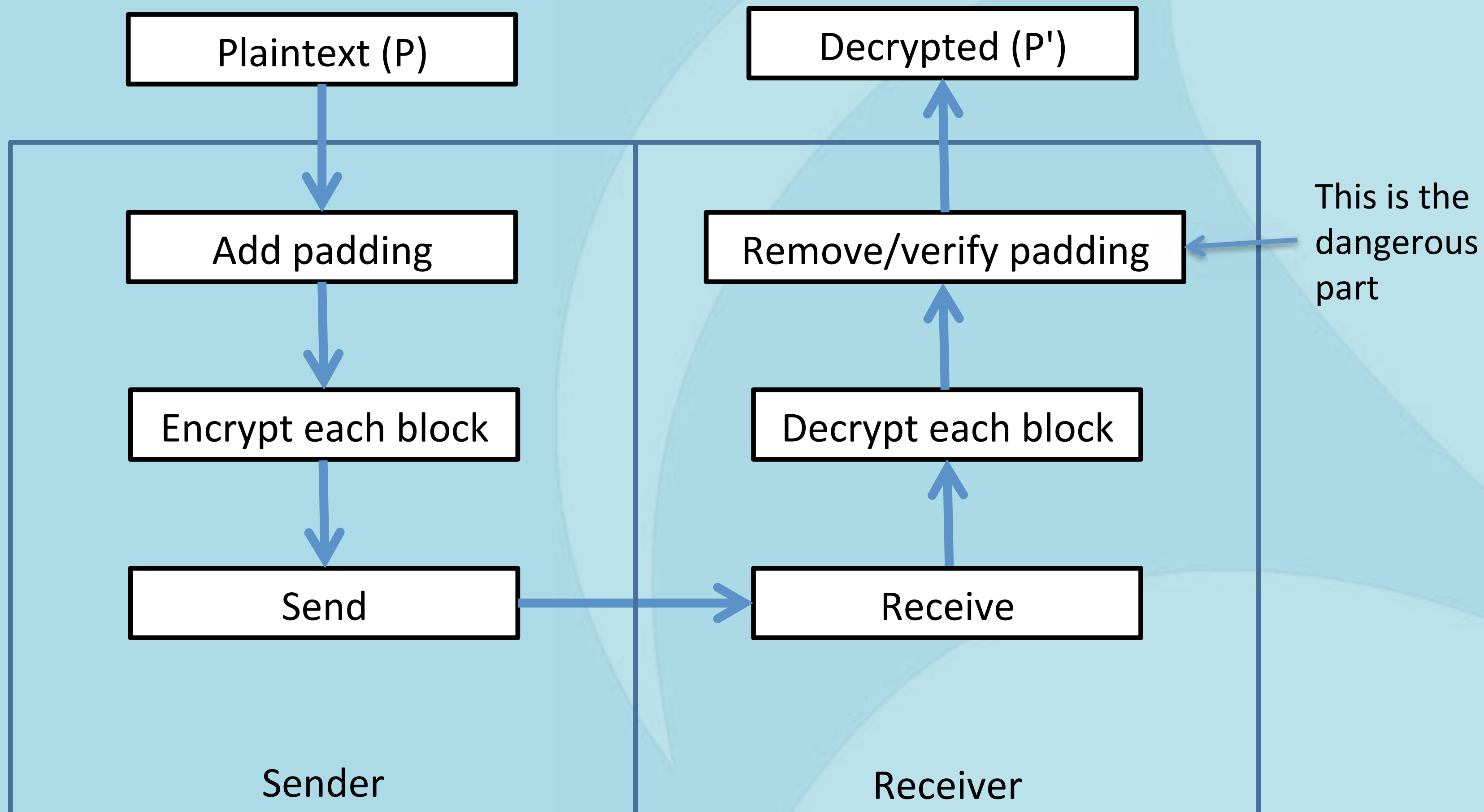
Decrypt the encrypted data

Two XORs cancel out

Success!

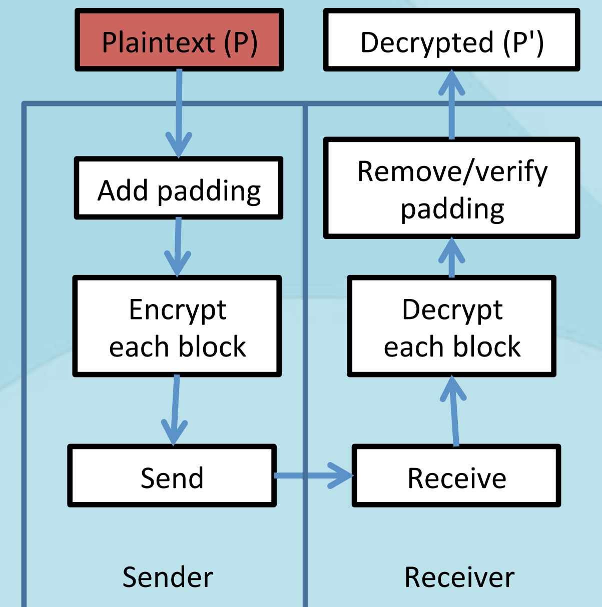
# Padding oracles

- Encryption steps…



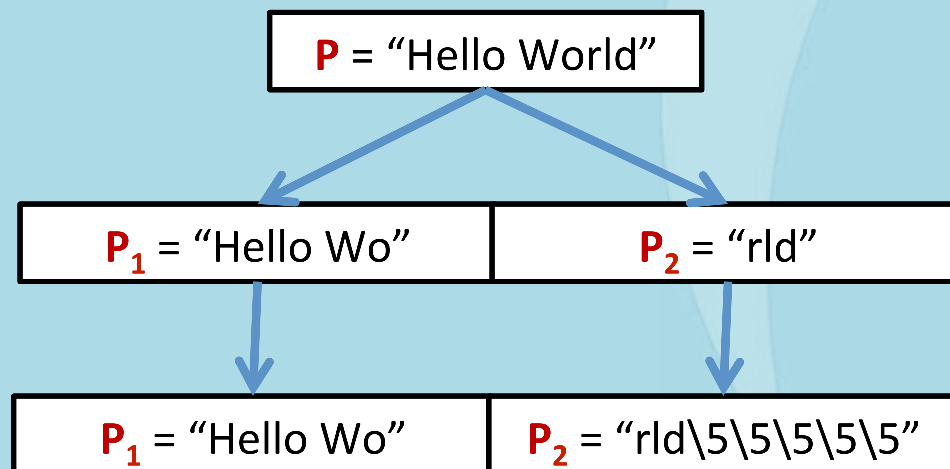
# Padding oracles

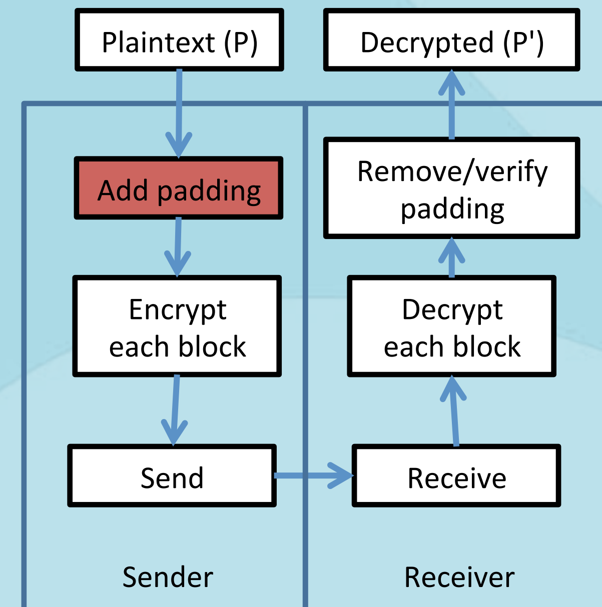
- Let's start with how this is "supposed" to work
  - Example string: "Hello World"
  - **P** = "Hello World"



# Padding oracles

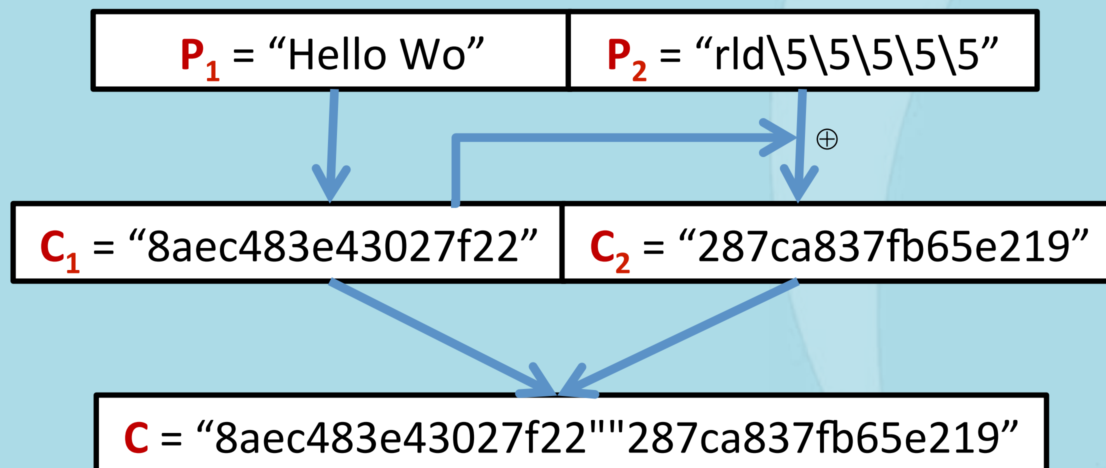
- Adding padding
  - "Hello World" is 11 characters
  - With a blocksize of 8, that means we have one full block ("Hello Wo"), and one block of 3 characters ("rld")
  - Therefore, we need 5 bytes of padding

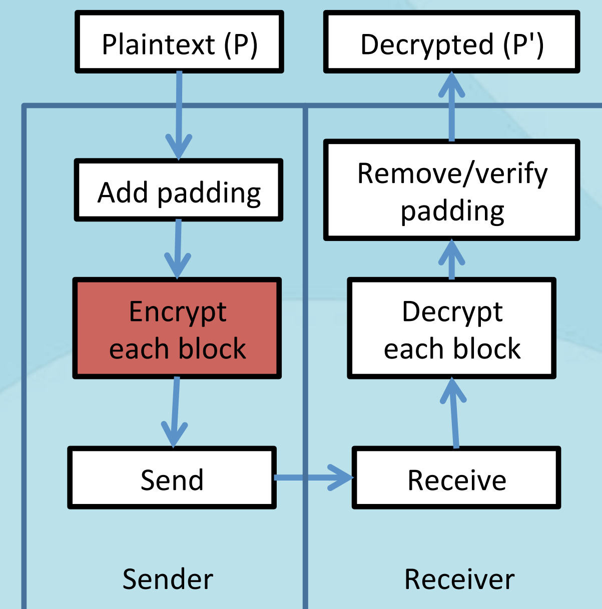


# Padding oracles

- $P_1$ is encrypted to become $C_1$
- $P_2$ is encrypted, then XORed with $C_1$, to become $C_2$.
- $C_1$ and $C_2$ are combined to make C.

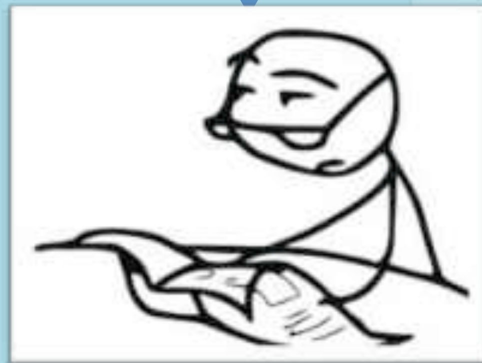


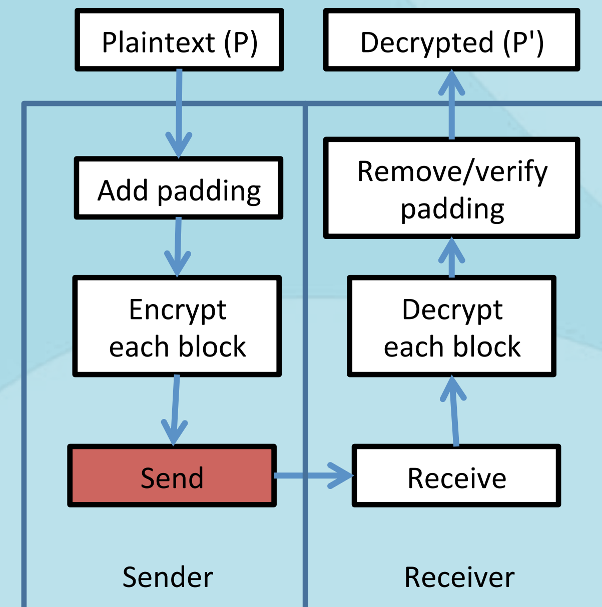



# Padding oracles

- The ciphertext – **C** – is transmitted to the (possibly malicious) user
- It's important to remember that while the user can store it and send it back to the server, the user ***cannot* decrypt it**

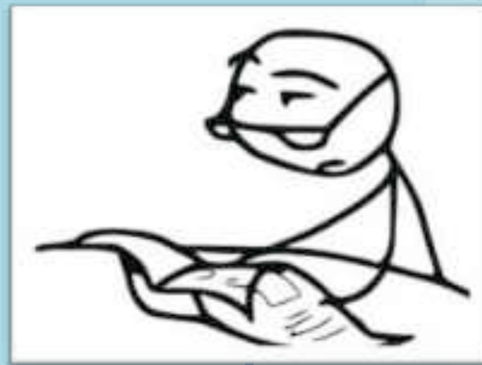
**C** = “8aec483e43027f22""287ca837fb65e219”



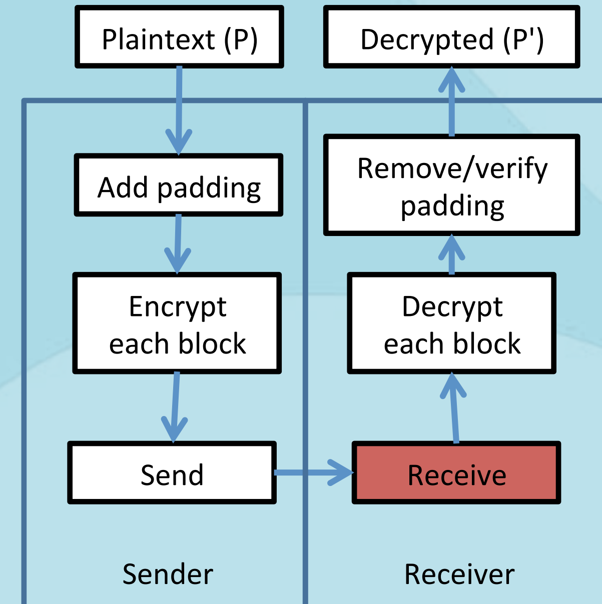


# Padding oracles

- At some point in the future, the user will return the encrypted data to the server
- This is where the attack and normal usage diverge, as we'll see

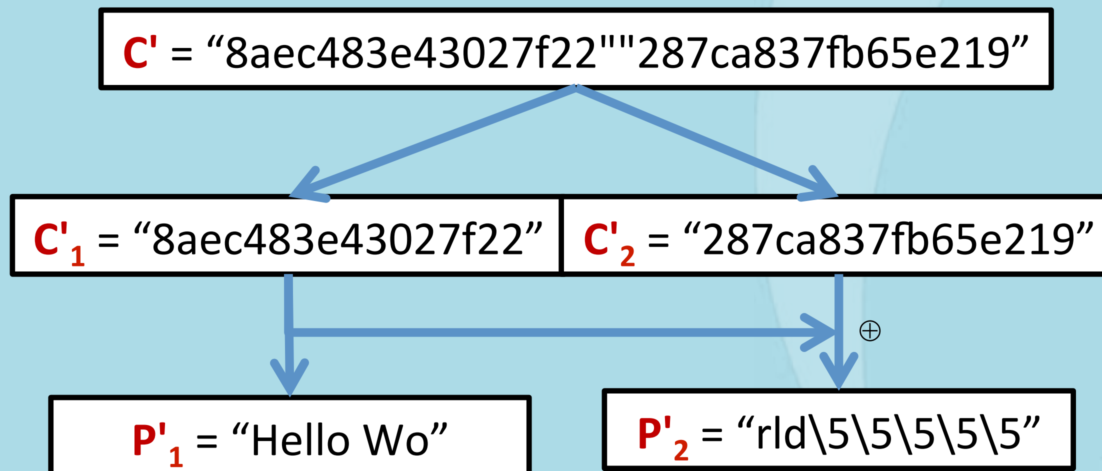


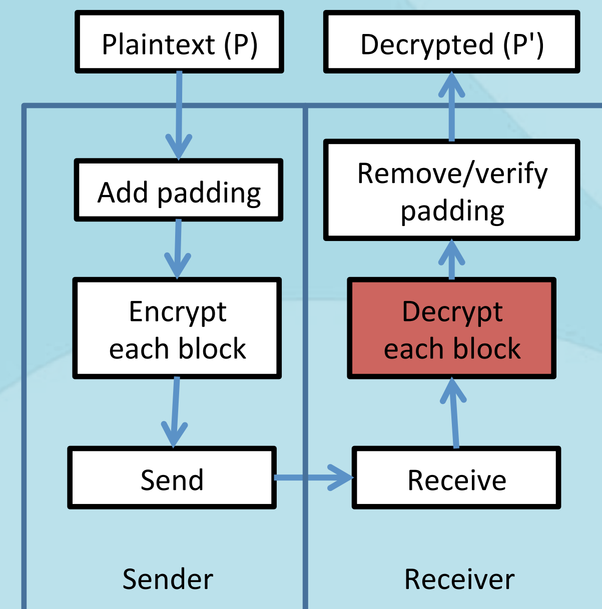
C' = “8aec483e43027f22""287ca837fb65e219”



# Padding oracles

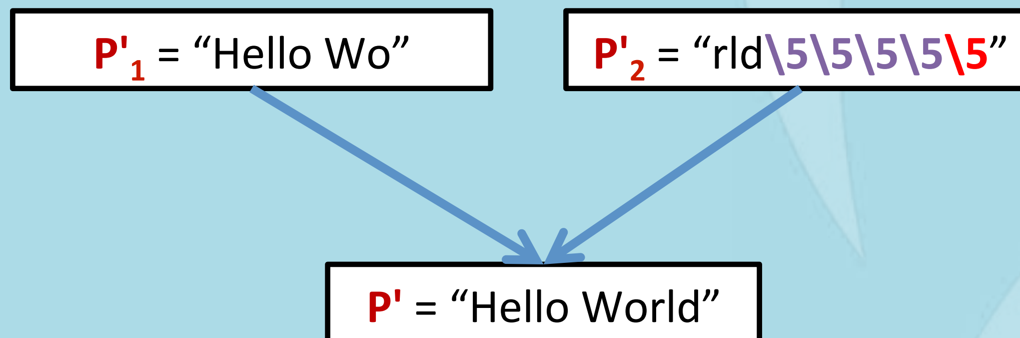
- The server breaks **C'** back into **$C'_1$** and **$C'_2$**
- **$C'_2$** is decrypted then XORed with **$C'_1$**
  - Note: the server, at this point, *doesn't know if valid data was produced*

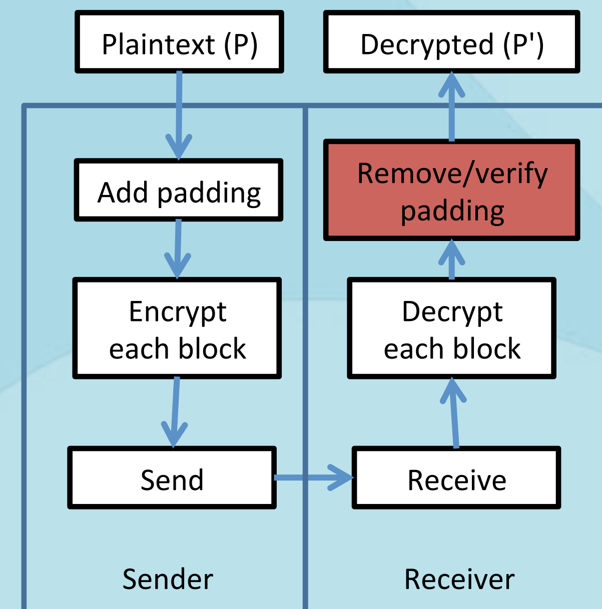


# Padding oracles

- The server looks at the last byte of the last block – "\5" – and verifies that the last 5 bytes are all equal to "\5" – otherwise, the padding is wrong
- Once padding is verified, it's removed and the blocks are reunited
- This is *all that's done* to verify that the data decrypted properly

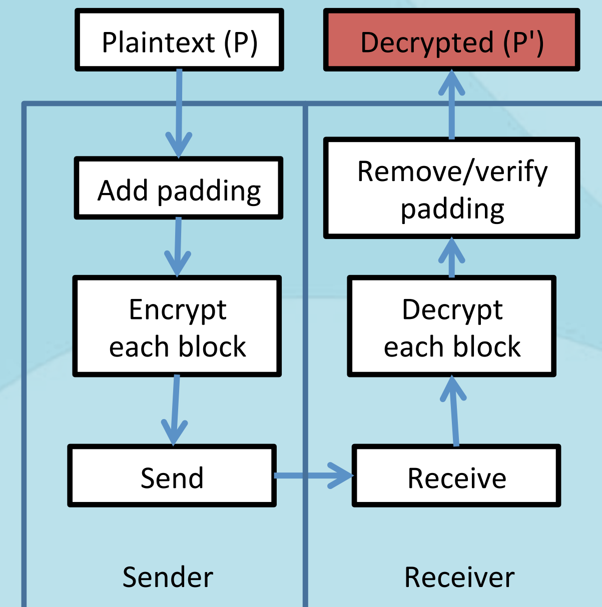


# Padding oracles

- And now, we're back where we started
  - P' = P, as it's supposed to

**P' = “Hello World”**

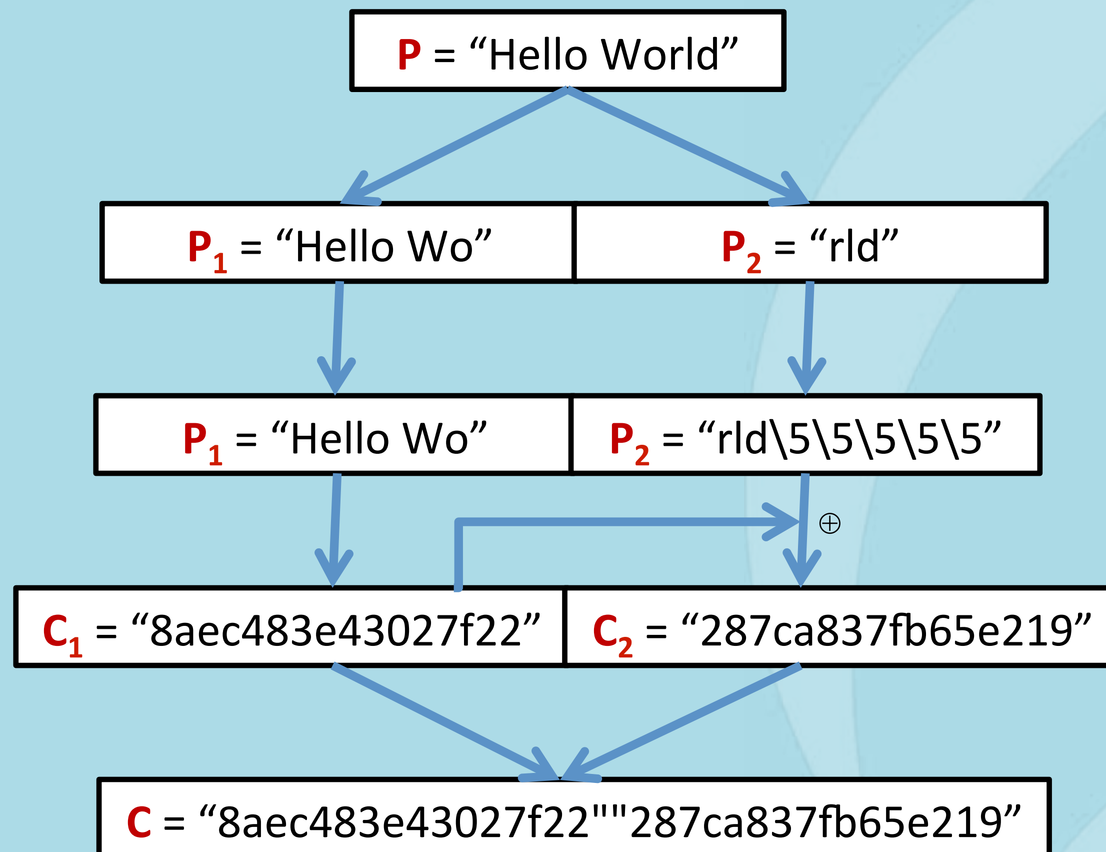


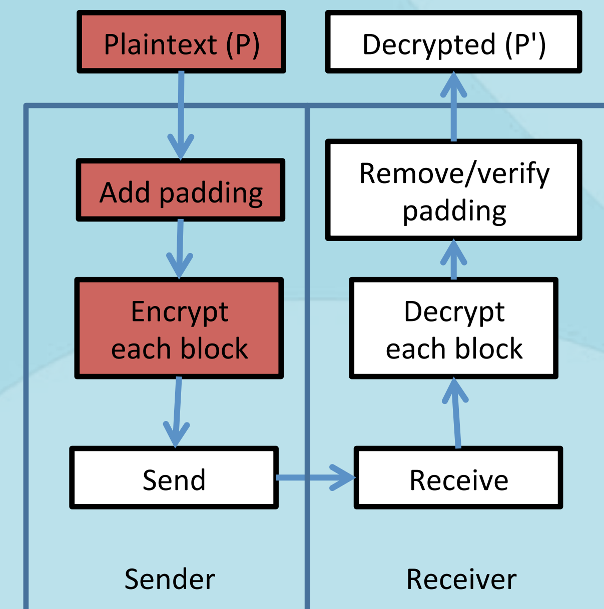
# Padding oracles

- Now let's look at an example of how we can attack this…

# Padding oracles

- Here are the first three steps again, same as last time

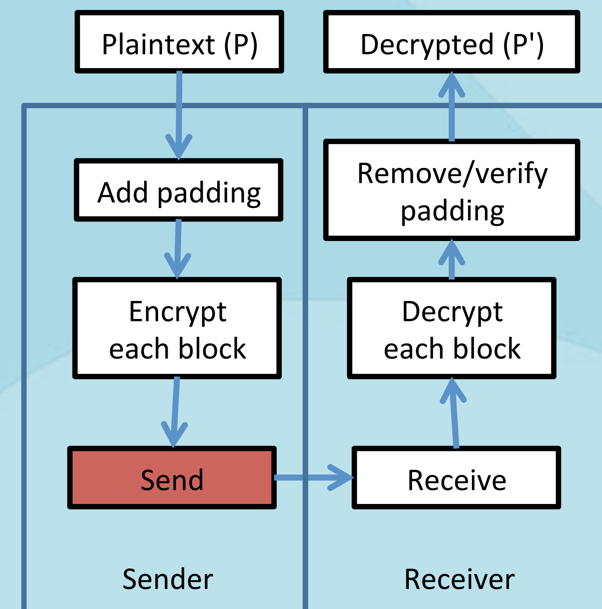


# Padding oracles

- This time, the data is sent to a malicious user, such as Eve
- As before, Eve cannot decrypt this. But she *wants* to, and will find a way!

**C** = “8aec483e43027f22""287ca837fb65e219”



# Padding oracles

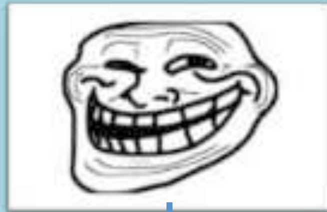
- Eve is going to focus only on the second half of the cipher, $C_2$ (note that this can apply to any block)

$C'_2$ = “287ca837fb65e219”

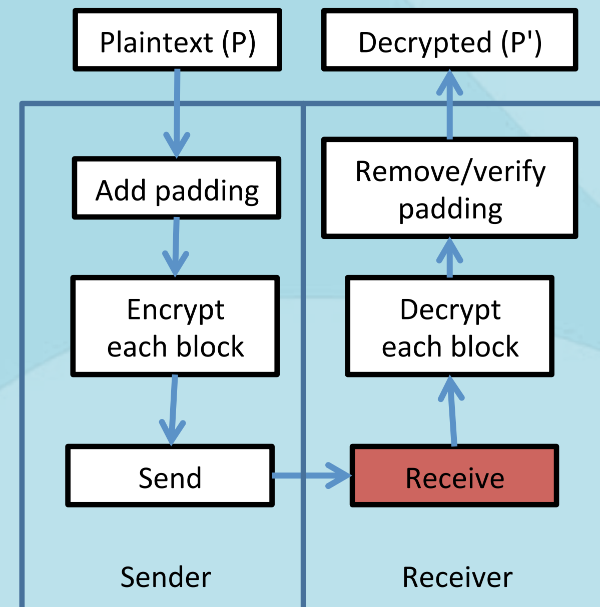
- She generates a brand new block for $C'_1$, then prepends it to $C_2$ to form $C'_2$

| $C'_1$ = “0000000000000000” | $C'_2$ = “287ca837fb65e219” |
|---|---|

Used all zeroes for convenience – not necessary

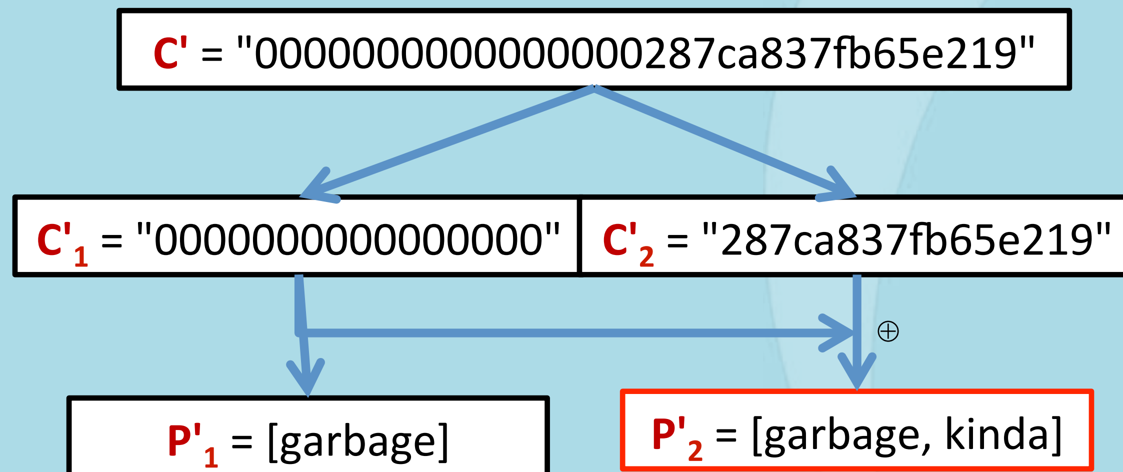


$C'$ = “0000000000000000""287ca837fb65e219”

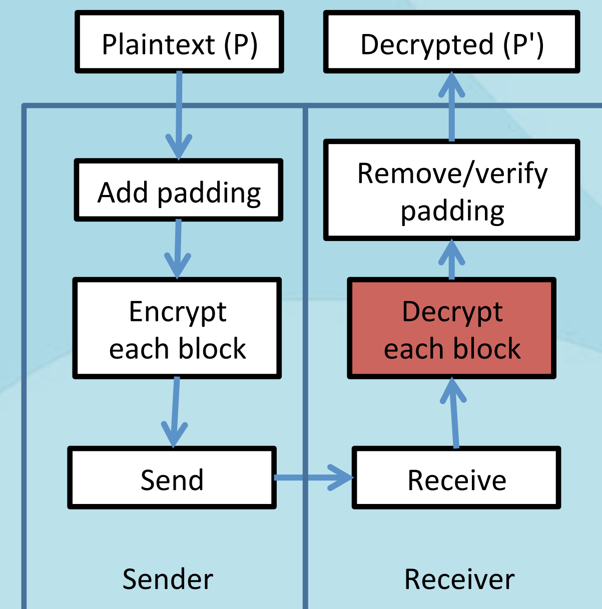


# Padding oracles

- Now, what happens when the server tries to decrypt this new **C'**?
- Remember, $C'_2$ is XORed with $C'_1$ after it's decrypted




Let's take a closer look at $P'_2$...



# Padding oracles

- Recall our decryption formula:

$$P_n = D(C_n) \oplus C_{n-1}$$

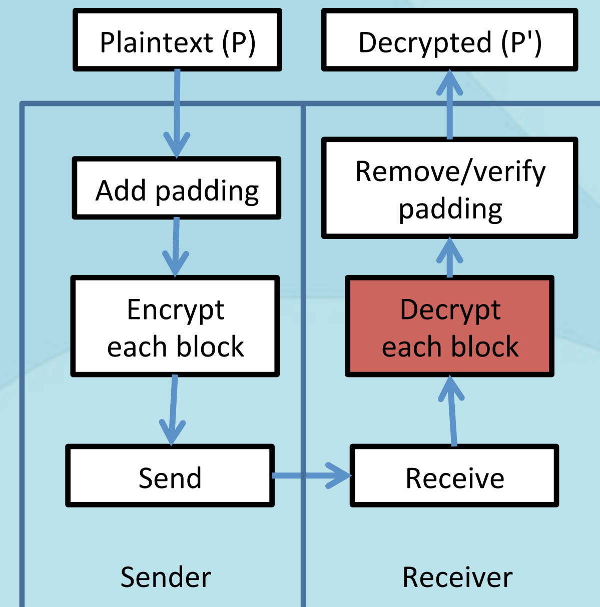
$$P_2 = D(C'_2) \oplus C'_1$$

$C'_1$ = "0000000000000000 "    $C'_2$ = "287ca837fb65e219"

- So $C'_2$ is being decrypted in the usual way, *then XORed with* $C'_1$ instead of with the actual $C_1$*!*
- Instead of undoing the original XOR, it's adding another XOR layer:

$$P'_2 = P_2 \oplus C_1 \oplus C'_1$$

$$P'_2 \; != P_2$$



# Padding oracles

- To put it another way…
- Original decryption:

$$P_n = D(E(P_n \oplus C_{n-1})) \oplus C_{n-1}$$
$$P_n = P_n \oplus C_{n-1} \oplus C_{n-1}$$
$$P_n = P_n$$

- New decryption:

$$P_n = D(E(P_n \oplus C_{n-1})) \oplus C'_{n-1}$$
$$P_n = P_n \oplus C_{n-1} \oplus C'_{n-1}$$

…can't reduce any further

# Padding oracles

- So now we have this formula:

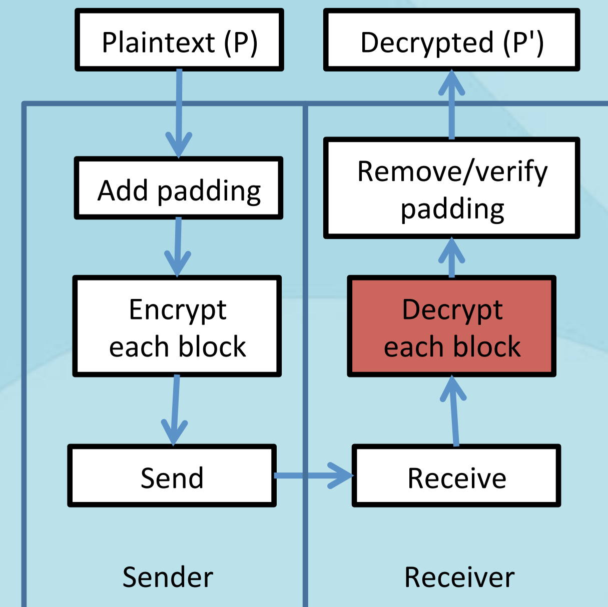
$$P'_2 = P_2 \oplus C_1 \oplus C'_1$$

- But what's it *mean*?
- The first thing we want to do is re-arrange it, because we want to solve for the original plaintext, $P_2$ :

$$P_2 = P'_2 \oplus C_1 \oplus C'_1$$

- This is legal, because

$$A \oplus B = B \oplus A$$

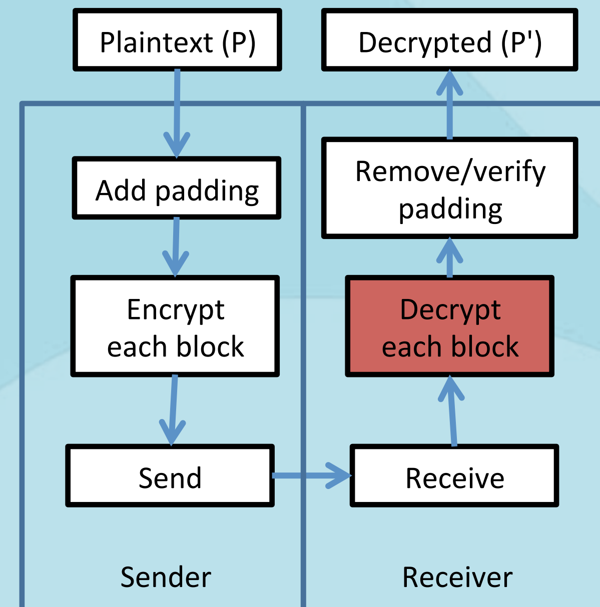


# Padding oracles

$P_2 = P'_2 \oplus C_1 \oplus C'_1$

Here's that formula again. And here's what the terms mean:
$P_2$ = The original plaintext value (our goal)
$P'_2$ = The value the server calculates (mostly a garbage string)
$C_1$ = The previous ciphertext block (known to us)
$C'_1$ = The ciphertext block chosen by the attacker (“0000000000000000”)

- We ***control*** $C'_1$, and we ***know*** $C_1$, but what about $P_2$ and $P'_2$?
- We have an equation with two unknowns!
  - Or do we?

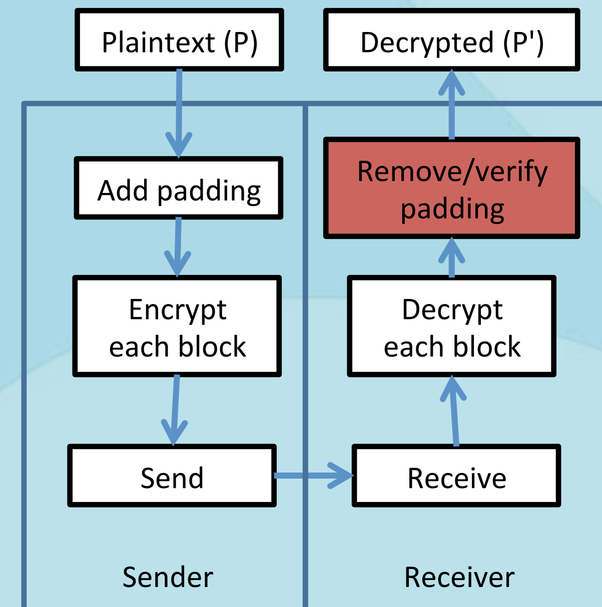


# Padding oracles

$$P'_2 = P_2 \oplus C_1 \oplus C'_1$$

- Let's focus on $P'_2$, and move on to the next step: Remove/verify padding
- If $P'_2$ decrypts to:
  - "[garbage]\x00" ← Bad padding
  - "[garbage]**\x01**" ← Good padding
  - "[garbage]\x02" ← Bad padding (probably*)
  - "[garbage]\x03" ← Bad padding
  - "[garbage]\x04" ← Bad padding
  - "[garbage]\x05" ← Bad padding
  - ……
- By definition – for this attack – the server tells us when the padding is good or bad

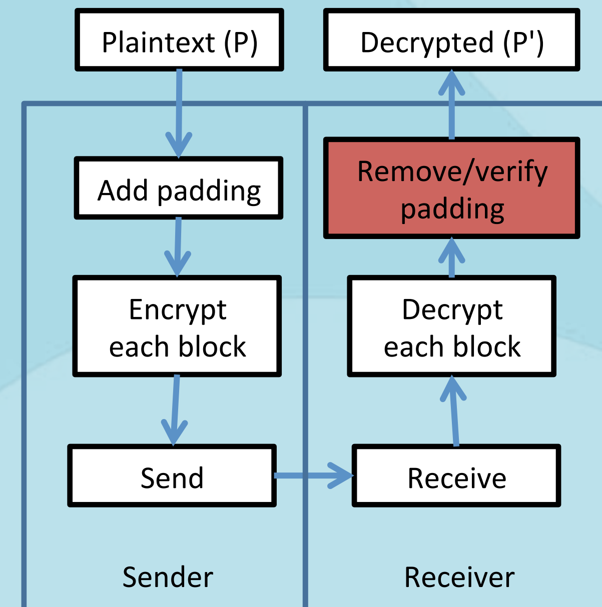
Let's ignore the rare case where the string happens to end with \x02\x02 or \x03\x03\x03 or …



# Padding oracles

$$P'_2 = P_2 \oplus C_1 \oplus C'_1$$

- What's this mean for our attack?
- As soon as the padding is correct, we know the last byte of the new plaintext (**$P'_2$**)
- That's big. That's HUGE!
  - Suddenly, our equation for the last byte of **$P_2$** only has one unknown!
  - All we have to do is send 256 values for the last byte of **$C'_1$**, and we're eventually going to get valid padding



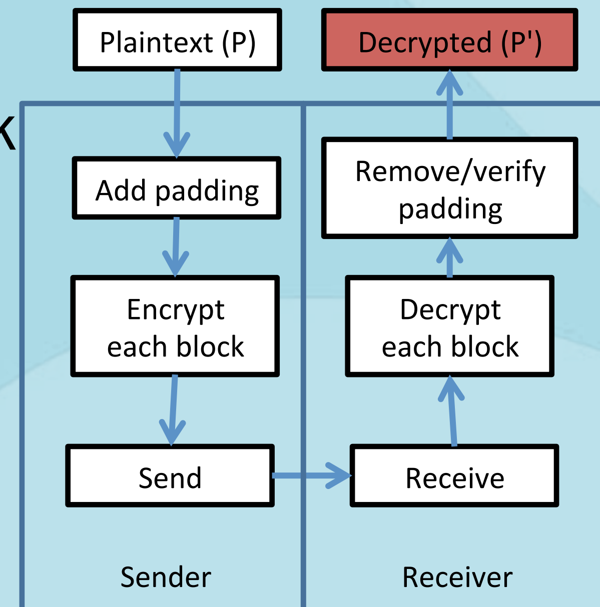
# Padding oracles

- Let’s only focus on the last bytes:

$$P_2[n] = P'_2[n] \oplus C_1[n] \oplus C'_1[n]$$

- To define everything again…
  - $P_2[n]$ = The last byte of plaintext (the value we want!)
  - $P'_2[n]$ = The last byte of what the server ends up decrypting (proper padding = "\x01")
  - $C_1[n]$ = The last byte of the original first block
  - $C'_1[n]$ = The last byte of the new first block
- **We can now calculate $P_2[n]$!**

$$P_2[n] = 1 \oplus C_1[n] \oplus C'_1[n]$$



# Padding oracles

- So, to summarize:
  - Choose a new block, which we call **C'**, and prepend it to the block you're trying to decrypt:

| **C'** = "0000000000000000" | $C_2$ = "287ca837fb65e219" |
|---|---|

  - Change the last byte of **C'** until you stop getting a padding error:

| **C'** = "0000000000000026" | $C_2$ = "287ca837fb65e219" |
|---|---|

  - Plug it into the formula:

$$P_2[n] = P'_2[n] \oplus C_1[n] \oplus C'_1\,[n]$$

$$P_2[n] = 0x01 \oplus 0x22 \oplus 0x26$$

  - And solve!

$$P_2[N] = 0x05$$

*Recall:*
*$P_2$ = "rld\5\5\5\5\5"*

# Padding oracles

- By having the server tell us when the last byte of the decrypt block is right, we can trivially decrypt and encrypt it using only the XOR operation
- The last byte can be set to \x02, and the second-last byte can be guessed using the same formula
- The last and second-last bytes can be set to \x03\x03, and the third-last byte can be guessed using the same formula
- …and so on, until the whole block is decrypted

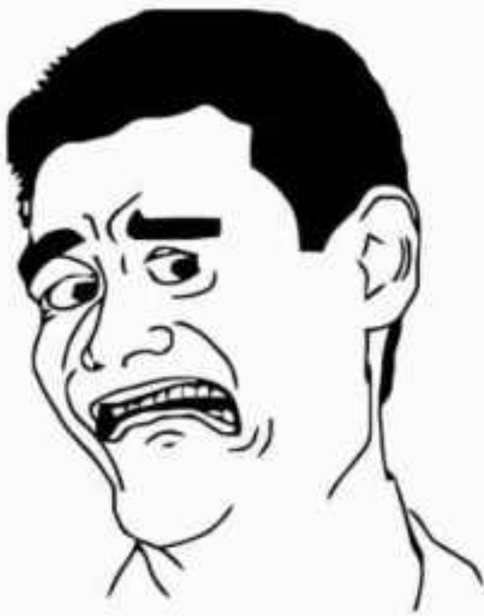
# Introducing: Poracle

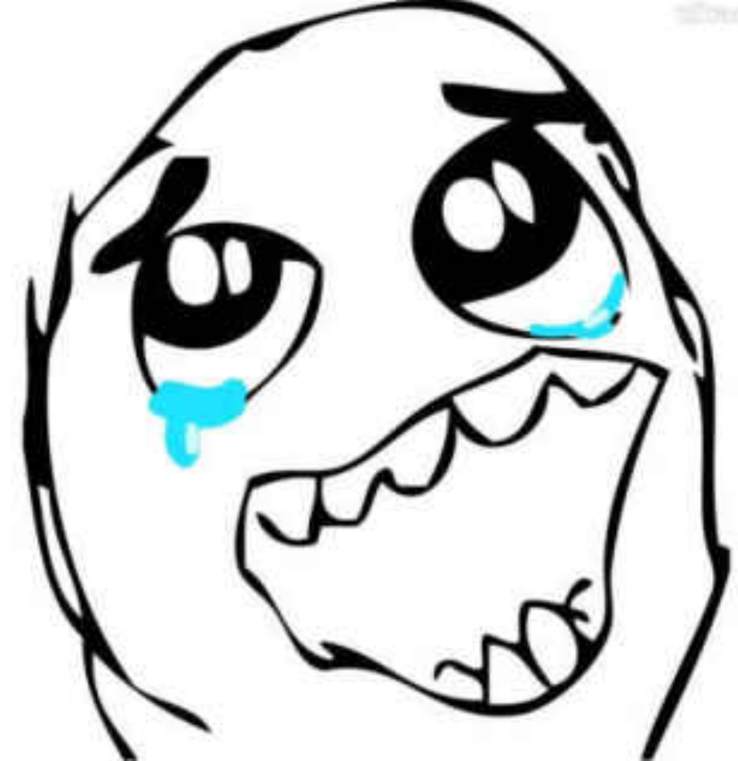
- Like all these attacks, I wrote a tool
- This one's called "Poracle"

# Padding oracles: Prevention

- How do you prevent padding oracles?
  - HMAC!
- By prepending an HMAC hash to the encrypted data – *and validating it before the decryption is performed* – you can check if anybody has tampered with the hash!
- You can also prevent this by using a block cipher mode of operation other than cipher-block chaining – eg, counter mode, output feedback, plaintext feedback, etc.

# Almost there!





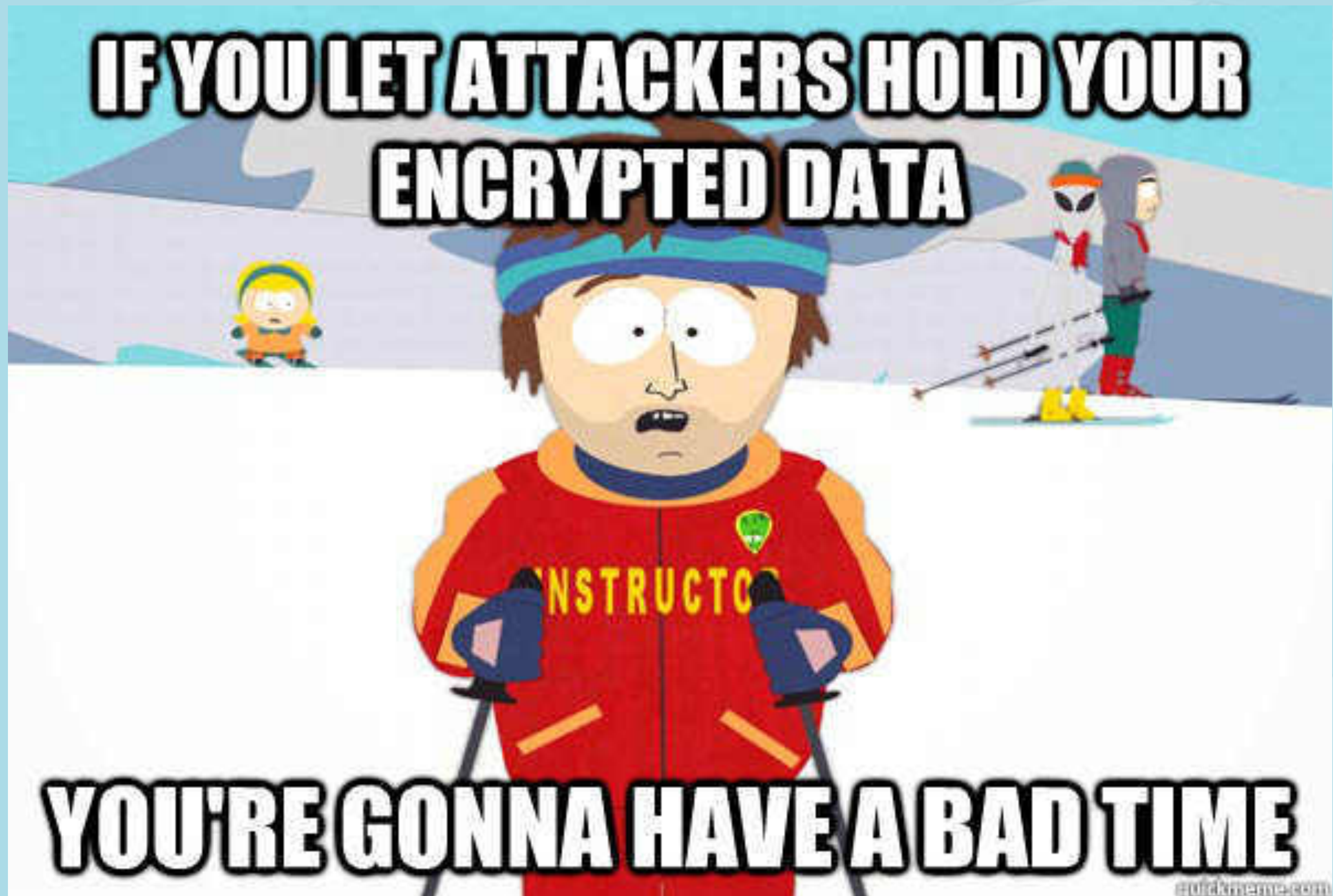
Because people get mad at me for just pointing out problems…

# SOLUTIONS

# Solution #1: don’t give attackers encrypted data

- This isn’t always possible
- When you can, give an index, a session, or something like that, rather than letting an attacker store state

# Or, to put it another way…



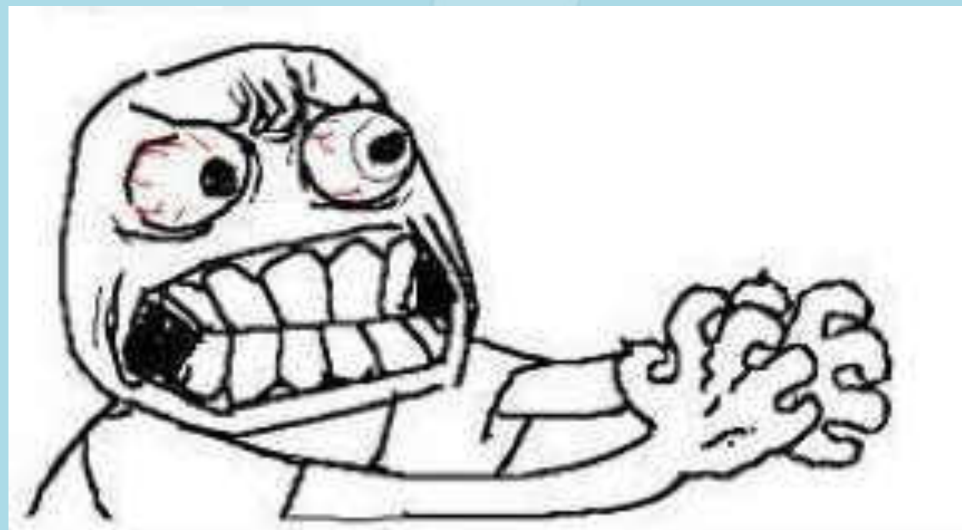
# Solution #2: When you give them encrypted data, validate it

- “The cryptographic doom principle”
- Calculate a HMAC and send it with the encrypted data
  - Validate the HMAC before attempting to decrypt
- Alternatively, use authenticated encryption, for example, "GCM Mode"
- Coming soon: CAESAR
  - **CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness**

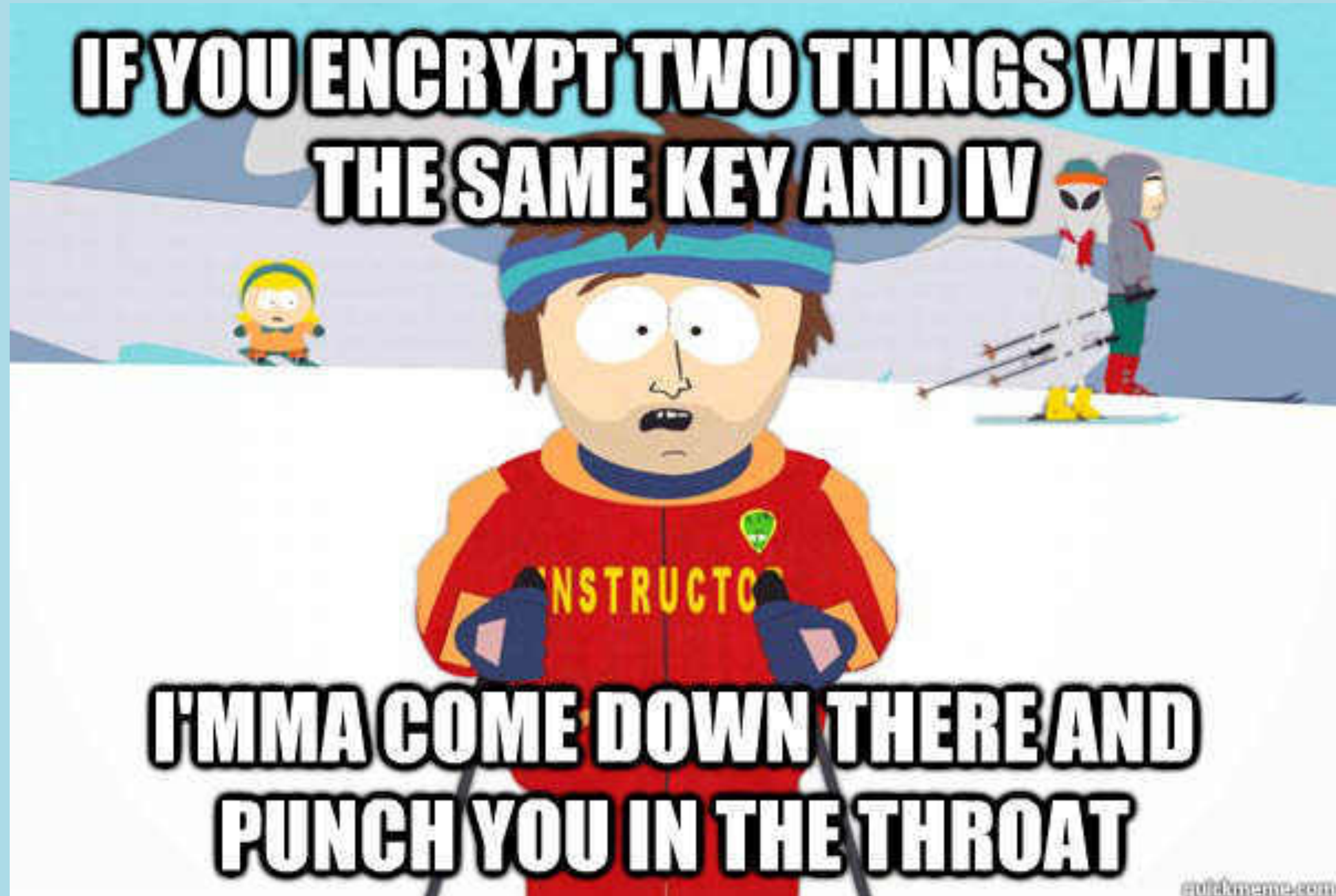
# “The cryptographic doom principle”

# Solution #3: Never encrypt data with the same key and IV

- Almost every cipher fails if you use the same key and IV
- Change keys when it makes sense, and change IVs *every time*



# One last ski instructor, then we're done!



# THAT’S ALL!

# Links + Contact info

- Me:
  - Ron Bowes <ron.bowes@leviathansecurity.com>
  - @iagox86
  - http://www.skullsecurity.org
  - http://www.leviathansecurity.com
- Tools:
  - https://www.github.com/iagox86/prephixer
  - https://www.github.com/iagox86/poracle
  - https://www.github.com/iagox86/hash_extender
  - https://www.github.com/iagox86/unzipher
- This talk will be on https://www.github.com/iagox86 as well