How to Implement Crypto Poorly

Sean Cassidy
CTO, DefenseStorm
@sean_a_cassidy

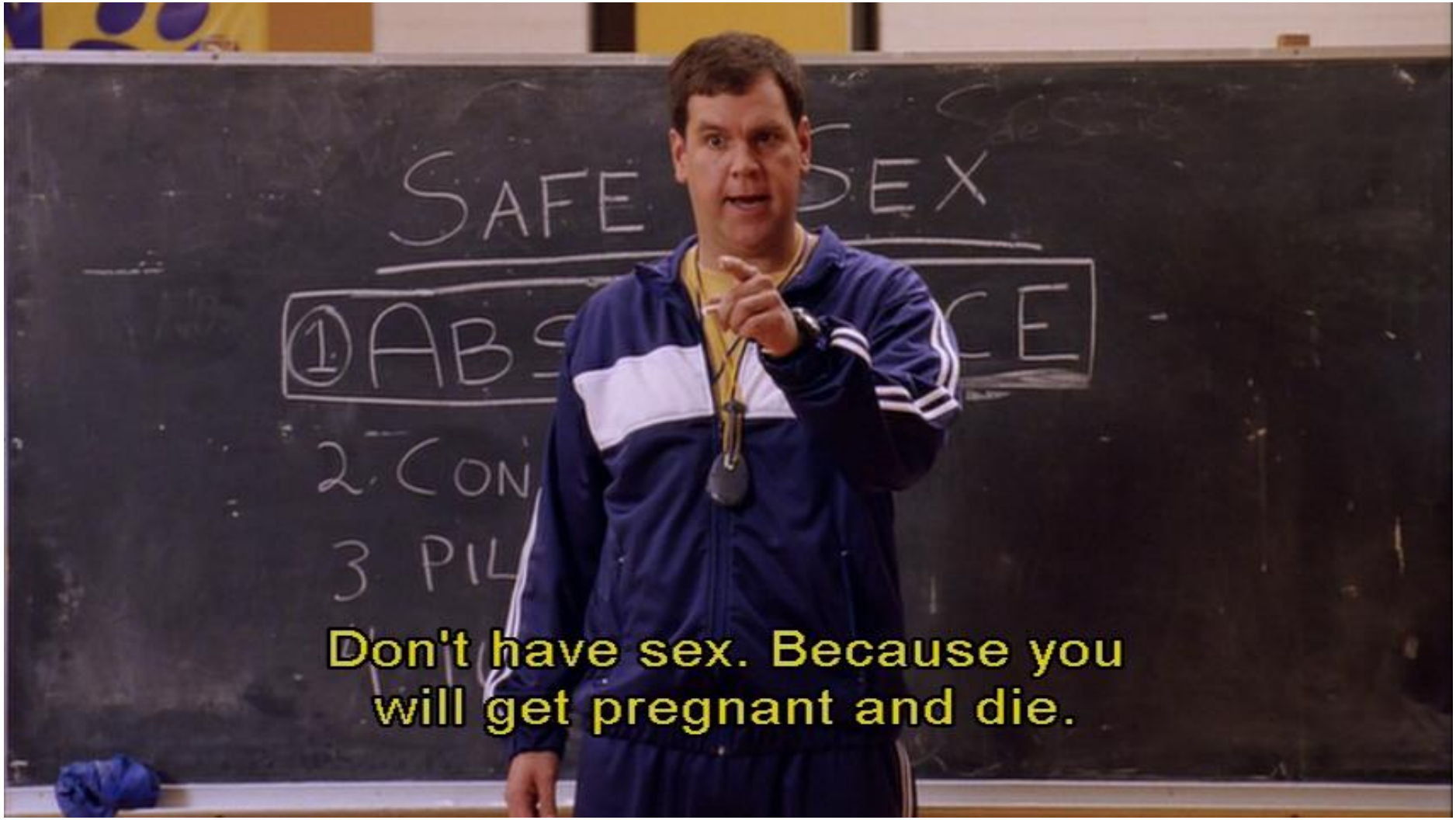
DefenseStorm

Who am I?

1. CTO at DefenseStorm
2. Security Researcher
3. Crypto hobbyist

How to Implement Crypto Poorly

We're always told:
Don't roll your own crypto!


SAFE SEX
1 ABS CE
2 CON
3 PIL
Don't have sex. Because you will get pregnant and die.

Don't roll your own crypto!

- Is this belief justified by evidence?
- Where can you find lots of crypto?
- Further, where can you find lots of hand rolled crypto?
- Custom authentication!
  - Single sign-on

Custom Single Sign-on Survey

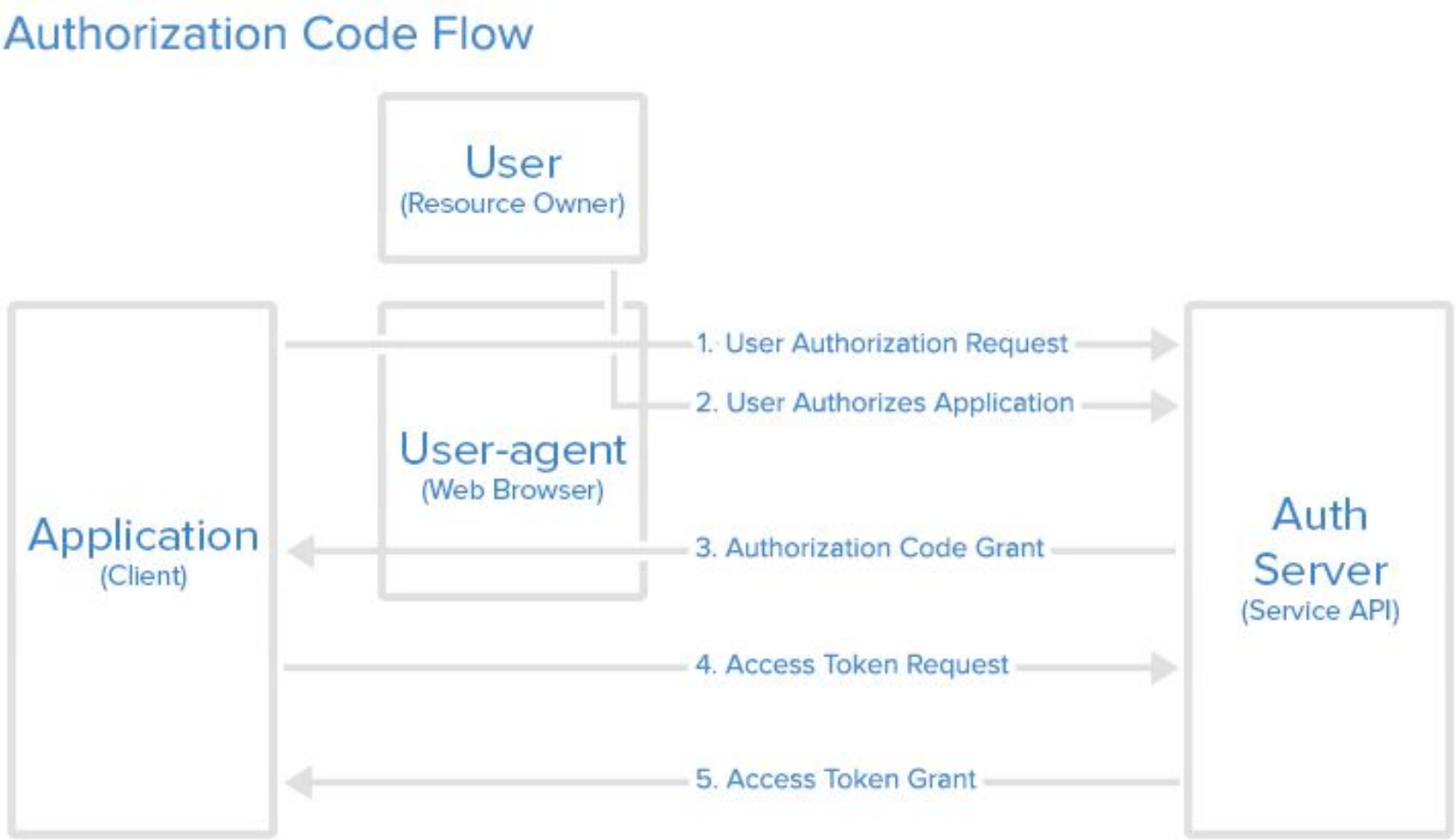
What's single sign-on?

- Log on to a central server which grants access to other systems without needing to log into each directly
- Examples
  - OAuth
  - Facebook Connect
  - SAML
- Benefits
  - Convenience
  - Fewer passwords

What's *custom* single sign-on?

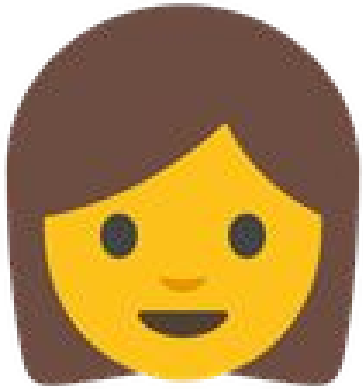
- What if you don't want to use OAuth, Facebook Connect, SAML, LDAP, Shibboleth, etc.?
- Why not?
  - Too hard to implement
  - Want "a few lines of PHP"
- Instead
  - Give a secret to the auth provider
  - Combine it with a few pieces of info
  - Produce secret
  - Check to see if it matches

OAuth


Authorization Code Flow
User
(Resource Owner)
User-agent
(Web Browser)
Application
(Client)
Auth
Server
(Service API)
1. User Authorization Request
2. User Authorizes Application
3. Authorization Code Grant
4. Access Token Request
5. Access Token Grant

What's *custom* single sign-on?

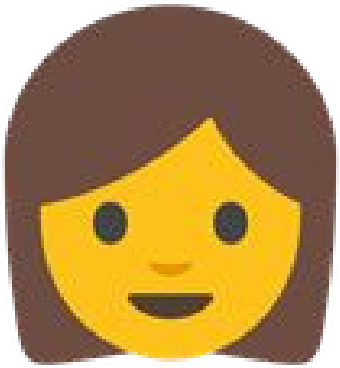
Monkey uses alice.com



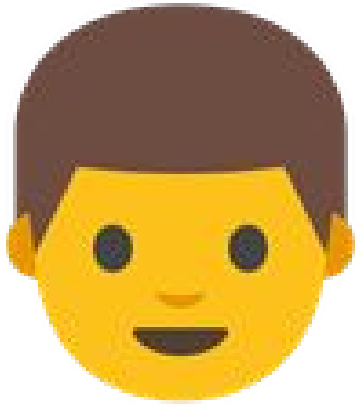
Runs alice.com, a super cool website.

Runs bob.com, a helpdesk website.

What's *custom* single sign-on?



d41d8cd98f00b204e9800998ecf8427e



"Okay, thanks!"

"Your users will need this to sign into my site!"

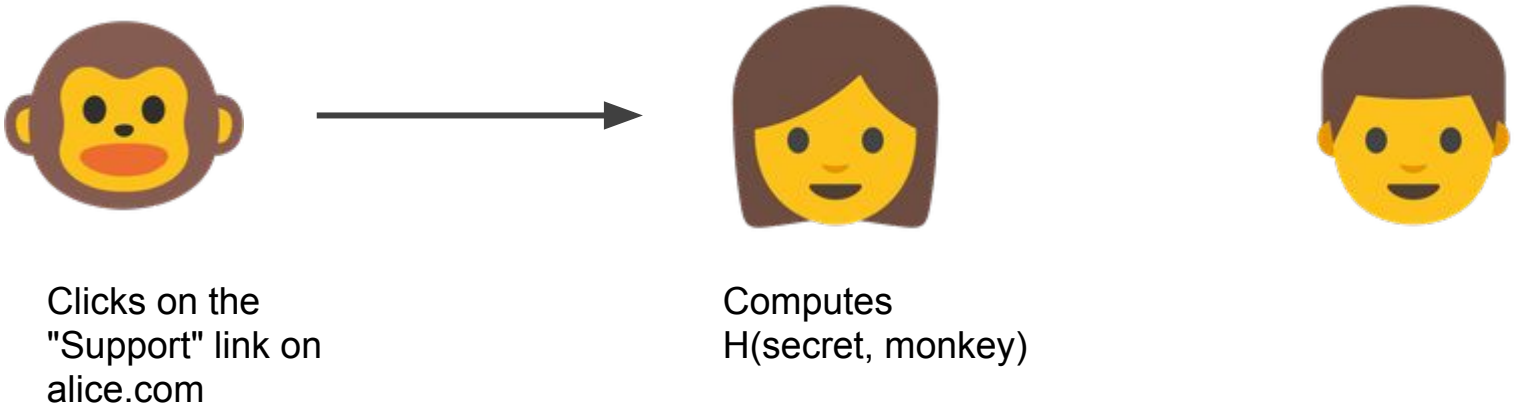
What's *custom* single sign-on?

Logs onto alice.com

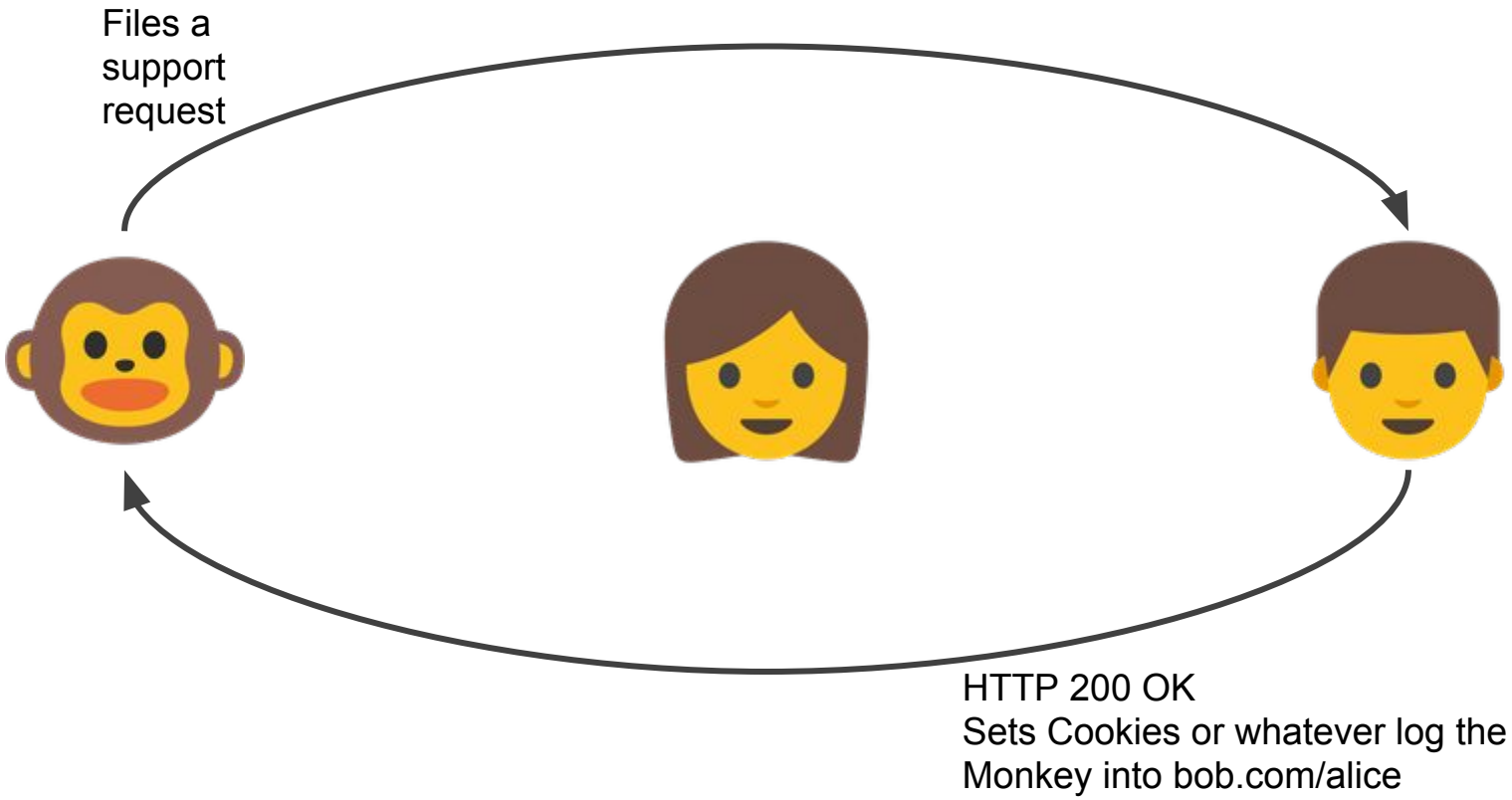
Uses the website like normal

Returns cookies or whatever auth is necessary to be logged into the site

What's *custom* single sign-on?



What's *custom* single sign-on?



What's *custom* single sign-on?

- The good
  - It's simple and fast
- The bad
  - Implementations vary widely
  - No standard implementation
- The ugly
  - They're often extremely insecure
  - Like, really bad

An example problem with custom SSO

I found this emergency fix

- Some kind of HMAC-MD5 known plaintext vuln?
- I wasn't sure what the flaw actually was

```
private static String getHMACHash(
    String name,
    String email,
    long time) throws Exception {
        byte[] keyBytes = sharedSecret.getBytes();
-       String movingFact =name + email + time;
+       String movingFact =name + sharedSecret + email + time;
        byte[] text = movingFact.getBytes();
```

The Freshdesk Flaw

- The plaintext authenticated is

```
name + email + time
```

- So:

```
"Sean" + "sean@defensestorm.com" + time
```

- Is the same as

```
"Seans" + "ean@defensestorm.com" + time
```

The Freshdesk Flaw

https://example.freshdesk.com/login/sso?
name=Seans&email=ean%40defensestorm.com&
timestamp=1475093961&hash=59bcc3ad677556
2f845953cf01624225

is the same as

https://example.freshdesk.com/login/sso?
name=Sean&email=sean%40defensestorm.com&
timestamp=1475093961&hash=59bcc3ad677556
2f845953cf01624225

The Freshdesk Fix

- The *new* plaintext authenticated is

```
name + sharedSecret + email + time
```

- So:

```
"Sean" + "abcdef" +
"sean@defensestorm.com" + time
```

- Is *not* the same as:

```
"Seans" + "abcdef" +
"ean@defnesestorm.com" + time
```

The Custom Single Sign-on Survey

- Goal: identify and catalogue common issues
- Goal: report issues and have them fixed
- Goal: recommend ways to use crypto better
- Non-goal: to deeply inspect each implementation
- Non-goal: to merely recommend OAuth2 or SAML (even though they're objectively better)
- Non-goal: to say "Don't roll your own crypto!" five hundred times

Basic Stats

- 21 Implementations
  - Mostly helpdesk and knowledge base software
  - Two education platforms
- Only one implementation was free from all of the common problems I identified
- One implementation even created their own cipher
- Total user count for all services in the millions

Common Issues with DIY Crypto

No HMAC

- What's an HMAC?
  - It authenticates and validates a message using a hash function
- Why use an HMAC?
  - It's a secure way to combine a secret and a message
  - Prevents length extension attacks
  - Resists most preimage attacks

No HMAC: Length Extension Attacks

- Let's design a basic "message authenticator"

```
H(key || message)
```

- For instance:

```
md5("abcdef" + "/login?admin=false")
```

- Use URL:

```
http://whatever/login?admin=false
   &hash=614c1ae4d5fa3556f092ce79cb7a9e2b
```

No HMAC: Length Extension Attacks

- `H(key || message)` or `H(message || key)`
- If the attacker knows the output of the hash and the message, the attacker can learn enough of the internal state to add to the end of the message without knowing the key
- This is not cheap computationally

No HMAC: Length Extension Attacks

- Sign this URL part:

  `/login?admin=false`

- Results in:

  `/login?admin=false&hash=05f061d1b4b8ca8d`
  `f756928a1170db7a`

No HMAC: Length Extension Attacks

- Results in:

  `/login?admin=false&hash=05f061d1b4b8ca8df756928a1170db7a`

- Length extension attack:

  `/login?admin=false&admin=true&hash=0a0d24f4217f5f8e75277ef5408939d7`

No HMAC: Length Extension Attacks

- Almost all of the custom SSO that did not use an HMAC could be exploited using a length extension attack
- This bug has hit
  - Flickr
  - Amazon
  - A popular content delivery network
  - Many others

No HMAC: Preimage Attacks

- `H(m) = y`
- `H_inv(y) = m`

  Preimage resistance means `H_inv` should be very expensive

- `H(m) = H(m*)`

  Secondary preimage resistance means it should be very hard to find `m*` given `m != m*`

No HMAC: Preimage Attacks

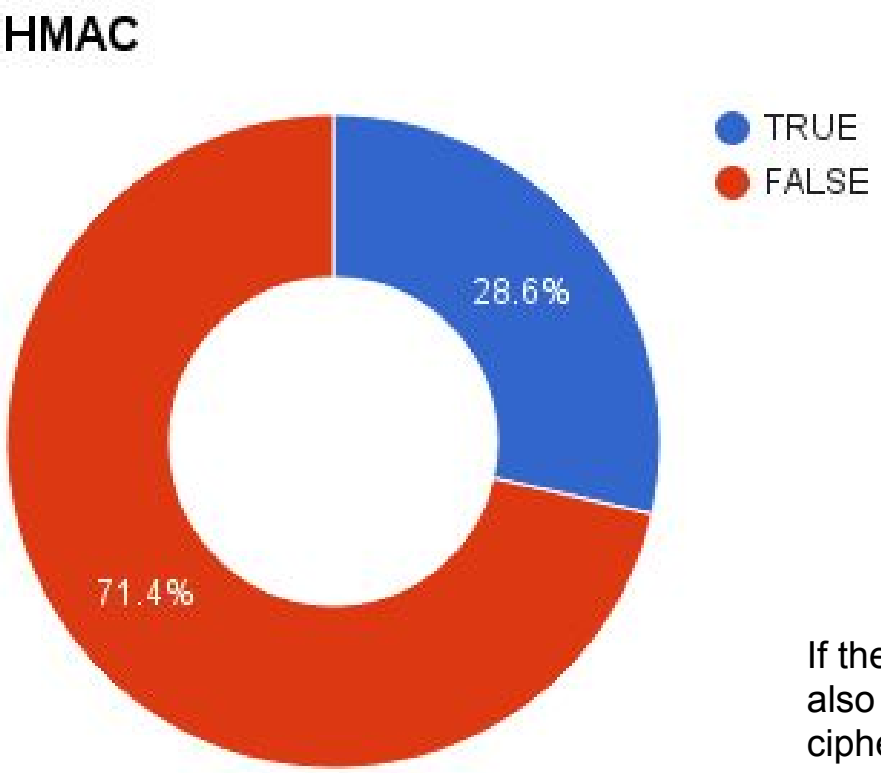
- HMAC construction resists preimage attacks even if the underlying hash is vulnerable to it
- MD5 might be broken, but HMAC-MD5 is still safe as far as we know
- Extends the useful life of your authenticator considerably
- However, preimage attacks aren't as relevant in SSO

HMAC: Explained

- K, a key
- m, a message
- H, a hash function
- K' = H(K) xor padding
- || means concatenate

$$HMAC(K, m) = H(K' || H(K' || m))$$

What percent actually used an HMAC?

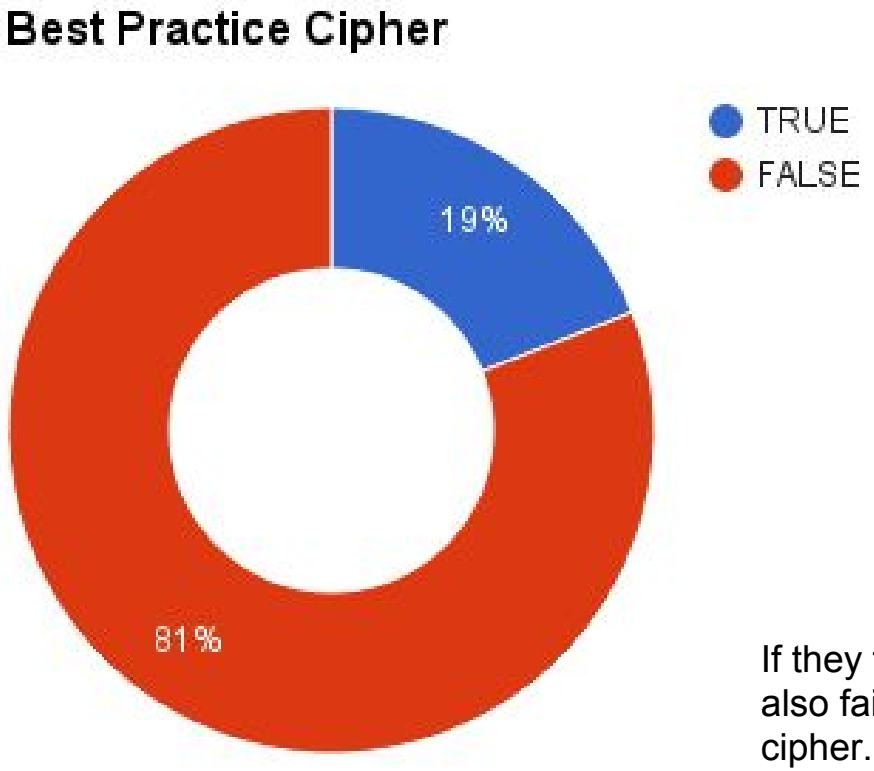


If they failed to use an HMAC, they also failed to use a best practice cipher.

Uses Obsolete Crypto Primitives

- Like MD5 or SHA-1
- Without HMAC, they're very weak
- Little to no excuse for using bare MD5 for anything anymore
- Best practice cipher
  - AES, in CBC, CTR, or GCM modes
  - SHA-256 or better
  - SHA-3

What percent used a best practice cipher?



If they failed to use an HMAC, they also failed to use a best practice cipher.

Short Keys

- getBytes doesn't do what you think it does

```
private static String getHMACHash(
    String name,
    String email,
    long time) throws Exception {
        byte[] keyBytes = sharedSecret.getBytes();
-       String movingFact =name + email + time;
+       String movingFact =name + sharedSecret + email + time;
        byte[] text = movingFact.getBytes();
```

Java getBytes

getBytes

```
public byte[] getBytes()
```

Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the default charset is unspecified. The `CharsetEncoder` class should be used when more control over the encoding process is required.

Returns:

`The resultant byte array`

Since:

`JDK1.1`

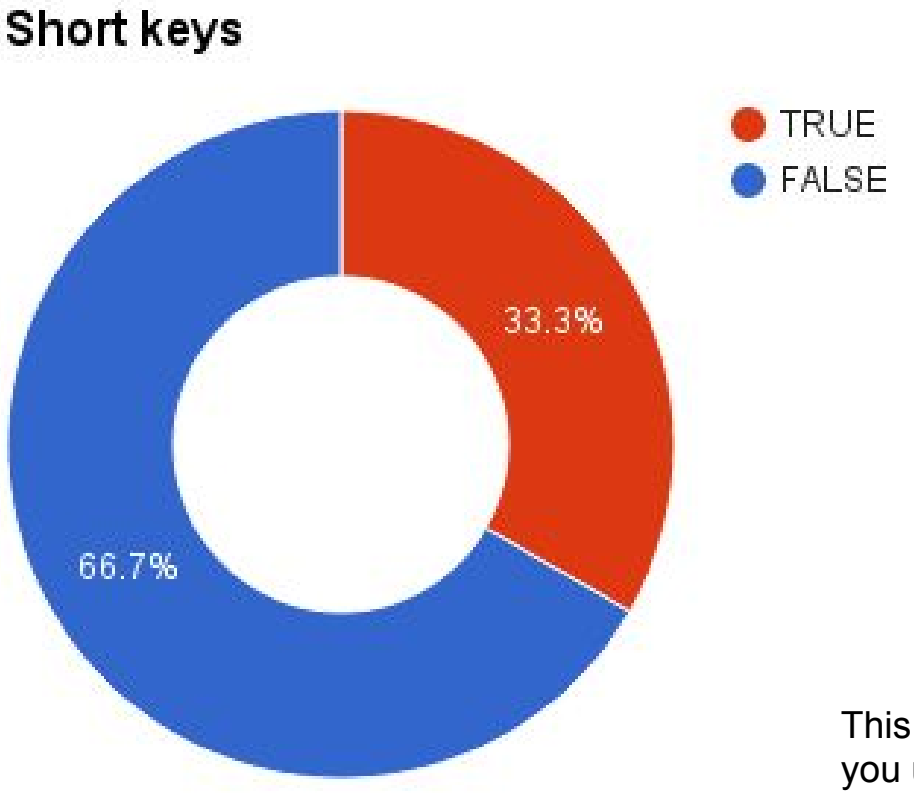
Java getBytes

- "AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB"
- 32 byte string
- `Hex.decodeHex(str) =`
  [0xAA, 0xAA, … , 0xBB, 0xBB] (16 bytes)
- `str.getBytes() =`
  [0x41, 0x41, …, 0x42, 0x42] (32 bytes)
- If your method is expecting 16 bytes of key, it may truncate the latter, resulting in no "B"s
- As there's only 16 possible values for each character, you've drastically reduced the possible combinations

Let's do the math

- Normal secrets are 128-bits, or 2^128 possibilities
- New secrets are 16 choices per hex character (0-9A-F)
- Each character consumes 8-bits, though
- 128 bits / 8 bits = 16 characters
- So there's 16^16 choices, instead of 2^128
- 2^128 = 340282366920938463463374607431768211456
- 16^16 = 2^64 = 18446744073709551616
- This is 99.99999999999999994% fewer choices

What percent made that silly error?

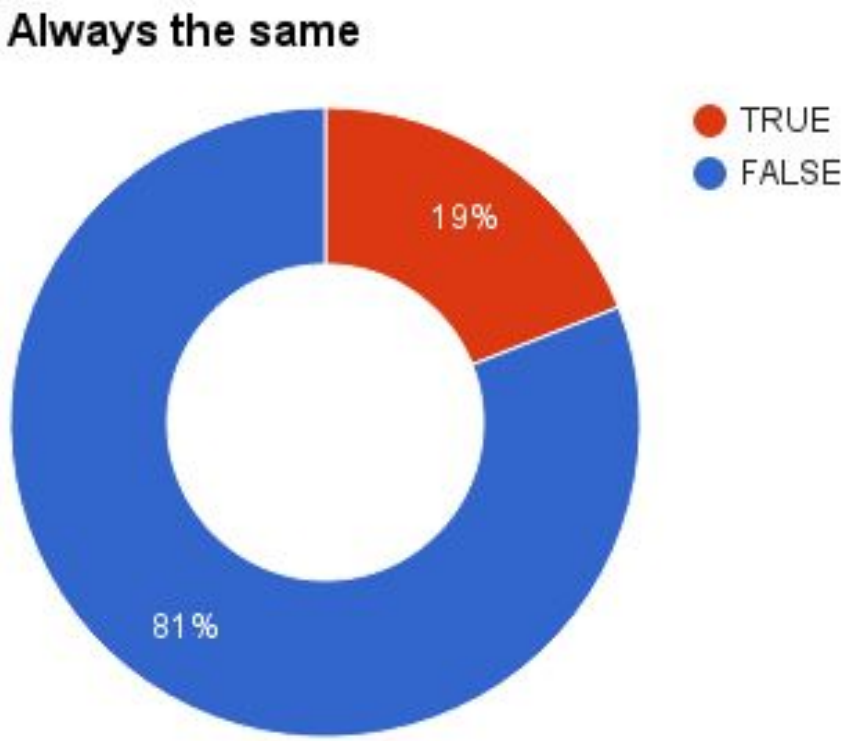


This isn't always a critical bug. If you use an HMAC, longer keys are supposed to be hashed.

Replay Attacks

- Since (potentially untrusted) users are being given the URLs to redirect to the SSO endpoint, it's critical that those URLs have a time component
- `/login?admin=false`**`&time=1474666031`**
- Or a nonce
- Otherwise users can just remember what their URL is and login forever, even after their account is deactivated on the other side

What percent kept their hash the same each time?



Static Initialization Vector

- Some just encrypt text with a shared secret key
- Using AES-128-CBC, for instance
- This mode requires an Initialization Vector (IV)
- Reusing the IV leaks information in CBC mode
  - Specifically, CBC has similar vulnerabilities to ECB mode when reusing IV

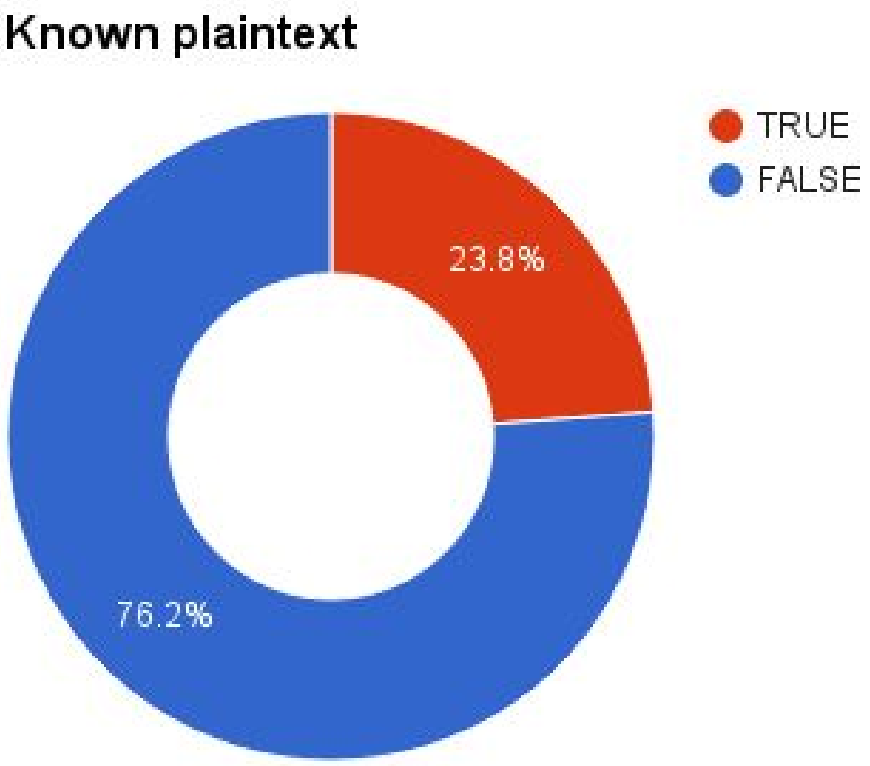
What percent used a static IV?

- Only one implementation used a block cipher
  - AES-128-CBC
-
-
-
-
-
-
-
-
-
- And they messed it up
- `byte[] INIT_VECTOR = "OpenSSL for Ruby";`

Known Plaintext

- In cryptography, it's best to limit what the attacker knows
  - Secret keys
  - Internal state
  - Plaintext
- Knowing or controlling the plaintext makes some attacks possible
  - Especially when not using an HMAC

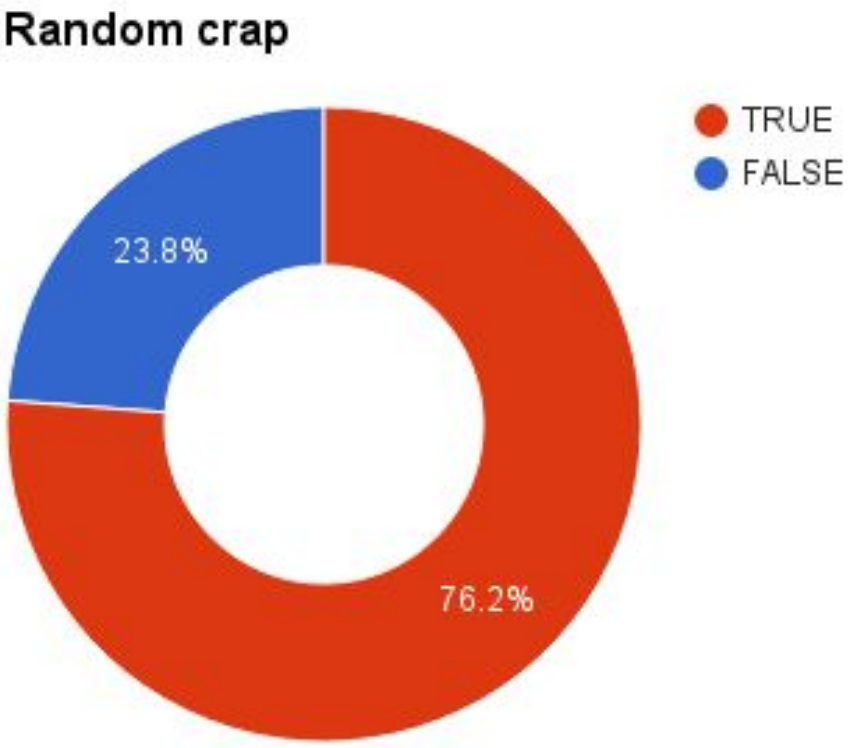
What percent of attackers know all the plaintext?



Random Crap

- Anything that doesn't add actual security, but makes some programmer feel better by twiddling bits
- Examples include
  - One's complement
  - XOR'ing random data against a constant
  - Reversing strings occasionally
  - md5(sha1(md5(sha1(key))))
- Demonstrates a fundamental misunderstanding how cryptography manipulates the data

How much random crap is there?



An Example of Artisan Bespoke Local Organic Cryptography

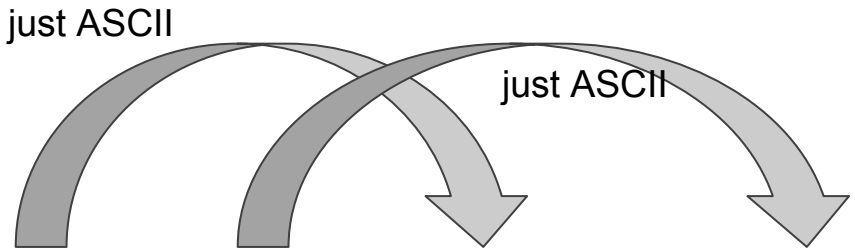
One implementation wrote their own cipher!

```
def encrypt(plaintext, input_key):
    key = hashlib.sha1(input_key).hexdigest()
    r = ''
    for idx, character in enumerate(plaintext):
        val = ord(character)
        adder = ord(key[idx % len(key)])
        r = r + base36encode(val + adder)[::-1]

    return r
```

```
print encrypt('0' * 40, 'hello')
```

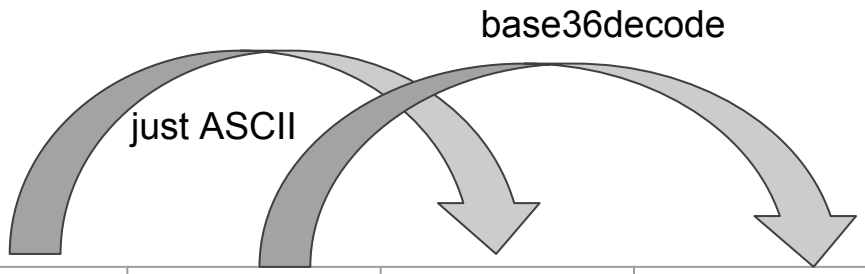
141464S234U2P244443434
T254W214Q2441424544454
O264R224S2W2Q23444X21
45414X2S2R2S244



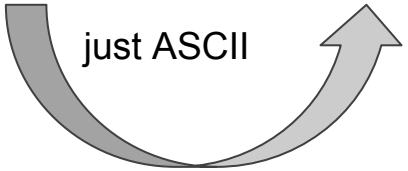
Plaintext	Key	Val	Adder	Addition	Base36
0	a	48	97	145	41
0	a	48	97	145	41
0	f	48	102	150	46
0	4	48	52	100	2S

Val + Adder = Addition

```
print get_key('0' * 40, '4141462S...')
```



Plaintext	Base36	Val	Decimal	Subtract	Key
0	41	48	145	97	a
0	41	48	145	97	a
0	46	48	150	102	f
0	2S	48	100	52	4



```
def get_key(plaintext, ciphertext):
    if len(ciphertext) != 2 * len(plaintext):
        return None
    key = ''
    for i, character in enumerate(plaintext):
        base36 = ciphertext[i * 2 : i * 2 + 2][::-1]
        value = int(base36, 36)
        key = key + chr(value - ord(character))

    return key
```

What went wrong?

- Ciphertext is strongly nonrandom

```
141464S234U2P244443434T254W214Q24414245444540264R224S2
W2Q23444X2145414X2S2R2S244
```

- Each input character always results in two output characters
- Key stream has a very short period
- No key schedule (makes it easy to get key out)
- Uses ASCII values of key rather than hex values
- Doesn't use HMAC or any authentication at all
- Base 36? Uses addition?

What went wrong?

- No avalanche effect

Input	Output
00000….0	141464S234...
10000….0	241464S234...
11000….0	242464S234...

What went wrong?

- No avalanche effect
- Ideally, each bit of change in the input has a 50% chance of changing each output bit

Input	**Ideal Output**
00000….0	eIXmxUXTOi...
10000….0	ihjohK5dTN...
11000….0	u4czK0YxHg...

What went wrong?

- Basically everything

Implications for the application

- One attacker can, with one use of SSO, get the shared secret key by knowing the plaintext and ciphertext
- The attacker can then authenticate as anyone, since they have the secret key
- It's a classic privilege escalation attack, done over SSO

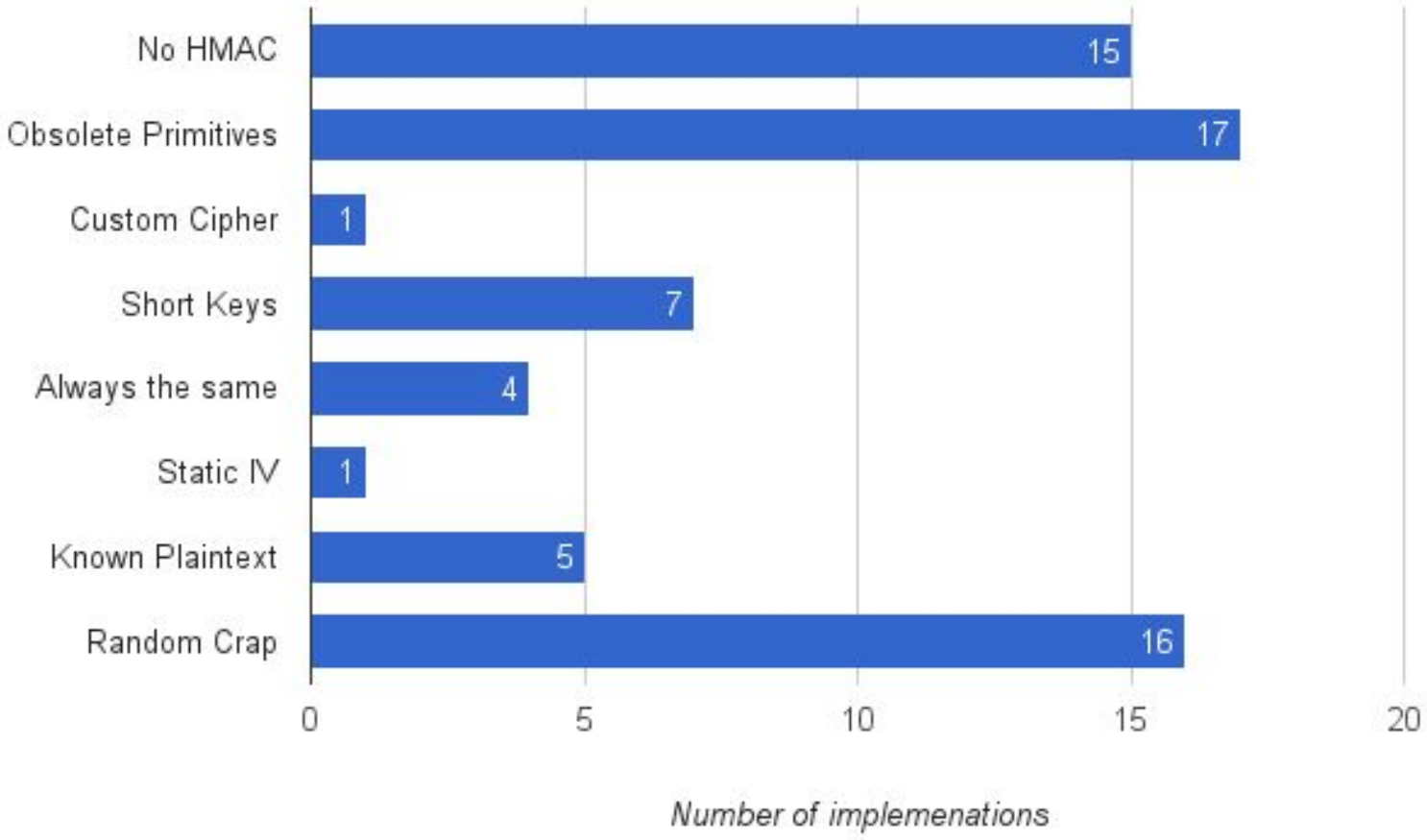
Should you roll your own crypto?

- We now have empirical evidence to say:

  No.

Results

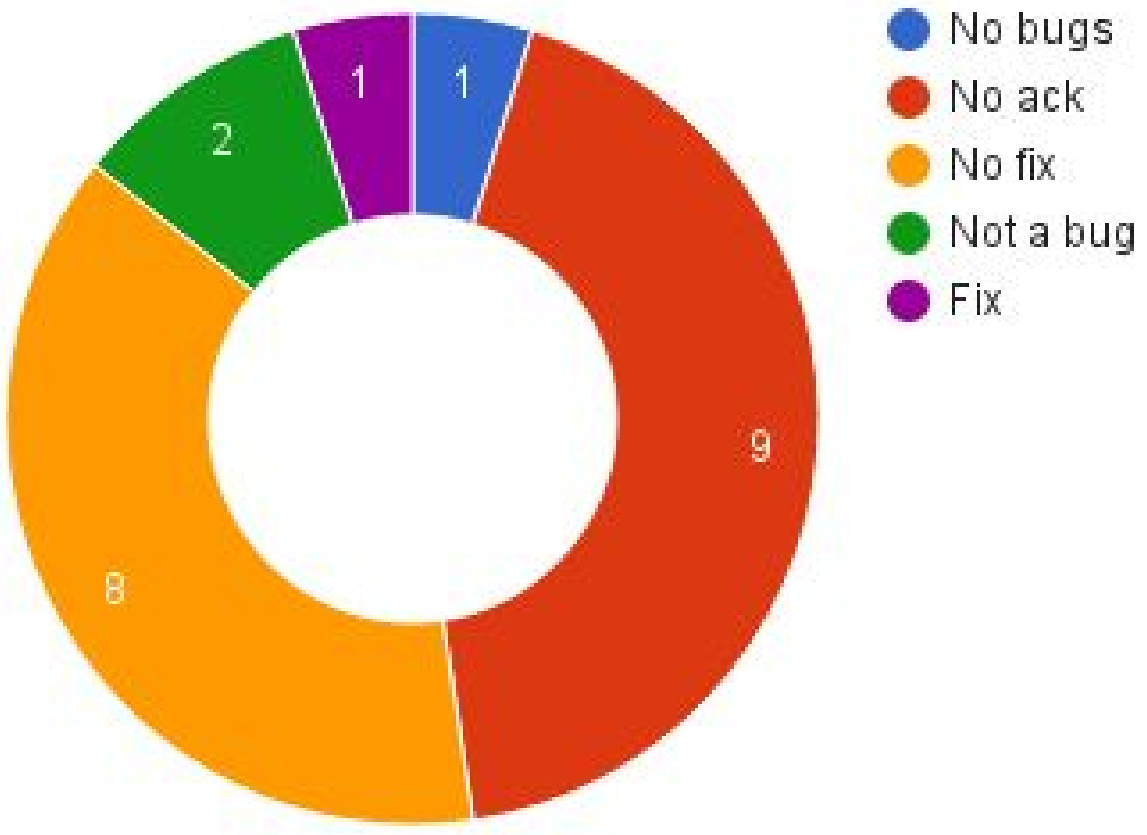
Overall Results



Vendor Response

- Only one vendor has fixed the bugs I've reported
  - UserEcho, customer service software
- Two have claimed that they're not bugs at all
- About half never even acknowledged
- Some of them were "unsupported" optional plugins, so they weren't a priority to fix
- Think about this when you're evaluating software that has some kind of SSO

Vendor Response



Custom SSO:
The "Right" Way

Custom SSO: The Right Way

- First, decide if you really need Custom SSO
- OAuth2 or SAML 2.0 is a way better choice
- Can't use OAuth2 or SAML?
  - Closely examine why
  - This is a critical part of your application

Custom SSO: The Right Way

- If you must have custom SSO, do something like this:
- Use HMAC-SHA-256 or HMAC-SHA-3
- Use 256-bits of secret
  - Generated using /dev/urandom
  - Triple check to make sure you're actually using it
- Hash several pieces of data, including some that the user does not control, like a UUID
- Use a nonce or timestamp
  - Check the nonce and timestamp
- Rate limit requests
- Have a security consultant review implementation and code

Dumb ideas for your crypto

- Use weird block cipher modes
  - Like Telegram's IGE
- Create impossible to win "contests"
  - Like Telegram's contest
- Military grade bank-trusted 1,000,000-bit garbage
- Use a bad random number generator
- Make your crypto hard to get right
- Ignore everything other people have learned about crypto

Abstinence-only crypto education doesn't work

Why did these companies make these mistakes?

- It's not because
  - They're dumb
  - They want to make insecure products
  - They are incompetent
- They're not cryptographers
- and they shouldn't have to be

Why did these companies make these mistakes?

- We (the security community) have let them down
  - We shame people who try to learn crypto
  - We discourage any solution but the "standard"
  - Sometimes, OAuth2 and others are too complex
  - There needs to be simple, copyable crypto solutions for these cases
  - We need crypto APIs that are hard to use wrong
- We should encourage people to learn crypto
- They're going to write it regardless


crypto education

Cryptography is different

Cryptography is different

- Cryptography isn't something you can iterate on until you get it right
  - because you'll never know if you do.
- Get it right
  - Ask for help
  - Don't let users use your product until it is vetted
- You are no longer shooting yourself in the foot if you mess up
  - You're hurting other people
  - And you'll be responsible

Cryptography is awesome!

- It's hard, but don't let that stop you!
  - Crypto is fun and rewarding
  - More people should learn crypto
- We know that people will implement crypto
  - Regardless of whether or not they know it
  - So you might as well learn it
- Be welcoming!
  - It's easy to be critical
  - It's hard to give constructive criticism and recommend improvements

Resources for learning cryptography

- Courses
  - Cryptography I at Coursera
  - Your local university's cryptography course
- The building blocks
  - Cryptography Engineering by Ferguson et. al
- Learn to break it
  - Cryptopals Crypto Challenges
- Learn by imitation
  - Look at pre-existing solutions
  - AWS Authentication, OAuth2, Double Ratchet

Thanks!

Contact info:

Twitter: @sean_a_cassidy
Email: sean@defensestorm.com
Website: https://www.seancassidy.me