

# Crypto: You're doing it wrong

Ron Bowes, Leviathan Security Group

Shmoocon 2013

# About me

- Ron Bowes
  - @iagox86
  - <http://www.skullsecurity.org>
  - [ron.bowes@leviathansecurity.com](mailto: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



# 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!

# Comparing “modern” crypto methods

## Relative strength when used properly

	Completely broken	Somewhat broken	Probably good
DES (1979)		X	
RC4 (1987)		X	
3DES (1998)			X
AES (1998)			X
Blowfish			X
MD4 (1990)		X	
MD5 (1992)		X	
SHA1 (1995)		X	
SHA2 (2001)			X
SHA3 (2012)			X

## Relative strength when used incorrectly

	Completely broken	Somewhat broken	Probably good
DES (1979)	X		
RC4 (1987)	X		
3DES (1998)	X		
AES (1998)	X		
Blowfish	X		
MD4 (1990)	X		
MD5 (1992)	X		
SHA1 (1995)	X		
SHA2 (2001)	X		
SHA3 (2012)		???	

# For our purposes...

- “Crypto”, in the context of this talk, will cover:
  - Encryption
  - Hashing
  - Random numbers
  - Anything else I need it to cover



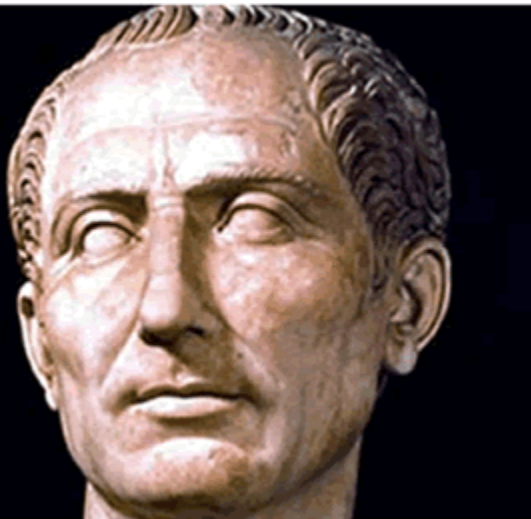


The somewhat accurate

# 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.

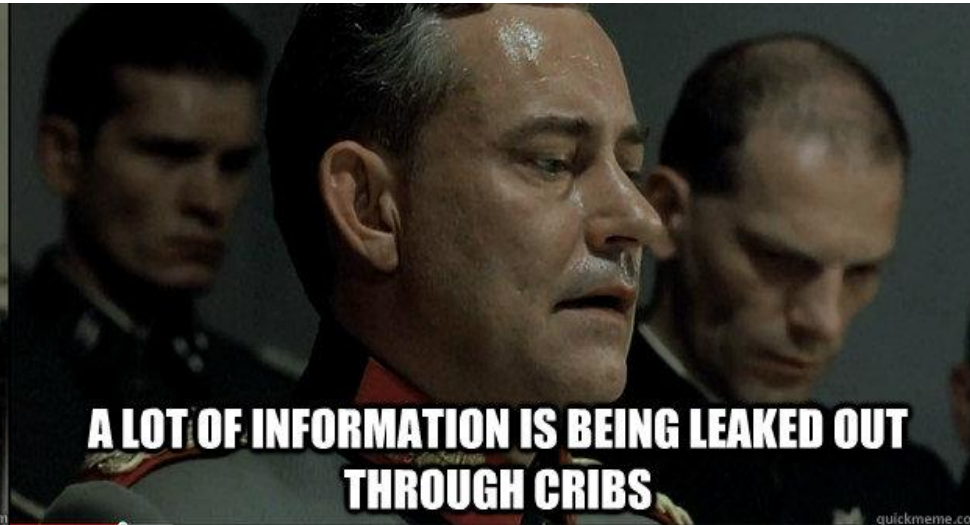
The End.

You're welcome.

[\[Return\]](#) [\[Catalog\]](#) [\[Top\]](#) [\[Update\]](#) [ ☐ Auto ]

- Not shown: the part where they call OP a “fag” (<3 4chan)

# World War II: Enigma Machine

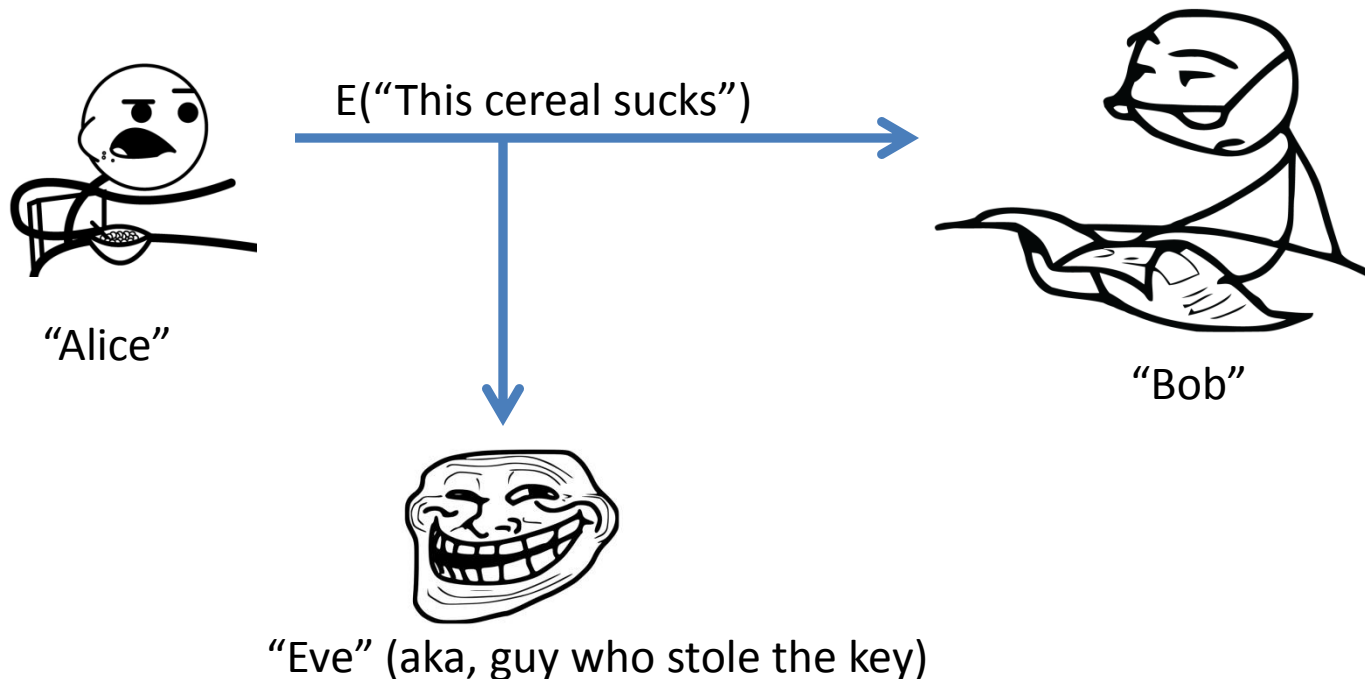


# Let's get more modern



# 1970s: DES was invented!

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



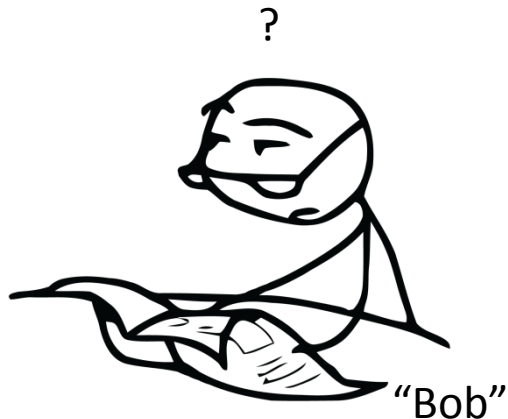
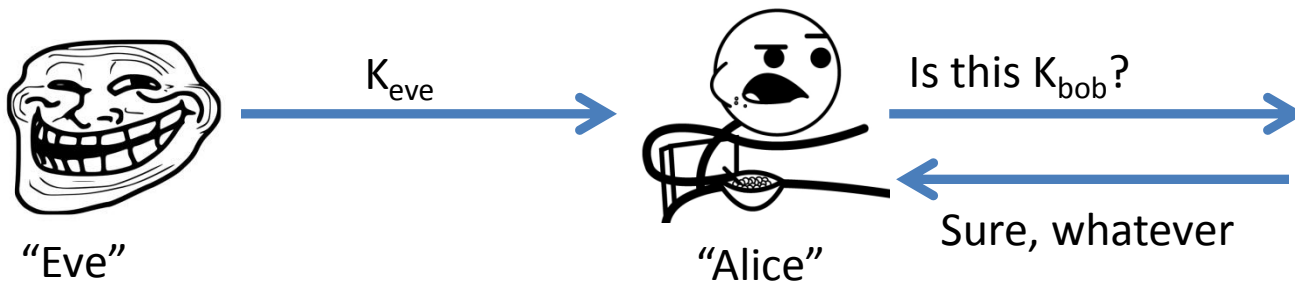
# Still 1970s: Along came DH and RSA

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



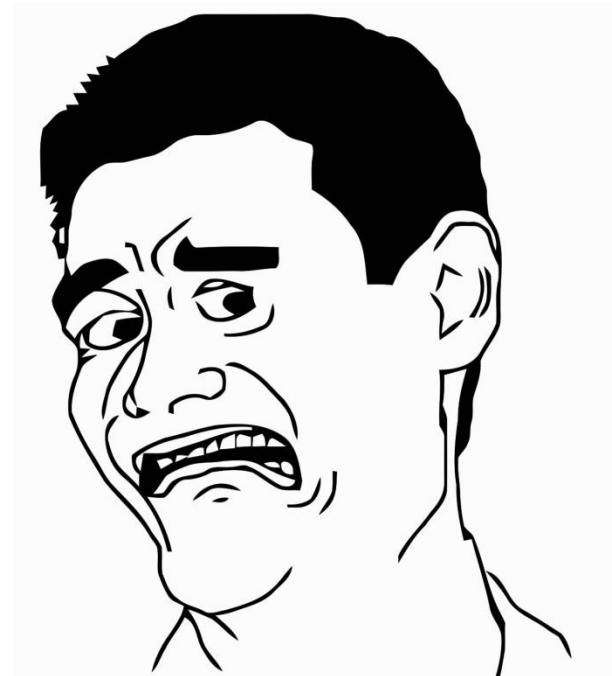
# 1990s: Certification Authorities

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



# 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



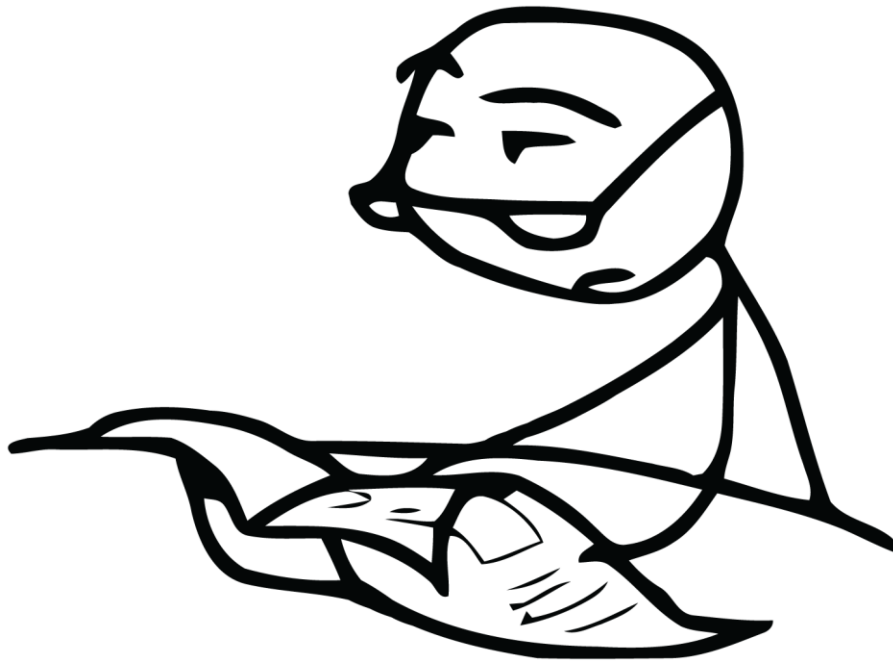
The image shows a screenshot of a file viewer interface. At the top, a header bar displays 'file | 17 lines (14 sloc) | 0.37 kb'. Below this, a list of line numbers from 1 to 16 is on the left. The corresponding code is on the right. Lines 1-3 contain header information for an RSA private key. Lines 5-14 contain an ASCII art drawing of a smiley face. Line 16 contains the footer information.

```
1  -----BEGIN RSA PRIVATE KEY-----
2  Proc-Type: 4,ENCRYPTED
3  DEK-Info: DES-EDE3-CBC,CAFEFABE
4
5          /~/)
6      ,/  /
7      /  /
8  /~/ /  /~/~
9  /'/ /  /  /'~\
10 ('(  '  '  '~/'  ')
11 \    '  '  '  /
12  '  \    -  /
13  \    (
14  \ == RSA == \
15
16  -----END RSA PRIVATE KEY-----
```



# 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!



Dangerously oversimplified, because this is Shmoocon and most of you probably know this...

## **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

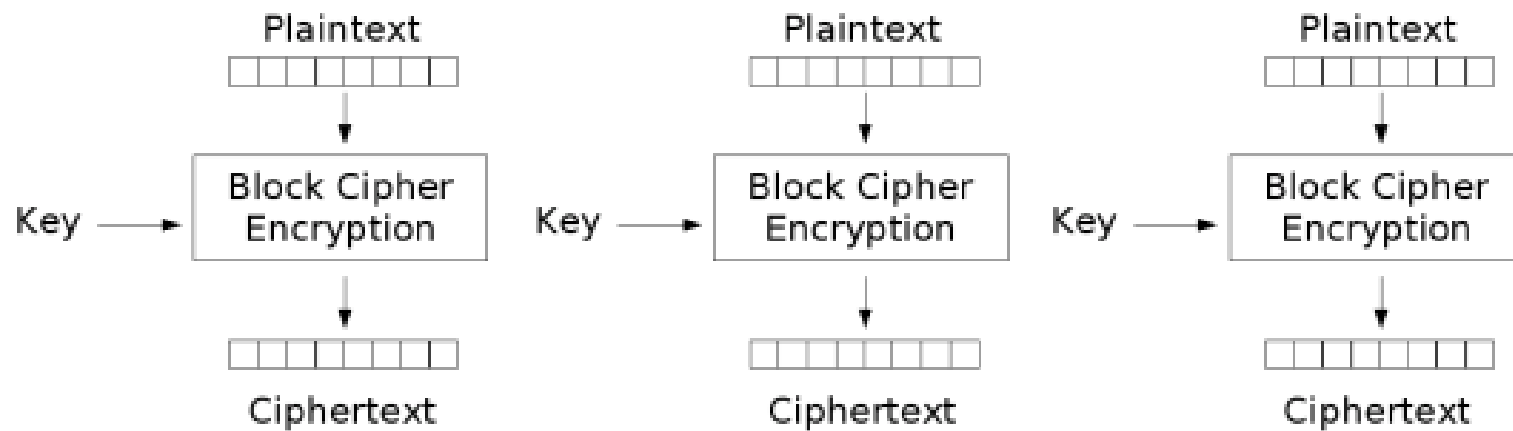
# Encryption: Stream cipher

- Each byte is encrypted by “XOR”ing it with a “keystream”
- Keystream is typically the output of a random number generator
- Meant to simulate a 1-time pad

Plaintext	H	E	L	L	O		W	O	R	L	D
Keystream	3e	83	82	d1	7a	5a	c5	b8	ca	3c	1d
<b>Cipher</b>	<b>76</b>	<b>c6</b>	<b>ce</b>	<b>9d</b>	<b>35</b>	<b>7a</b>	<b>92</b>	<b>f7</b>	<b>98</b>	<b>70</b>	<b>59</b>
Keystream (again)	3e	83	82	d1	7a	5a	c5	b8	ca	3c	1d
Plaintext	H	E	L	L	O		W	O	R	L	D

# Encryption: Block cipher

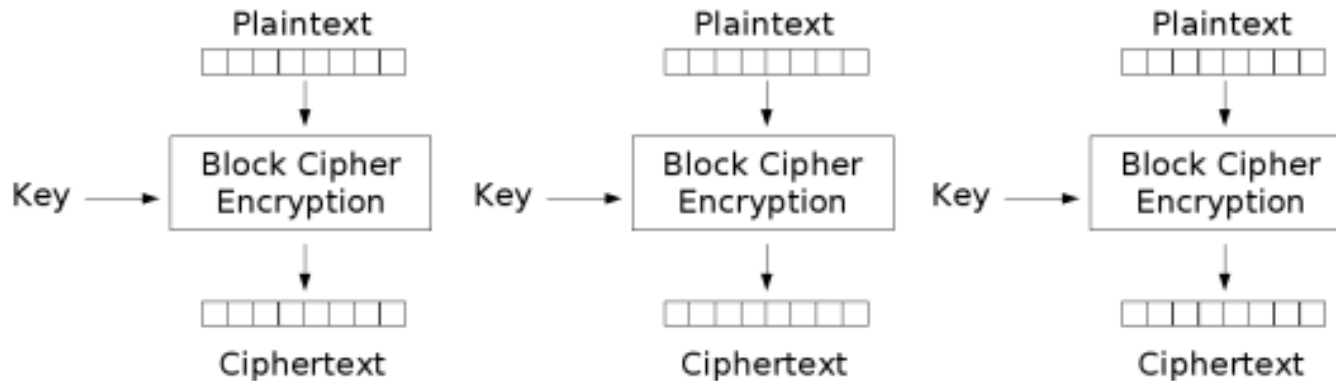
- Plaintext is broken into 8- or 16-byte blocks, each is encrypted individually
- Various “modes of operation” can be used to ensure that the ciphertext isn’t repeated



Electronic Codebook (ECB) mode encryption

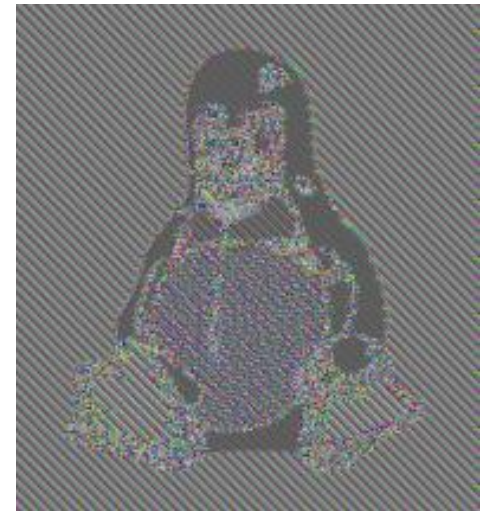
# Encryption: Block cipher modes of operation – ECB

- “Electronic codebook” mode encrypts each block individually:



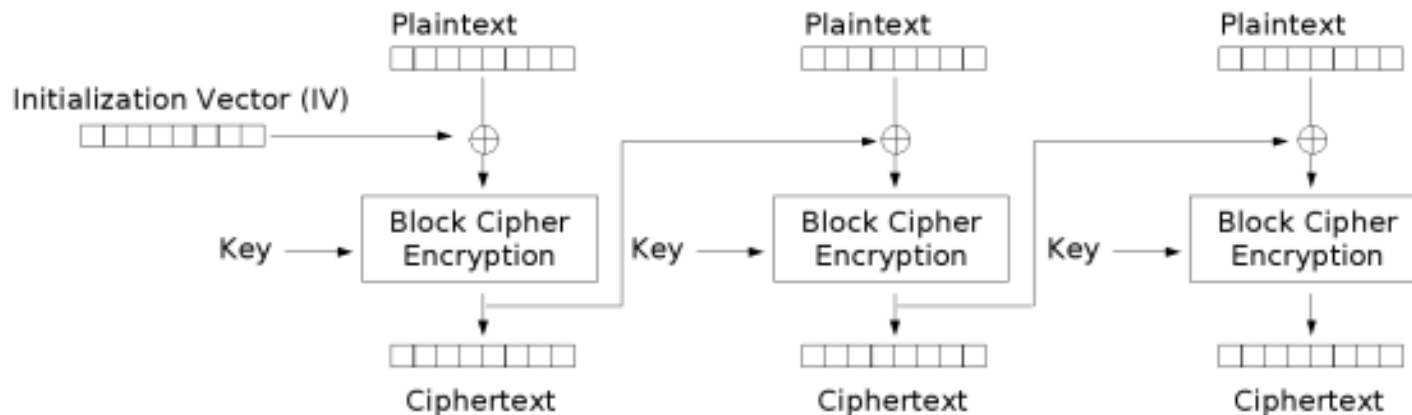
Electronic Codebook (ECB) mode encryption

- 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:



Cipher Block Chaining (CBC) mode encryption

- 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

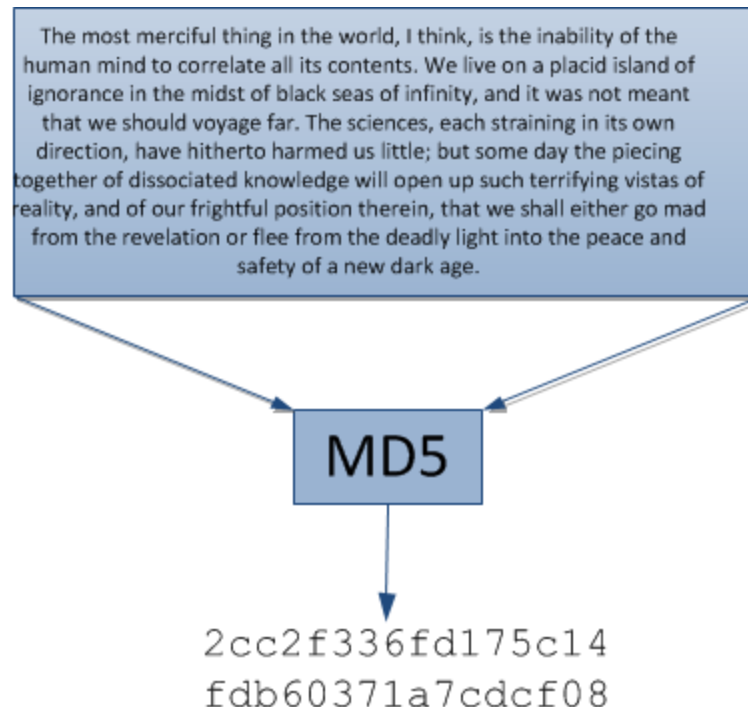
# Initialization vectors: IVs

- 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**



**fuck yeah**

Cut out due to lack of time ☹️

# KEY RE-USE IN STREAM CIPHERS



**fuck yeah**

# Bit-flipping against stream ciphers

- First, let's review stream ciphers...

$$C = (P \oplus K)$$

$$P = (C \oplus K) = ((P \oplus K) \oplus K)$$

Text is XORed with keystream, XORed again to get it back

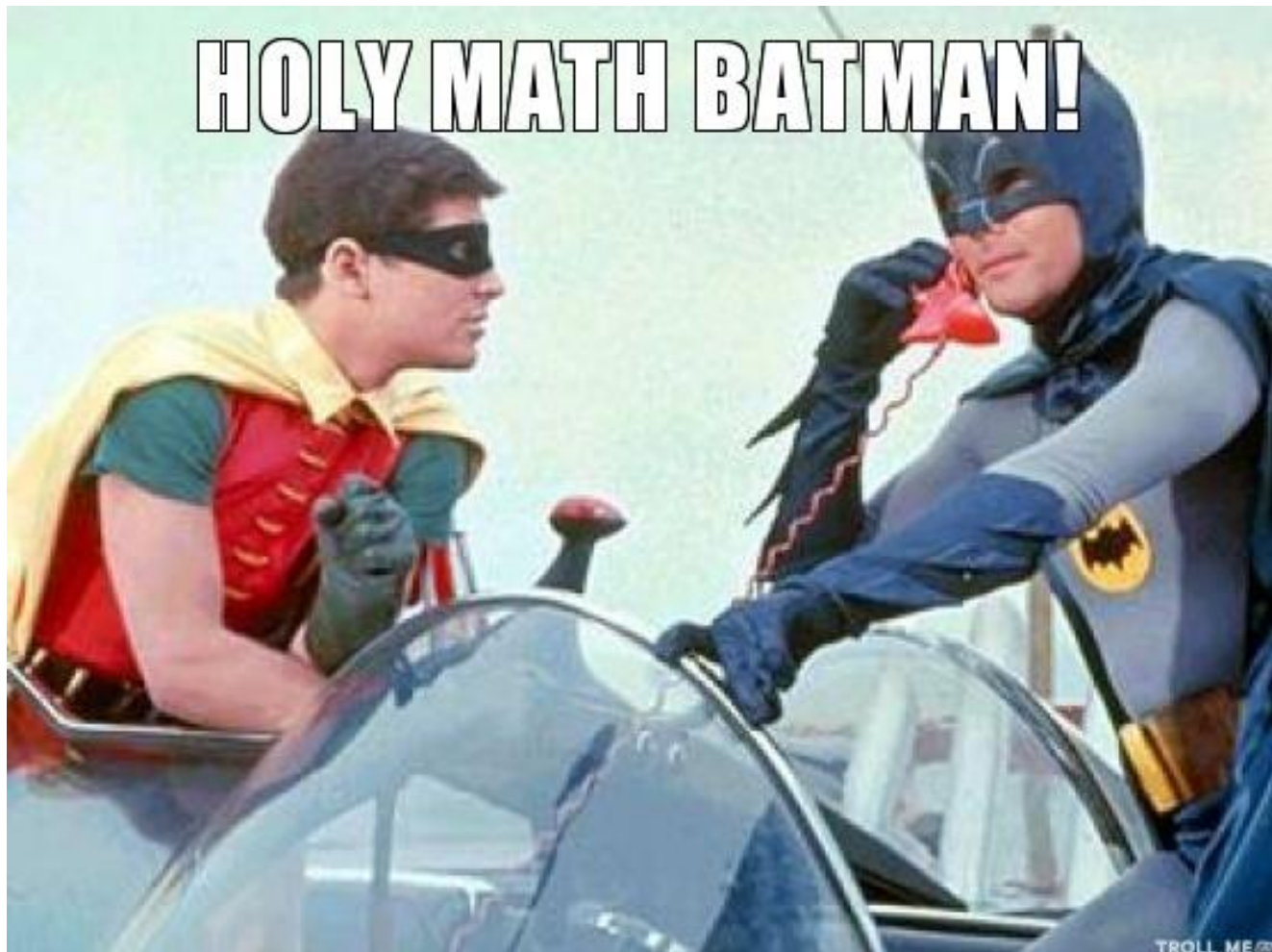
- Now the attack...

$$C = (P \oplus K)$$

$$C' = (C \oplus x) = ((P \oplus K) \oplus x)$$

$$\mathbf{P'} = (C' \oplus K) = (((P \oplus K) \oplus x) \oplus K) = \mathbf{(P \oplus x)}$$

That's a lot of math!



# Bit-flipping example

> Ping 4.2.2.1:

```
0000 00 ff 7b 92 b7 b8 00 ff 7a 92 b7 b8 08 00 45 00
0010 00 3c 2b 59 00 00 80 01 ff 3e 0a 15 00 12 04 02
0020 02 01 08 00 4d 2d 00 01 00 2e 61 62 63 64 65 66
0030 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76
0040 77 61 62 63 64 65 66 67 68 69
```

```
..{.....z.....E.
.<+Y.....>.....
....M-....abcdef
ghijklmnopqrstuv
wabcdefghi
```

Pcap from pinging 4.2.2.1  
(the DST IP field is highlighted)

> Note index 0x1e – “04 02 02 01” – the IP address

> RC4(‘THISISMYGOODRC4KEY’, packet):

```
0000 08 98 56 8E 46 2F 29 58 F4 08 9E C0 D7 6B 76 29
0010 FB BD DE F4 71 E7 0F CF 35 81 21 EC 01 E7 17 AC
0020 A5 9F 0E 42 9D 66 D9 38 B4 A5 30 D3 C9 26 3D DE
0030 E2 18 EE 1D 53 93 BC 54 C4 77 4A 24 8D 6E 10 9E
0040 88 D4 37 9F 65 64 29 25 7D 8E
```

```
..V.F/)X.....kv)
....q...5.!.....
...B.f.8..0..&=.
....S..T.wJ$.n..
..7.ed)%}.
```

Encrypt the full packet  
with IP4 (even the link  
header, for simplicity)

> Index 0x1e is now “17 ac a5 9f”

> 17 ac a5 9f  $\oplus$  04 02 02 01  $\oplus$  08 08 08 08 = 1b a6 af 96:

```
0000 08 98 56 8E 46 2F 29 58 F4 08 9E C0 D7 6B 76 29
0010 FB BD DE F4 71 E7 0F CF 35 81 21 EC 01 E7 1B A6
0020 AF 96 0E 42 9D 66 D9 38 B4 A5 30 D3 C9 26 3D DE
0030 E2 18 EE 1D 53 93 BC 54 C4 77 4A 24 8D 6E 10 9E
0040 88 D4 37 9F 65 64 29 25 7D 8E
```

```
..V.F/)X.....kv)
....q...5.!.....
...B.f.8..0..&=.
....S..T.wJ$.n..
..7.ed)%}.
```

XOR the DST IP field with  
the real IP (4.2.2.1) then  
another IP (8.8.8.8)

> RC4(‘THISISMYGOODRC4KEY’, updated\_packet):

```
0000 00 FF 7B 92 B7 B8 00 FF 7A 92 B7 B8 08 00 45 00
0010 00 3C 2B 59 00 00 80 01 FF 3E 0A 15 00 12 08 08
0020 08 08 08 00 4D 2D 00 01 00 2E 61 62 63 64 65 66
0030 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76
0040 77 61 62 63 64 65 66 67 68 69
```

```
..{.....z.....E.
.<+Y.....>.....
....M-....abcdef
ghijklmnopqrstuv
wabcdefghi
```

Decrypt the packet, note  
the DST IP is now 8.8.8.8

# Why does bit flipping matter?

- The plaintext packet can be controlled by modifying the encrypted packet
- This works against any algorithm that XORs the plaintext with a keystream
  - RC4
  - One-time pads
  - Block ciphers in OFB, PFB, or CTR mode (we'll talk more about those later!)

# How to prevent bit-flipping attacks

- Prepend a HMAC() hash to encrypted data
- **Proper** HMAC(), not  $H(\text{secret} \parallel \text{data})$ 
  - We'll see why when we talk about hash-length-extension attacks
- To not fail crypto:
  - Encrypt: Encrypt, **then** hash
  - Decrypt: Verify hash, **then** decrypt



# “The Cryptographic Doom Principle”



# Key re-use in stream ciphers

- In stream ciphers, the same key will generate the same keystream
- If you encrypt different data with the same keystream, you're gonna have a bad time
  - (I'm looking at you, WEP)
- That's why the initialization vector – IV – was invented

# Key re-use in stream – example

```
> ping = [Ping 4.2.2.1]
```

```
> enc_packet = RC4('THISISMYGOODRC4KEY', ping):
```

```
0000 08 98 56 8E 46 2F 29 58 F4 08 9E C0 D7 6B 76 29 ..V.F/)X....kv)
0010 FB BD DE F4 71 E7 0F CF 35 81 21 EC 01 E7 17 AC ....q...5.!.....
0020 A5 9F 0E 42 9D 66 D9 38 B4 A5 30 D3 C9 26 3D DE ...B.f.8..0..&=.
0030 E2 18 EE 1D 53 93 BC 54 C4 77 4A 24 8D 6E 10 9E ....S..T.wJ$.n..
0040 88 D4 37 9F 65 64 29 25 7D 8E ..7.ed)%}.
```

Encrypt something that's well known to the attacker

```
> keystream = [Ping 4.2.2.1]  $\oplus$  RC4('THISISMYGOODRC4KEY', [Ping 4.2.2.1]):
```

```
0000 08 67 2D 1C F1 97 29 A7 8E 9A 29 78 DF 6B 33 29 .g-...)...)x.k3)
0010 FB 81 F5 AD 71 E7 8F CE CA BF 2B F9 01 F5 13 AE ....q.....+.....
0020 A7 9E 06 42 D0 4B D9 39 B4 8B 51 B1 AA 42 58 B8 ...B.K.9..Q..BX.
0030 85 70 87 77 38 FF D1 3A AB 07 3B 56 FE 1A 65 E8 .p.w8.....;V..e.
0040 FF B5 55 FC 01 01 4F 42 15 E7 ..U...OB..
```

XOR the ciphertext with the known plaintext

```
> telnet = [Telnet www.skullsecurity.org "PASS: ???"]
```

```
> secret = RC4('THISISMYGOODRC4KEY', telnet):
```

```
0000 08 98 56 8E 46 2F 29 58 F4 08 9E C0 D7 6B 76 29 ..V.F/)X....kv)
0010 FB BE 8B 1B 31 E7 0F C8 2B D9 21 EC 01 E7 DD 72 ....1...+.!....r
0020 66 06 CA C8 D7 9B A7 E6 5E C1 A5 F6 D7 A0 08 A0 f.....^.....
0030 BA 87 41 38 38 FF 81 7B F8 54 01 76 93 63 16 8D ..A88..{.T.v.c..
0040 9C C7 30 88 71 60 3C 31 62 88 CE 0A F1 ..0.q`<1b....
```

Encrypt something unknown using the same key

```
> result = (secret  $\oplus$  keystream)
```

```
0000 00 FF 7B 92 B7 B8 00 FF 7A 92 B7 B8 08 00 45 00 ..{.....z.....E.
0010 00 3F 7E B6 40 00 80 06 E1 66 0A 15 00 12 CE DC .?~.@....f.....
0020 C1 98 CC 8A 07 D0 7E DF EA 4A F4 47 7D E2 50 18 .....~..J.G}.P.
0030 3F F7 C6 4F 00 00 50 41 53 3A 20 6D 79 73 65 ?..O..PASS:.myse
0040 63 72 65 74 70 61 73 73 77 6F cretpasswo
```

XOR the secret with the keystream we derived

# Key re-use in stream: What's happening?

- By encrypting known data, we can derive the keystream
- We can then use that keystream to decrypt anything else encrypted with that key

# Key re-use in stream: How to prevent?

- Use different and random initialization vectors (IVs) when encrypting data
- If possible, use a different key (not always possible)

# KEY RE-USE IN BLOCK CIPHERS



**fuck yeah**

# 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)`:

<b>P<sub>1</sub></b>	'A'	'A'	'A'	'A'	'A'	'A'	'A'	'A'	Two blocks of just "A"s (note that the ciphertext is the same)
<b>C<sub>1</sub></b>	\x74	\x31	\xe1	\xf0	\xc6	\x1b	\x35	\x11	
<b>P<sub>2</sub></b>	'A'	'A'	'A'	'A'	'A'	'A'	'A'	'A'	
<b>C<sub>2</sub></b>	\x74	\x31	\xe1	\xf0	\xc6	\x1b	\x35	\x11	
<b>P<sub>3</sub></b>	'T'	'h'	'i'	's'	' '	'i'	's'	' '	The rest of the string encrypted as-is
<b>C<sub>3</sub></b>	\x35	\x13	\x7b	\x27	\xb6	\xf5	\xda	\x9c	
<b>P<sub>4</sub></b>	's'	'o'	'm'	'e'	' '	't'	'e'	's'	
<b>C<sub>4</sub></b>	\xb1	\x0e	\xdf	\x42	\x93	\xe8	\x17	\x42	
<b>P<sub>5</sub></b>	't'	' '	'd'	'a'	't'	'a'	\x02	\x02	
<b>C<sub>5</sub></b>	\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)`:

<b>P<sub>1</sub></b>	'A'	'A'	'A'	'A'	'A'	'A'	'A'	'T'	← GOAL
<b>C<sub>1</sub></b>	\xea	\xca	\x59	\x30	\x3d	\x8b	\xe6	\x0f	
<b>P<sub>2</sub></b>	'h'	'i'	's'	' '	'i'	's'	' '	's'	Stuff That We Don't Care About ... ... ... ...
<b>C<sub>2</sub></b>	\xf2	\xaa	\xb1	\xfb	\x54	\xb4	\xb5	\x87	
<b>P<sub>3</sub></b>	'o'	'm'	'e'	' '	't'	'e'	's'	't'	
<b>C<sub>3</sub></b>	\x34	\x87	\x06	\x80	\x9a	\xcc	\xad	\x43	
<b>P<sub>4</sub></b>	' '	'd'	'a'	't'	'a'	\x03	\x03	\x03	
<b>C<sub>4</sub></b>	\xd3	\x71	\x2a	\xf5	\x79	\x10	\x25	\xea	
<b>P<sub>1</sub></b>	'A'	'A'	'A'	'A'	'A'	'A'	'A'	'T'	
<b>C<sub>1</sub></b>	\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

← 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'	'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)`:

<b>P<sub>1</sub></b>	'A'	'A'	'A'	'A'	'A'	'A'	'T'	'h'	← GOAL
<b>C<sub>1</sub></b>	\xcb	\x7a	\x74	\xd0	\x38	\x45	\xbf	\x21	
<b>P<sub>2</sub></b>	'i'	's'	' '	'i'	's'	' '	's'	'o'	Stuff That We Don't Care About ... ... ... ...
<b>C<sub>2</sub></b>	\xf9	\x8e	\xcd	\xdf	\x49	\xf0	\x86	\xcb	
<b>P<sub>3</sub></b>	'm'	'e'	' '	't'	'e'	's'	't'	' '	
<b>C<sub>3</sub></b>	\x70	\x8c	\xc0	\x1d	\xe5	\xf2	\xdc	\x01	
<b>P<sub>4</sub></b>	'd'	'a'	't'	'a'	\x04	\x04	\x04	\x04	
<b>C<sub>4</sub></b>	\xb4	\x74	\xfc	\x99	\xd9	\xbe	\xd2	\x70	
<b>P<sub>1</sub></b>	'A'	'A'	'A'	'A'	'A'	'A'	'T'	'h'	
<b>C<sub>1</sub></b>	\xcb	\x7a	\x74	\xd0	\x38	\x45	\xbf	\x21	

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

P <sub>1</sub>	'A'	'A'	'A'	'A'	'A'	'A'	'T'	'h'
C <sub>1</sub>	\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 <sub>1</sub>	'A'	'A'	'A'	'A'	'A'	'A'	'T'	'g'
C <sub>1</sub>	\xbb	\x48	\x96	\xa3	\xb9	\xb5	\xc4	\x32

← Nope

P <sub>2</sub>	'T'	'h'	'i'	's'	' '	'i'	's'	' '
C <sub>2</sub>	\x35	\x13	\x7b	\x27	\xb6	\xf5	\xda	\x9c

P <sub>1</sub>	'A'	'A'	'A'	'A'	'A'	'A'	'T'	'h'
C <sub>1</sub>	\xcb	\x7a	\x74	\xd0	\x38	\x45	\xbf	\x21

← Oh hai!

P <sub>1</sub>	'A'	'A'	'A'	'A'	'A'	'A'	'T'	'i'
C <sub>1</sub>	\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 and random 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**



**fuck yeah**

# 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!

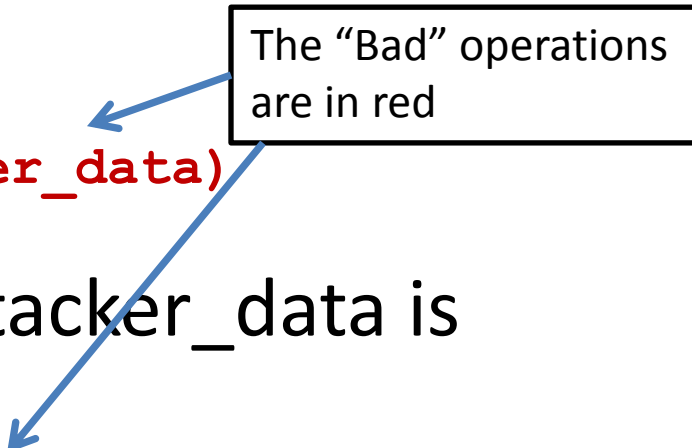
# Hash length extension: the setup

- This is an attack where the following happens:

- Server calculates:

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

The “Bad” operations  
are in red



- Then later, to validate that attacker\_data is valid:

```
if(H(secret || new_attacker_data) != verifier)  
    throw error()  
else  
    trusted_operation(attacker_data)
```

# How hashing works...


- Example hash:
  - `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.") = ed8873c107d882b7b6fcbbbc19a272d614cf9e7ee`

# How hashing works

- First, it's broken up into blocks...
  - "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

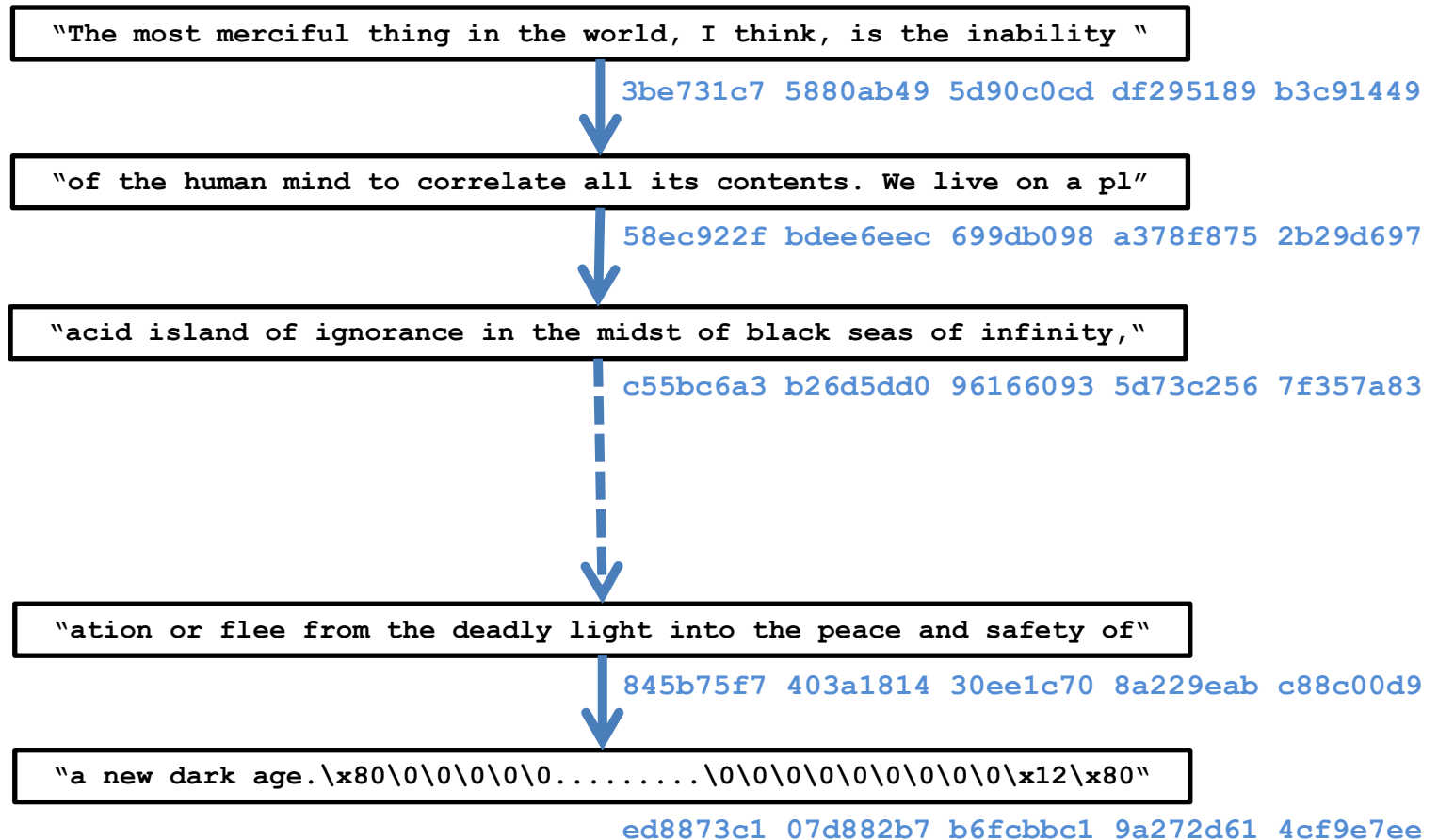
- Padding is added to the last block
  - "The most merciful thing in the world, I think, is the inability "
  - "of the human mind to correlate all its contents. We live on a pl"
  - "acid island of ignorance in the midst of black seas of infinity,"
  - " and it was not meant that we should voyage far. 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\x00\x00\x00.....\x00\x00\x00\x12\x80"
- The padding is equal to a 1-bit followed by a bunch of zero bits (" \x80\x00\x00...")
- The last eight bytes are equal to the length of the string (0x250 bytes) in bits (0x1280 bits)



Dropped a bunch of "\x00" to conserve space

# How hashing works

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

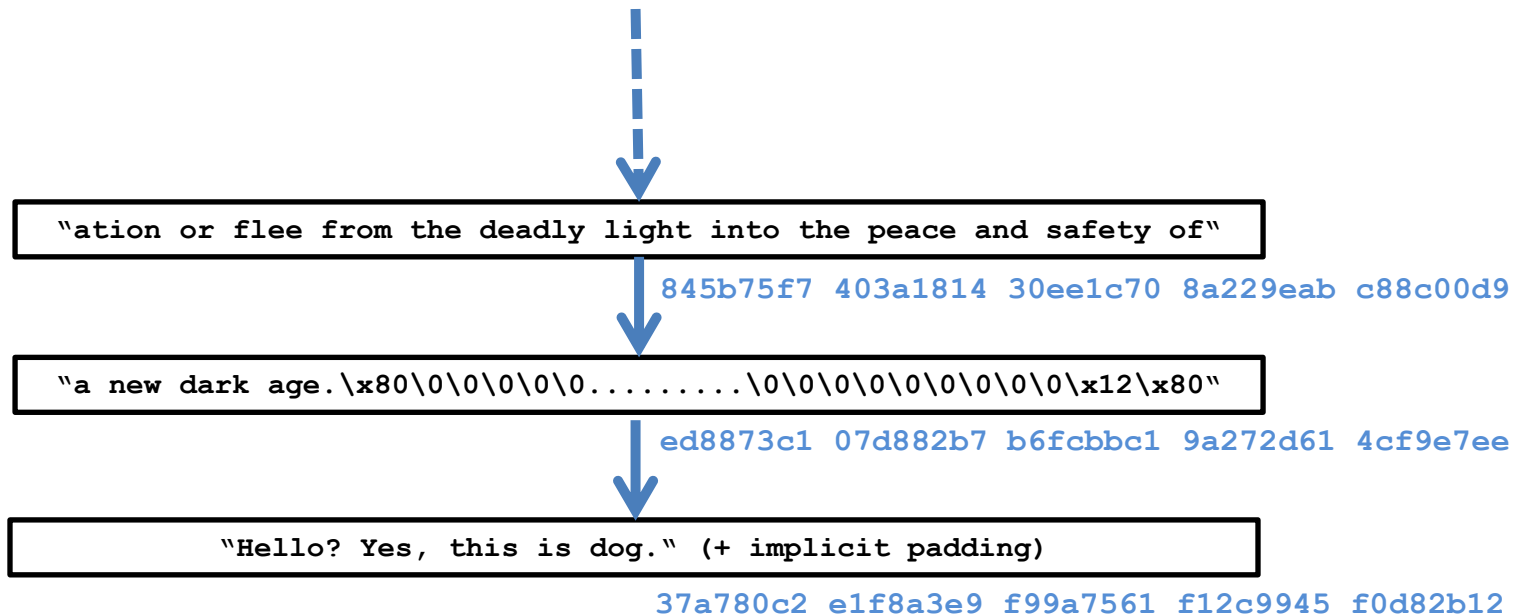


- The output of the last block is our hash!



# Hash extension: We're almost there!

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




- What good is that?



# Hash extension: And here we are!

- 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!



"||" is the "concatenate" operator in crypto.  
@mak\_kolybabi yells at me if I don't use it.

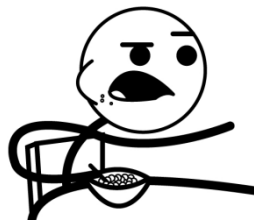
# Hash extension: Applying it

- Let's say you have an API that requires a shared secret
  - `message = SHA1(shared_secret || commands) + commands`
- And let's say commands is a URL-like list of commands.. For example:
  - `commands = "set_firstname=ron&set_lastname=bowes"`
  - Or, `commands = "deletemyaccount=1"`
  - Etc.

# Hash extension: Applying it

- Let's use this API to change my firstname to "ron":

```
- message = SHA1("secretkey" + "set_firstname=ron") +  
  "set_firstname=ron"  
- message =  
  "\x5e\x8a\x88\x6a\x5f\x9c\xa9\x77\x42\x1a\xe0\x67\xcc\x56\x  
55\xb7\xf3\x47\x97\x4dset_firstname=ron"
```

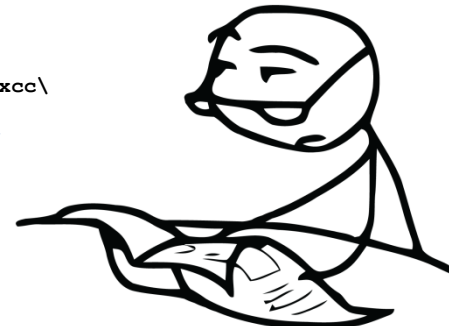


"Ron"

"\x5e\x8a\x88\x6a\x5f\x9c\xa9\x77\x42\x1a\xe0\x67\xcc\x  
56\x55\xb7\xf3\x47\x97\x4dset\_firstname=ron"



"Eve"



"Flickr"

# Hash extension: Applying it

- Now, Eve has a message and its associated hash. What can he use it for?
  - Remember: `SHA1("secretkey" || "set_firstname=ron")` is calculated like this:

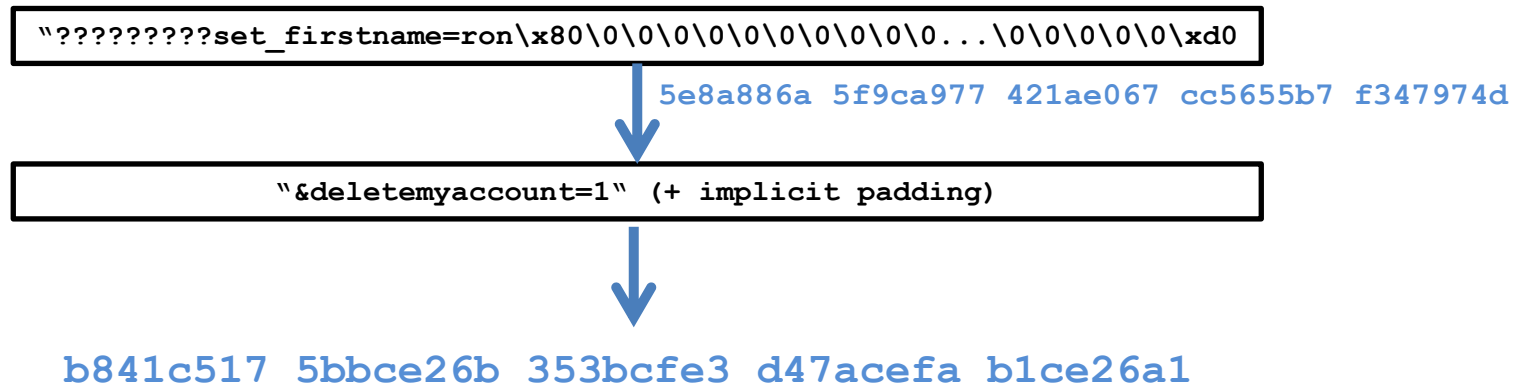
```
"secretkeyset firstname=ron\x80\0\0\0\0\0\0\0\0\0\0...\0\0\0\0\0\0\xd8
```



5e8a886a 5f9ca977 421ae067 cc5655b7 f347974d

# Hash extension: Applying it

- Eve can calculate the following message, given just the output (in blue):



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

# Hash extension: evil client

- Eve writes the following program:

```
12  /* ... */
13  SHA1_Init(&ctx);
14
15  /* This is the evil... */
16  ctx.h0 = 0x5e8a886a; /* This is the hash that Eve captured */
17  ctx.h1 = 0x5f9ca977;
18  ctx.h2 = 0x421ae067;
19  ctx.h3 = 0xcc5655b7;
20  ctx.h4 = 0xf347974d;
21  ctx.N1 = 512; /* The number of bits processed so far */
22              /* (512 bits = 64 bytes = one block) */
23  SHA1_Update(&ctx, "&deletemyaccount=1", 18);
24  SHA1_Final(buf, &ctx);
25
26  /* ... */
```

```
$ gcc -o test test.c -lcrypto
```

```
$ ./test
```

```
b841c5175bbce26b353bcfe3d47acefab1ce26a1
```

# Hash extension: Evil client -> legit server

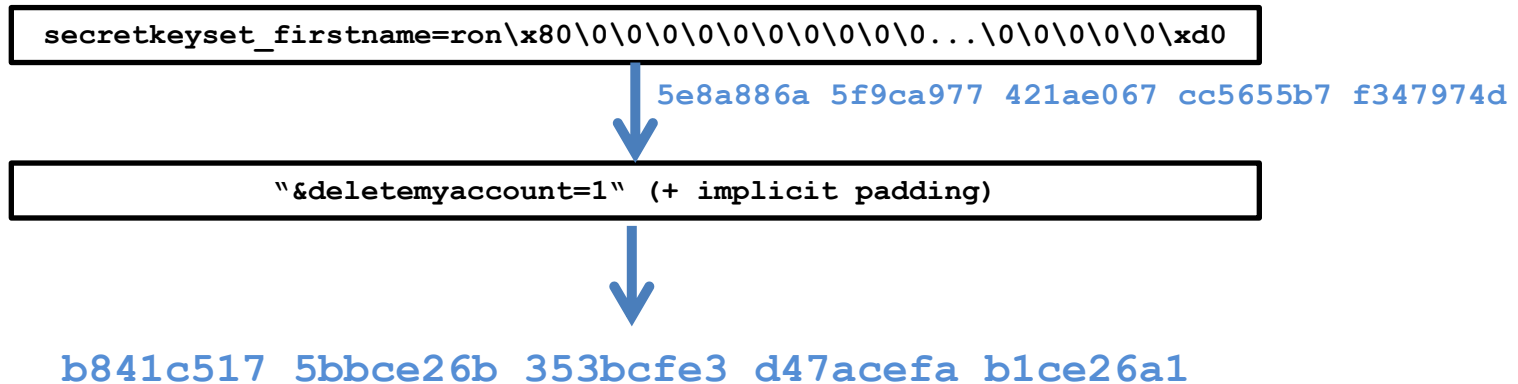
- Eve then sends the following to the server:

<code>"\xb8\x41\xc5\x17\x5b\xbc\xe2\x6b\x35\x3b" + "\xcf\xe3\xd4\x7a\xce\xfa\xb1\xce\x26\xa1" +</code>	The extended hash
<code>"set_firstname=ron" +</code>	The original data
<code>"\x80" + ("\x00" * 36) + "\xd0" +</code>	The original padding
<code>"&amp;deleteaccount=1"</code>	The new data



# Hash extension: Legit server

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



- This can easily be proven in Ruby's irb:

```
Digest::SHA1.hexdigest("secretkeyset_firstname=ron\x80" +  
  ("\\x00" * 36) + "\\xd0&deletemyaccount=1")
```

```
=> "b841c5175bbce26b353bcfe3d47acefab1ce26a1"
```

# 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

# hash\_extender in action

```
./hash_extender --data="set_firstname=ron" --
append("&deletemyaccount=1" --
signature="5e8a886a5f9ca977421ae067cc5655b7f347
974d" --format=sha1 --secret=9 --out-data-
format=cstr
Type: sha1
Secret length: 9
New signature:
b841c5175bbce26b353bcfe3d47acefab1ce26a1
New string:
set\x5ffirstname\x3dron\x80\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
ccount\x3d1
```

# 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:
  - $H(\text{secret} + \text{“knowndata”})$
- He can trivially calculate:
  - $H(\text{secret} + \text{“knowndata”} + \text{padding} + \text{anything})$

# Hash extension: Defense

- HMAC.
- Next topic.

 **DANGER**



**PADDING ORACLES**



**fuck yeah**

# 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 predictable
- 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):

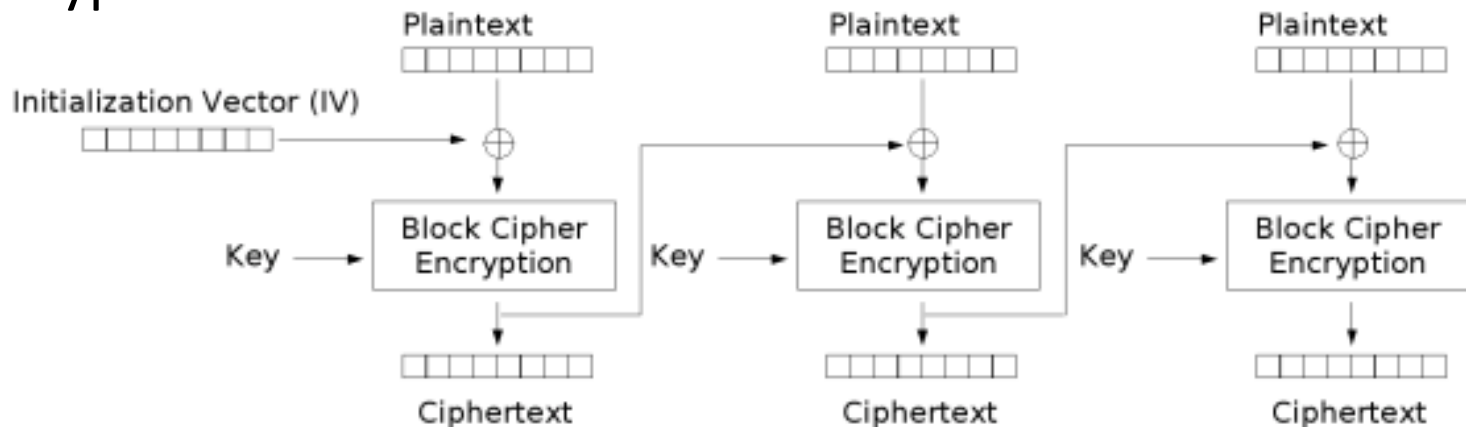
H	e	l	l	o	\x03	\x03	\x03								
H	e	l	l	o		W	o	r	l	d	\x05	\x05	\x05	\x05	\x05
P	a	s	s	w	o	r	d	\x08	\x08	\x08	\x08	\x08	\x08	\x08	\x08
Block 1								Block 2							

# 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

- Let's take a minute to review cipher-block chaining encryption

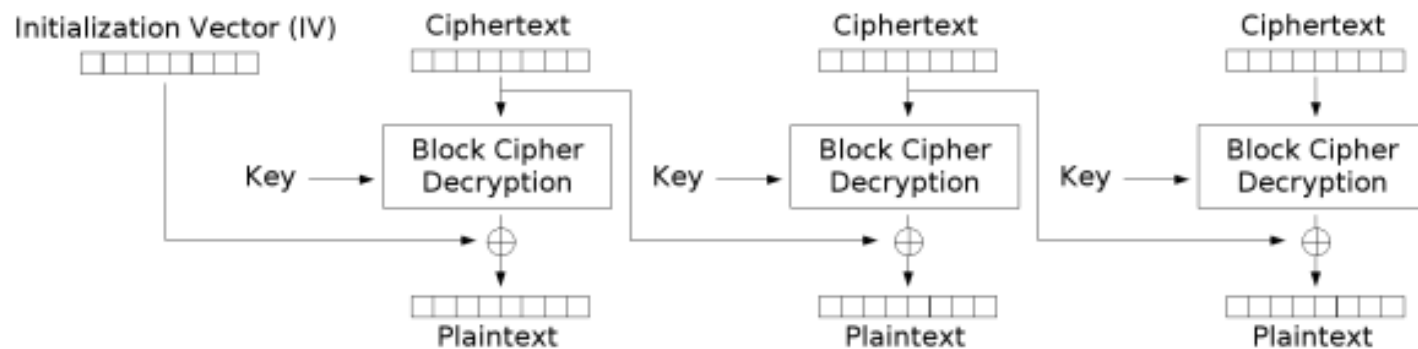


Cipher Block Chaining (CBC) mode encryption

- For any given block of plaintext,  $P_n$ , the corresponding ciphertext,  $C_n$ , can be calculated as:
  - $C_n = E(P_n \oplus C_{n-1})$

# Padding oracles: CBC mode decryption

- Now let's look at decryption:



Cipher Block Chaining (CBC) mode decryption

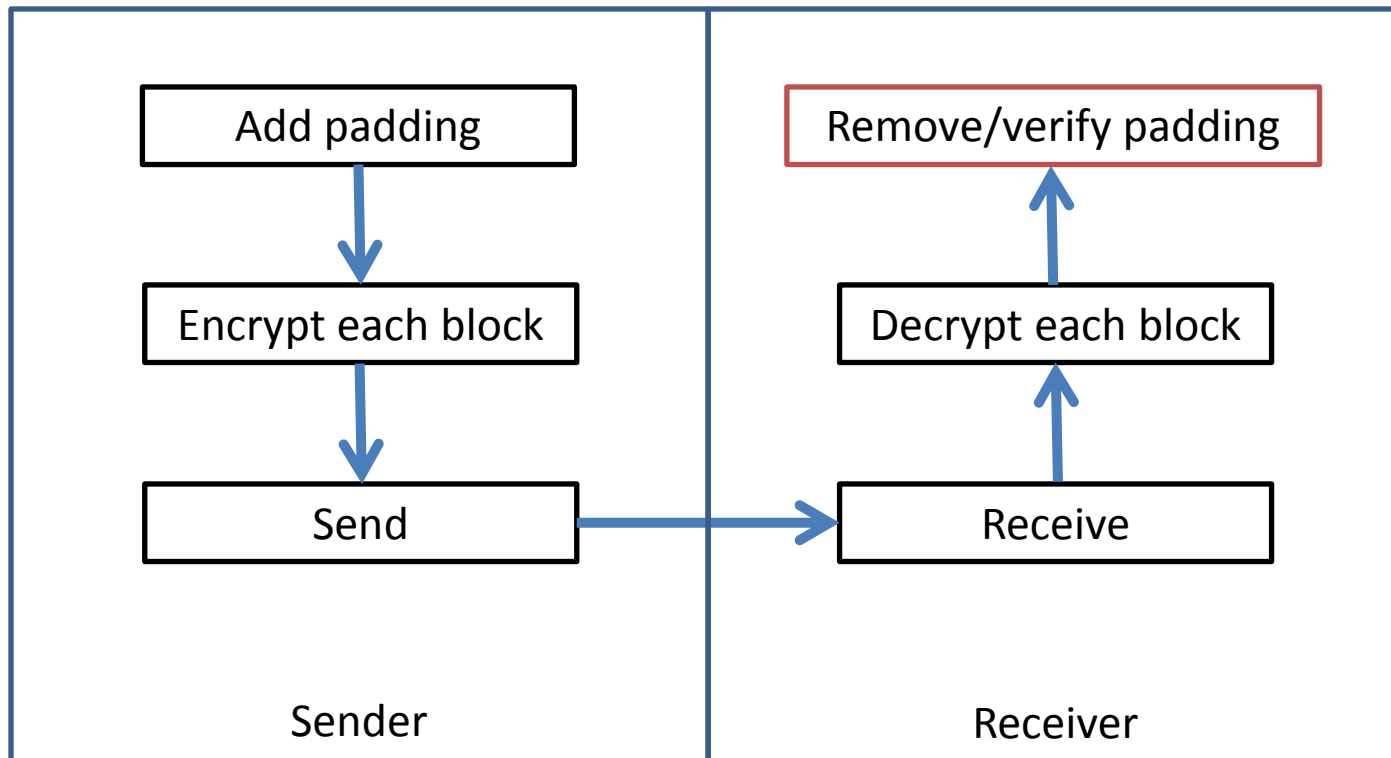
- For any given block of ciphertext,  $C_n$ , we can calculate the corresponding plaintext,  $P_n$ , as:
  - $P_n = D(C_n) \oplus C_{n-1}$

# 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:
  - $P_n = D(E(P_n \oplus C_{n-1})) \oplus C_{n-1}$  Decrypt the encrypted data
  - $P_n = P_n \oplus C_{n-1} \oplus C_{n-1}$  Two XORs cancel out
  - $P_n = P_n$  Success!

# Padding oracles

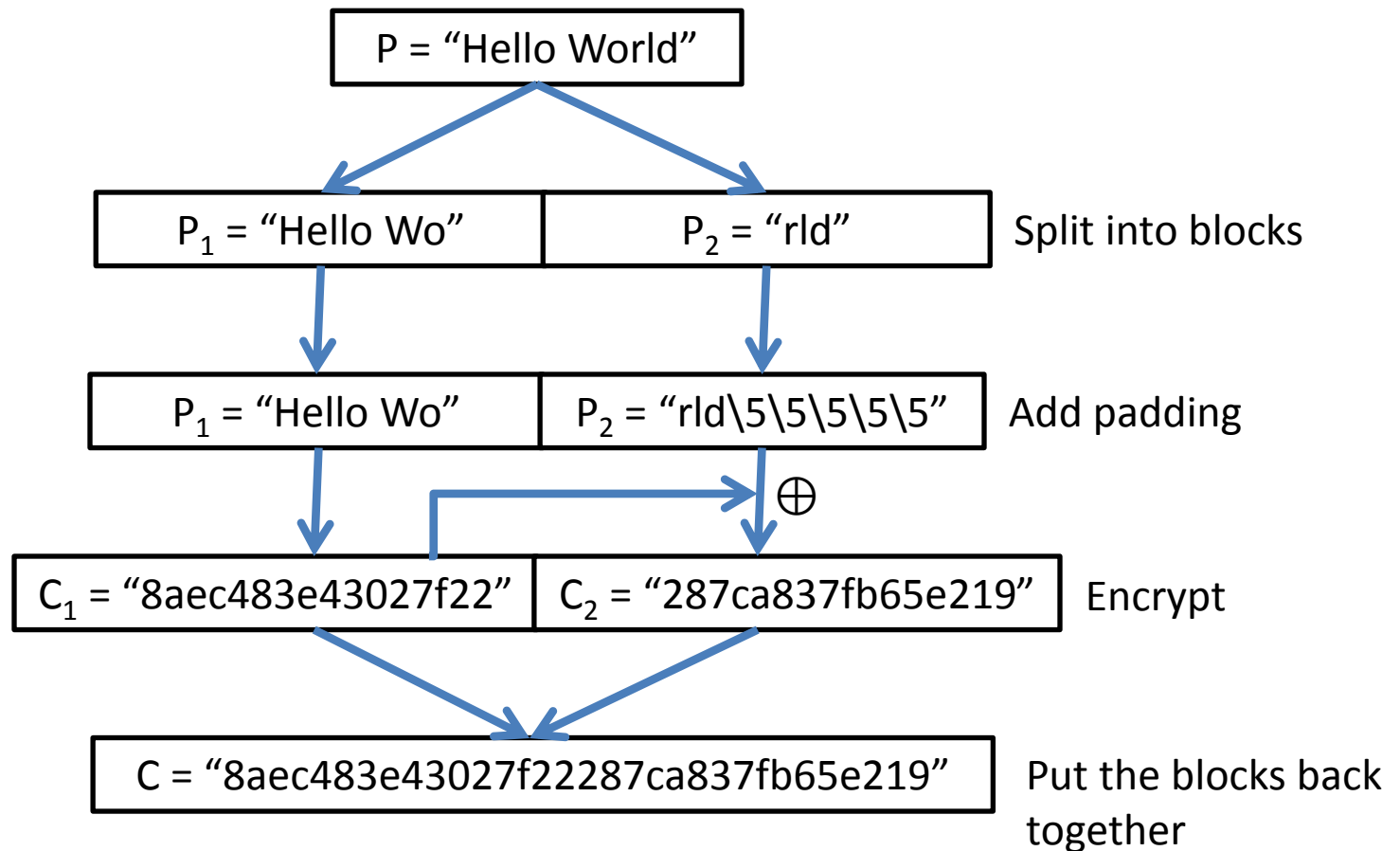
- Encryption steps...





# Padding oracles

- Let's use an example to learn this...



# Padding oracles

- Now, the attacker has the following encrypted string:

$C_1 = \text{"8aec483e43027f22"}$	$C_2 = \text{"287ca837fb65e219"}$
-----------------------------------	-----------------------------------



# Padding oracles

- As an attacker, we take each block individually.  
Let's start with  $C_2$ :

$C_2 = \text{"287ca837fb65e219"}$
-----------------------------------

- We prepend our own block to it, initialized to whatever we want (let's call it  $C'$ ):

$C' = \text{"00000000000000000000"}$	$C_2 = \text{"287ca837fb65e219"}$
$= \text{"00000000000000000000287ca837fb65e219"}$	

# Padding oracles

- The host will decrypt  $C_2$  first:


$$C_2 = \text{"287ca837fb65e219"}$$

- Recall our decryption formula:

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

- $C_{n-1}$  is  $C'$ , in this case, since we prepended  $C'$  to the encrypted string, so we wind up with:

"00000000000000000000000000000000"287ca837fb65e219"


$$P'_2 = D(\textcolor{red}{C}_2) \oplus \textcolor{teal}{C}'$$

# Padding oracles

“00000000000000000000287ca837fb65e219”

This is how the oracle attempts to decrypt the second block

$$P'_2 = D(\mathbf{C}_2) \oplus \mathbf{C}'$$

Let's look at where  $\mathbf{C}_2$  originally came from...

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

Recall our encryption formula from earlier

$$\mathbf{C}_2 = E(P_2 \oplus C_1)$$

$C_2$ , in particular, was encrypted after being XORed with  $C_1$

$$P'_2 = D(E(P_2 \oplus C_1)) \oplus \mathbf{C}'$$

In the formula for  $P'_2$  (above), expand  $\mathbf{C}_2$  to its original value

$$P'_2 = P_2 \oplus C_2 \oplus \mathbf{C}'$$

$E(D(X))$ , like before, becomes  $X$  – this is the value that the oracle calculates on behalf of the attacker

# Padding oracles

$$P'_2 = P_2 \oplus C_1 \oplus \mathbf{c'}$$

The server calculated this value, where...

$P'_2$  = The value the server calculates (mostly a garbage string)

$P_2$  = The original plaintext value (our goal)

$C_1$  = The previous ciphertext block (known to us)

$\mathbf{c'}$  = The ciphertext block chosen by the attacker

("0000000000000000")

- So now we have an equation with two unknowns –  $P_2$  and  $P'_2$ ...
  - Or do we?

# Padding oracles

$$P'_2 = P_2 \oplus C_1 \oplus c'$$

- We actually do know something about  $P'_2$ !
- Recall the last step of decryption...

.....➡ Remove/verify padding

- We know whether or not the padding is correct on the final (garbage) string!
- In fact, we can change the last byte  $C'$  to all 256 possible values, and the server will tell us when the padding is right

# Padding oracles

$$P'_2 = P_2 \oplus C_1 \oplus c'$$

- When the padding is right, we know something about  $P'_2$ , namely, it ends with:
  - `\x01`
  - `\x02\x02`
  - `\x03\x03\x03`
  - ...etc.
  - Let's assume it's `\x01` (the others are rare and can easily be eliminated)
- Remember,  $P'_2$  is mostly a garbage string, still, the result of decrypting a good block and XORing it with a bad block



# Padding oracles

- Let's take another look at that formula:

$$P'_2[N] = P_2[N] \oplus C_1[N] \oplus \mathbf{C}'[N]$$

- When the padding is right, we know the last bytes of all but one:
  - $P'_2[N] - 0x01$
  - $C_1[N]$  – The last byte of the previous block (we know it)
  - $\mathbf{C}'[N]$  – The last byte of the block that we created
- Now it's easy to determine the last byte of  $P_2$  – just re-arrange the formula!

$$P_2[N] = P'_2[N] \oplus C_1[N] \oplus \mathbf{C}'[N]$$

# Padding oracles

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

$C' = \text{"000000000000000000"}$	$C_2 = \text{"287ca837fb65e219"}$
------------------------------------	-----------------------------------

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

$C' = \text{"000000000000000026"}$	$C_2 = \text{"287ca837fb65e219"}$
------------------------------------	-----------------------------------

- Plug it into the formula:

$P_2[N] = P'_2[N] \oplus C_1[N] \oplus \mathbf{C'}[N]$
$P_2[N] = 0x01 \oplus 0x22 \oplus \mathbf{0x26}$

- And solve!

$P_2[N] = 0x05$
-----------------

<p><i>Recall:</i></p> $P_2 = \text{"rld\5\5\5\5\5"}$
--

# 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"

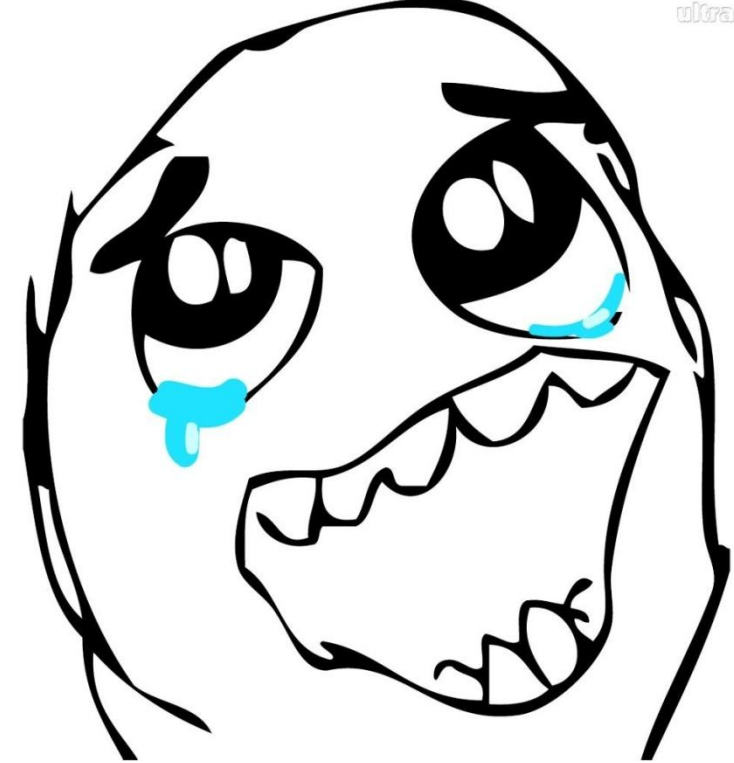
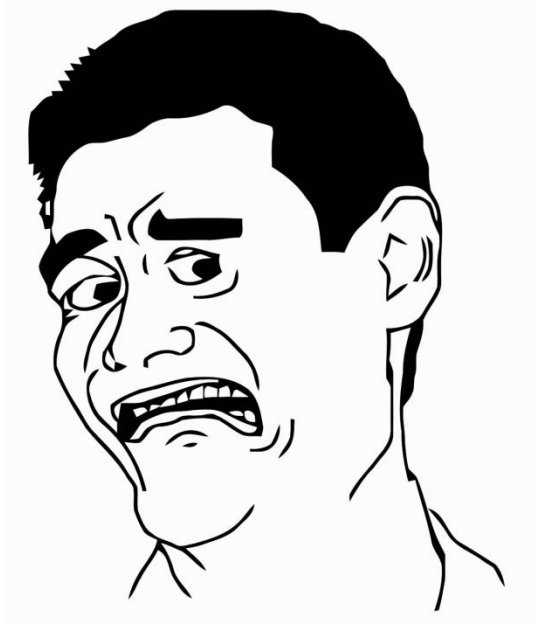


# Padding oracles: Prevention

- 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...

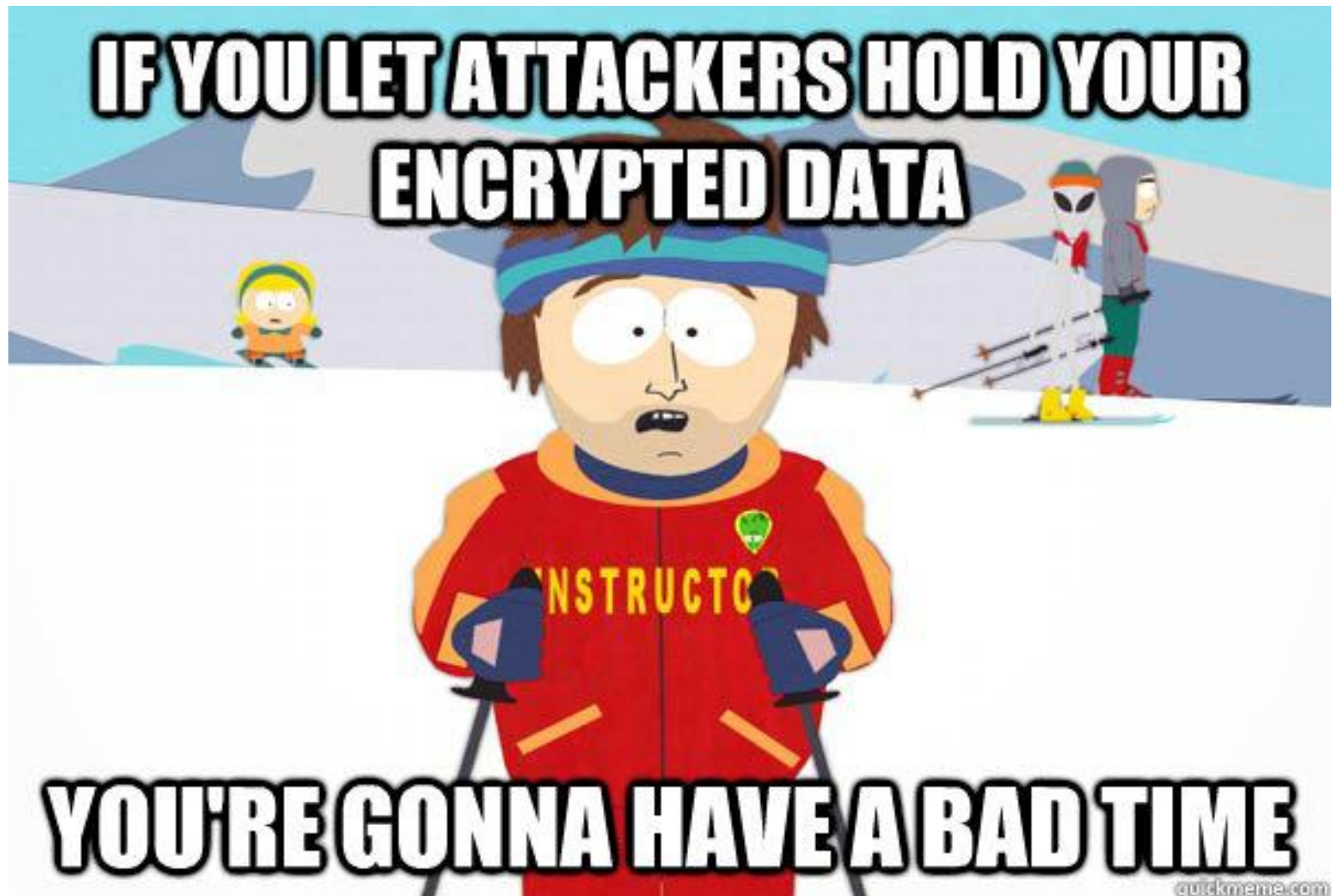
# SOLUTIONS

# 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 encryption in “EAX mode” – a NIST standard for authenticated encryption
- Coming soon: CAESAR
  - **CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness**

“The cryptographic doom principle”

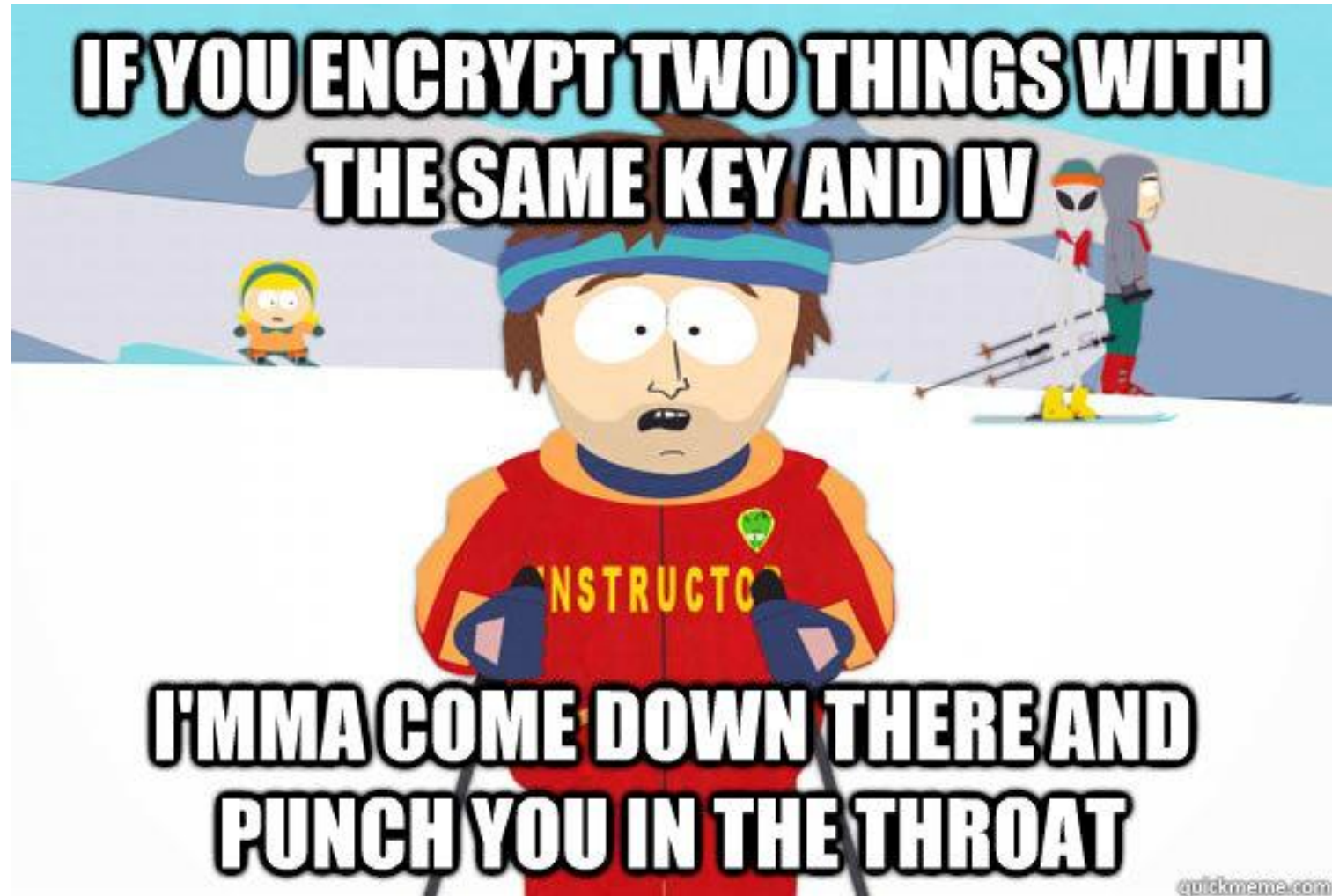


# Solution #3: Never encrypt data with the same key and IV

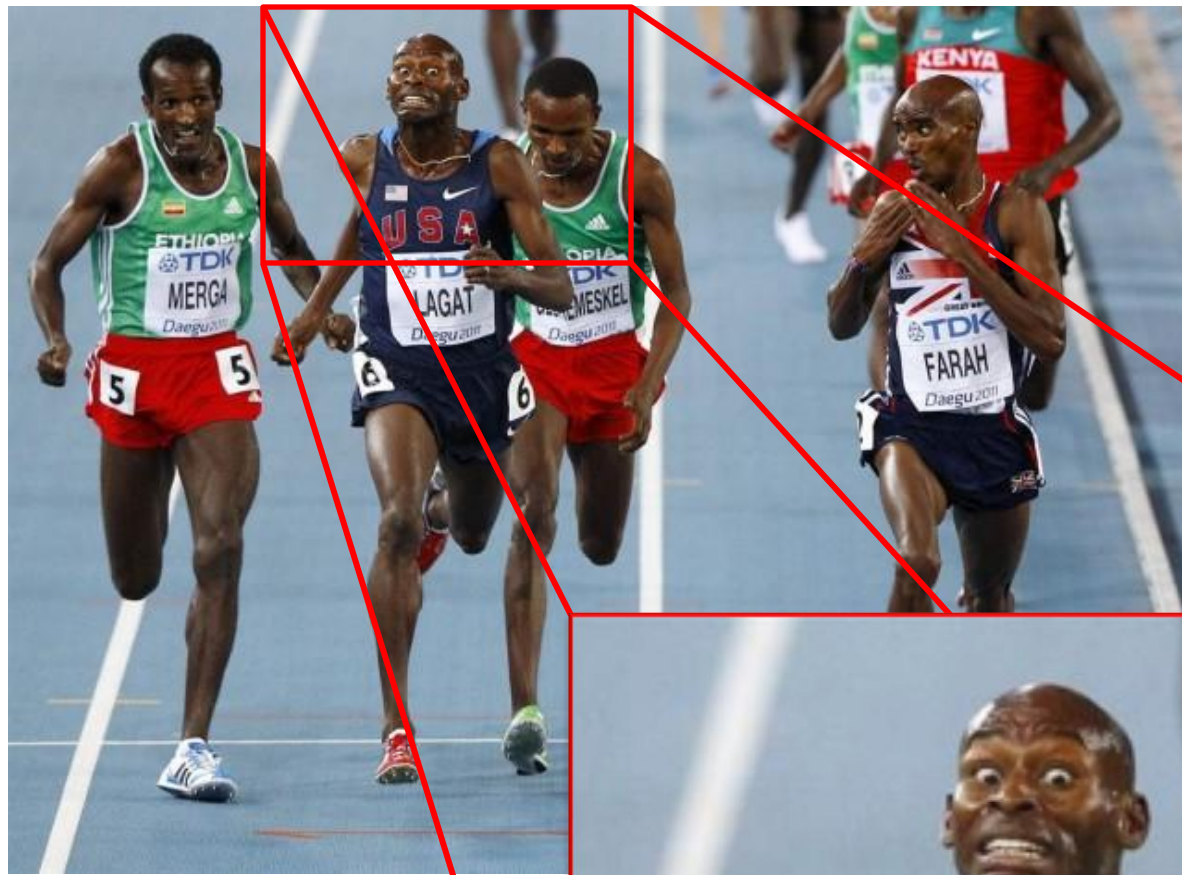
- Almost every cipher fails if you use the same key and IV
- Change keys when it makes sense, and change IVs *every time*



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 released:
  - <https://www.github.com/iagox86/prephixer>
  - <https://www.github.com/iagox86/poracle>
  - [https://www.github.com/iagox86/hash\\_extender](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