CS 4530: Fundamentals of Software Engineering
Module 6, Lesson 1
Security

Rob Simmons

Khoury College of Computer Sciences

Warmup: what’s wrong with how security.town does passwords?

Warmup:
what should we do about passwords?

- Common responses: make sure passwords that we store stay on the server
- But we also maybe **shouldn't have the passwords**
  - That may mean we *give someone else the passwords* and never keep them on our server.
  - That may also mean we *store something that's not the password.*

Cryptographic Primitive #1: Hash

square(-4.5) = 20.25

- Exactly two real numbers could have produced that output
- You can easily find the other one
- You can easily find *x* such that square(*x*) = 2**8**.25 easily

sha256("password") = 5e884898da28047151d0e56f8dc6292773603d0d6aa…

- (Probably) *infinitely many* strings could have produced that output
- You *cannot* easily find another one
- You *cannot* easily find *x* such that sha256(*x*) = 5e**9**84898da2804715…

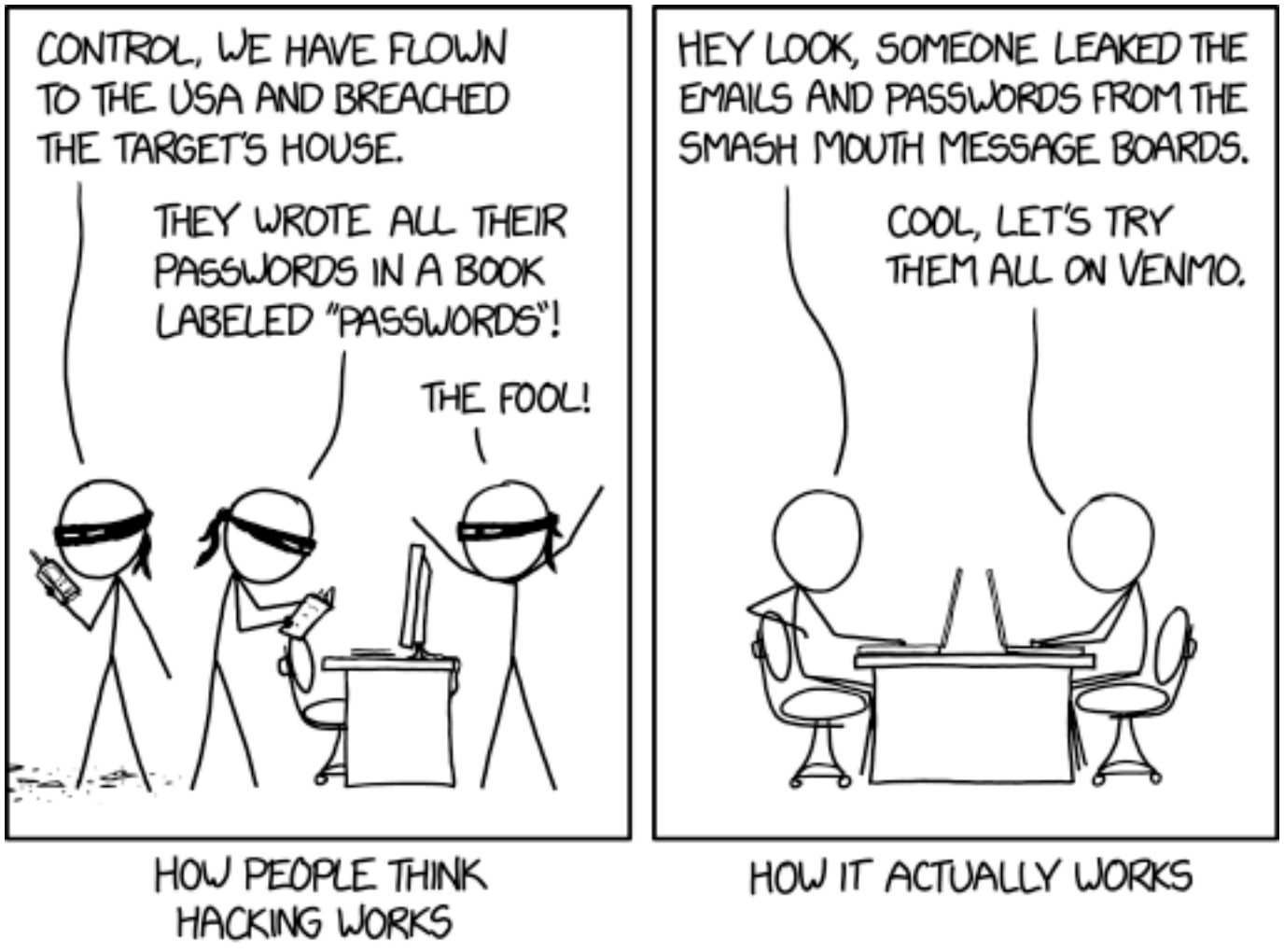
Cryptographic Primitive #1: Hash

- There's like, a couple dozen good cryptographic hashes out there. You should know about a like, six:
  - MD4 — run screaming from anyone who uses this for anything besides, like, a hashtable
  - MD5, SHA-0 — don't use these
  - SHA-1 — some issues for some applications, but like, git works because this is fine
  - SHA-256 — Bitcoin does its thing because this is fine
  - SHA-most-anything-else — fine-to-overkill

Warmup:
what should we do about passwords?

- Common responses:
- But we also **don't have to store the passwords**
  - That may mean we *give someone else the passwords* and never keep them on our server.
  - That may also mean we *store something that's not the password.*
- Storing the sha256(password) means that even if someone gets everything on our server, they can't easily find anyone's password… right?

Warmup:
what should we do about passwords?



…also, let's compute the sha256 of all of them and add them to our rainbow table: people reuse passwords

https://xkcd.com/2176/

Warmup:
what should we do about passwords?

To check auth, check if sha256(salt + password) = hash

```
_id: ObjectId('680fd9ee2b6f1eb9eb1c8b40')
user : ObjectId('680fd9ed2b6f1eb9eb1c8b3e')
username : "vihaarreddyy"
salt : Binary.createFromBase64('xVI7JEOwmqhcktREj7ElQ96BRple7QRxxbLcRcuGUWH1eGQJB+vBaF7rlvtgSNamsAqY3JgTlYNw3tU+6c490ens9xFyds2o6p/+..
hash : Binary.createFromBase64('jiSlBXys1P8HbuAAbgv7O/k54e5Z7+2/z3+DFqrL+4y0w2W6FNStdHz076vxoGUPtqwsfpTEvL7ua6q4vaSo1g==', 0)
__v : 0
```

```
_id: ObjectId('68136f1ec5df6e25e2c0a328')
user : ObjectId('68136f1ec5df6e25e2c0a326')
username : "sockpuppet"
salt : Binary.createFromBase64('em30O0mP2c+eoUu1F0hm1D3VKaqavvqYRFRR4tVd91GDq9aXBVow6ywvJPGJViy8VosDdcKNnnfA3dhGob04A/YWEai8UrgAaQ88..
hash : Binary.createFromBase64('q8MRMfM5GXMMsNKfqB1LaVDj/30OebLUObaHTcfCJ4XRAOIrEgQ6XHvh+YX9edRs6EldUoi4kE67tO/gJtdHHw==', 0)
__v : 0
```

```
_id: ObjectId('6818e2e2c5df6e25e2c0a481')
user : ObjectId('6818e2e2c5df6e25e2c0a47f')
username : "Satan"
salt : Binary.createFromBase64('nLFMXtqIiZWl0w8xmjvkgJOSx7fgcOHJAOwbp7RzGNlvKbdn57AUAfQPGvY4Gy9ppZ1p5ZqIqUyUBBnCGfI6RK+oMKQRhfcbZ1sc..
hash : Binary.createFromBase64('ae/AVfEwF7CilwREo5qW6KuzltTgpCPswugOW9S9GtxK4qcB5k/g9fcfHz97yqjpGAwk2tY3YkXumJ/jFIXYhA==', 0)
__v : 0
```

Security Principle #1: Use an established solution

Special case of this principle is “never roll your own crypto”

```
import { scrypt } from 'node:crypto';
```

```
(alias) function scrypt(password: BinaryLike, salt: BinaryLike, keylen: number, callback: (err: Error | null, derivedKey: Buffer) => void): void (+1 overload)
import scrypt
```

Provides an asynchronous scrypt implementation. Scrypt is a password-based key derivation function that is designed to be expensive computationally and memory-wise in order to make brute-force attacks unrewarding.

The `salt` should be as unique as possible. It is recommended that a salt is random and at least 16 bytes long. See NIST SP 800-132 for details.

When passing strings for `password` or `salt`, please consider `caveats when using strings as inputs to cryptographic APIs`.

The `callback` function is called with two arguments: `err` and `derivedKey`. `err` is an exception object when key derivation fails, otherwise `err` is `null`. `derivedKey` is passed to the callback as a `Buffer`.

An exception is thrown when any of the input arguments specify invalid values or types.

Security is all over the SE map

Learning Objectives for this Lesson

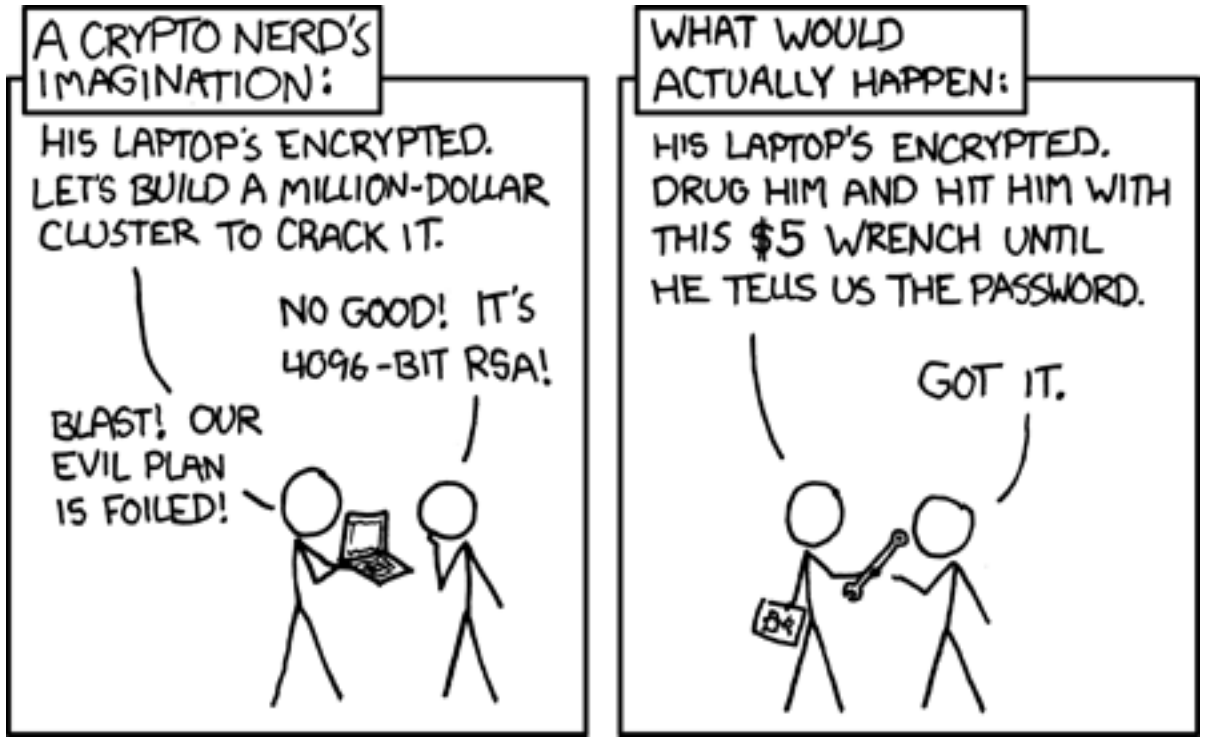
- By the end of this lesson, you should be able to:
  - Explain why you should always hash and salt your passwords
  - Have basic literacy in some key cryptographic primitives (hashes, message authentication codes, and encryption)
  - Define key terms (attack surface, threat model) relating to software/system security
  - Explain why all aspects of software engineering are necessary to think about in order to think about security

Security is a multiplayer game

The ***threat model*** specifies the rules you imagine that the other person is playing by

- Strategy.town has radically different applicable threat models than, say, Signal

The ***attack surface*** specifies where the other player gets to play the game



https://xkcd.com/538/

Security is a multiplayer game

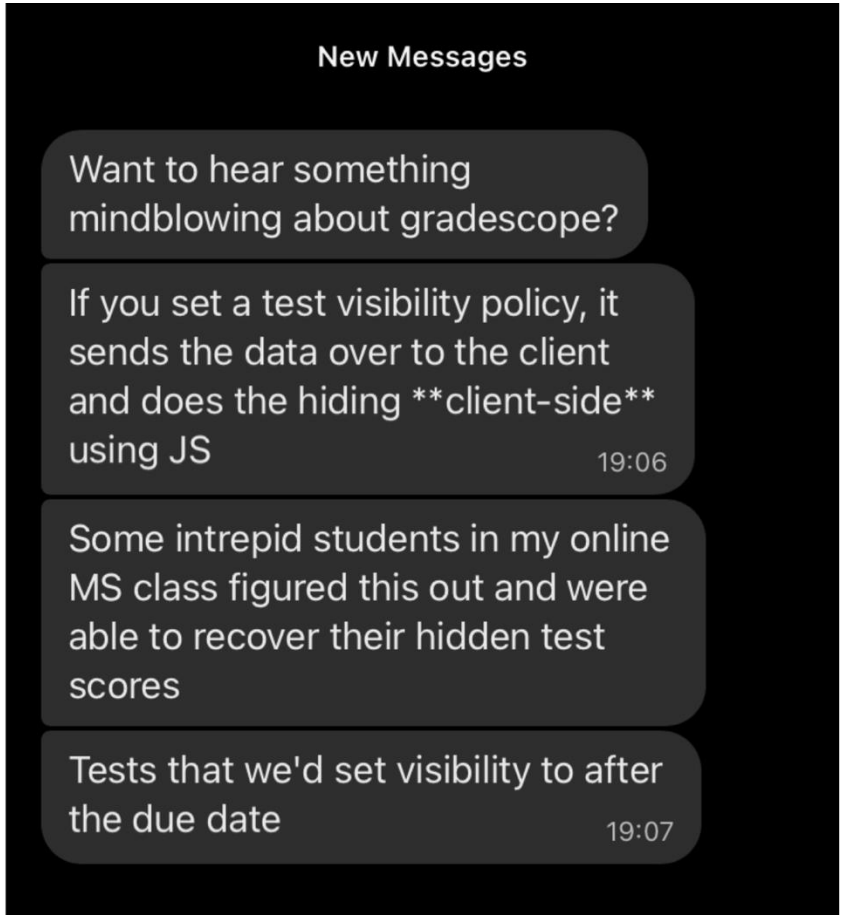
Simple threat model and attack surface: on the web, an attacker can make arbitrary requests to any API endpoints and send arbitrary messages over websockets, and can directly interpret every message that comes from the server.

Obvious consequences:

- Don’t put secrets in a game’s View type (the fact that it’s not shown in React isn’t sufficient!)
- Validate that it’s your turn on the backend (disabling the “make move button” in React isn’t sufficient!)

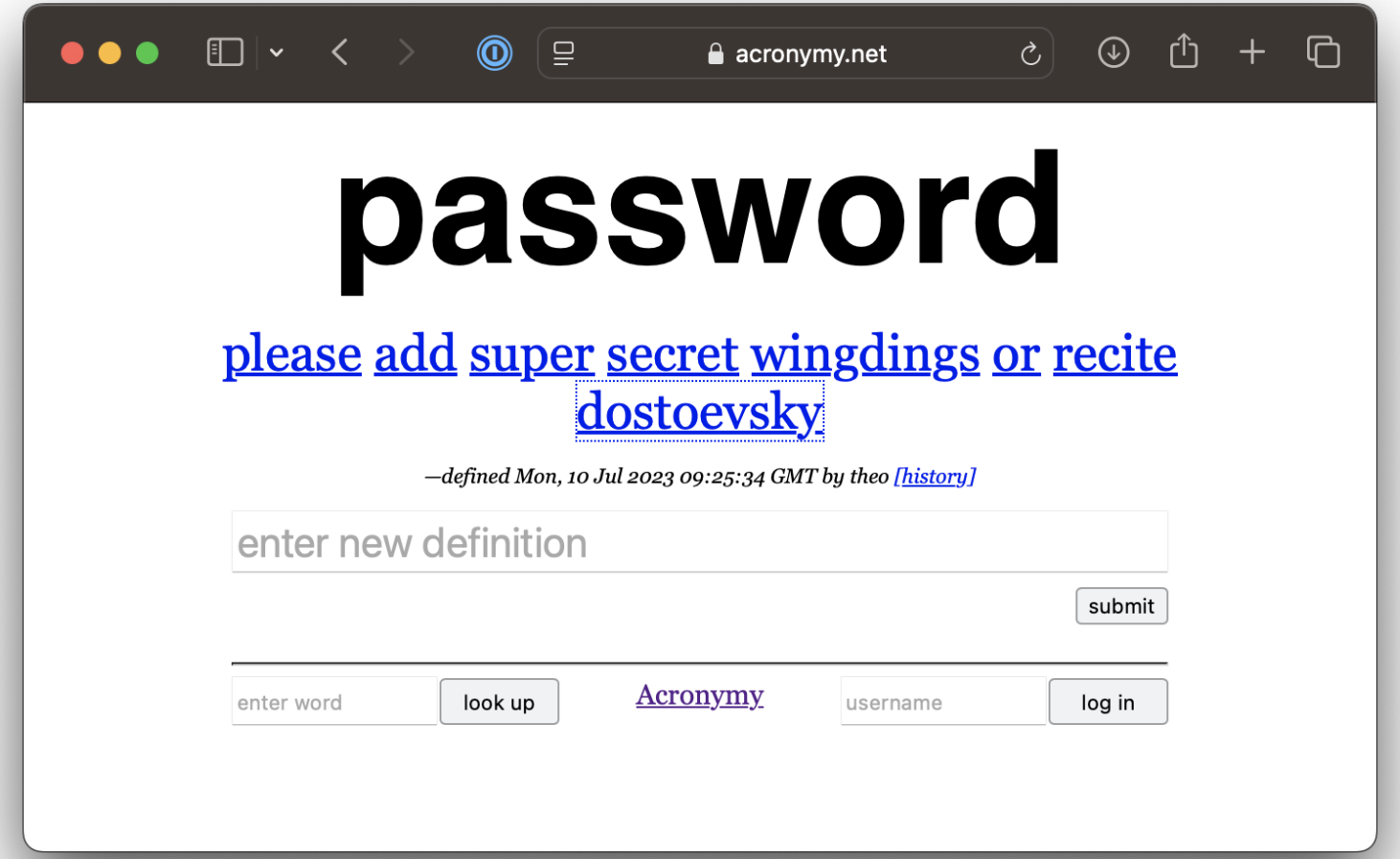
Security is a multiplayer game

Obvious consequences frequently are not:



Security is a multiplayer game

Keep an eye on which game you are playing



Security Principle #2: Watch for new attack surfaces

"Reverse shells" make a new attack surface

If your school uses Gradescope autograder, hidden test cases are now a thing of the past.

FOR EDUCATIONAL PURPOSES ONLY. DO NOT CHEAT

Set up a listener on a publicly accessible server with `nc -lk 1234 -vvv`. Then submit code to an assignment that creates a reverse shell and connects to your listener. This will allow you to type Linux commands into your listener and they will execute on the autograder machine, returning the results to you. From here you can exfiltrate hidden test cases. Or mine Bitcoin or whatever, there's no time limit on autograders unless your professor manually wrote one in.

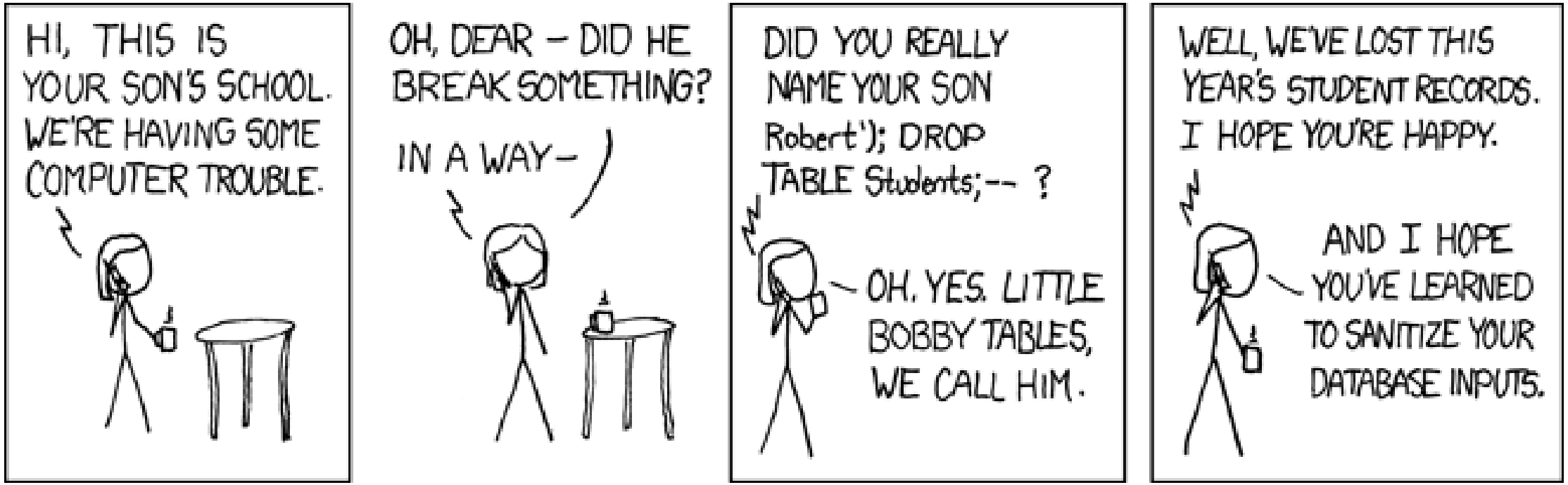
This can be as simple as copy-pasting some socket code, Google for "C reverse shell" or python or whatever your assignment uses. Put the IP of your public server into whatever shell code you end up using (make sure the port matches the listener, and you've allowed it thru the firewall if any). Unfortunately Gradescope's AG machines don't have `nc` on them and the old bash redirection to `/dev/tcp/IP address/1234` trick doesn't seem to work.

https://www.reddit.com/r/csMajors/comments/rlkf55/if_your_school_uses_gradescope_autograder_hidden/

Security Principle #2: Watch for new attack surfaces

Code and SQL injection is a new attack surface

Don’t roll your own solution — use established library! (Zod, Mongoose…)



https://xkcd.com/327/

Security Principle #2: Watch for new attack surfaces

Log4J could trigger arbitrary HTTP requests from places that weren't supposed to be able to make HTTP requests

Extremely Critical Log4J Vulnerability Leaves Much of the Internet at Risk

December 10, 2021 Ravie Lakshmanan

The Apache Software Fou
actively exploited zero-da
Apache Log4j Java-based
execute malicious code a
systems.

CVE-2021-44228 Detail

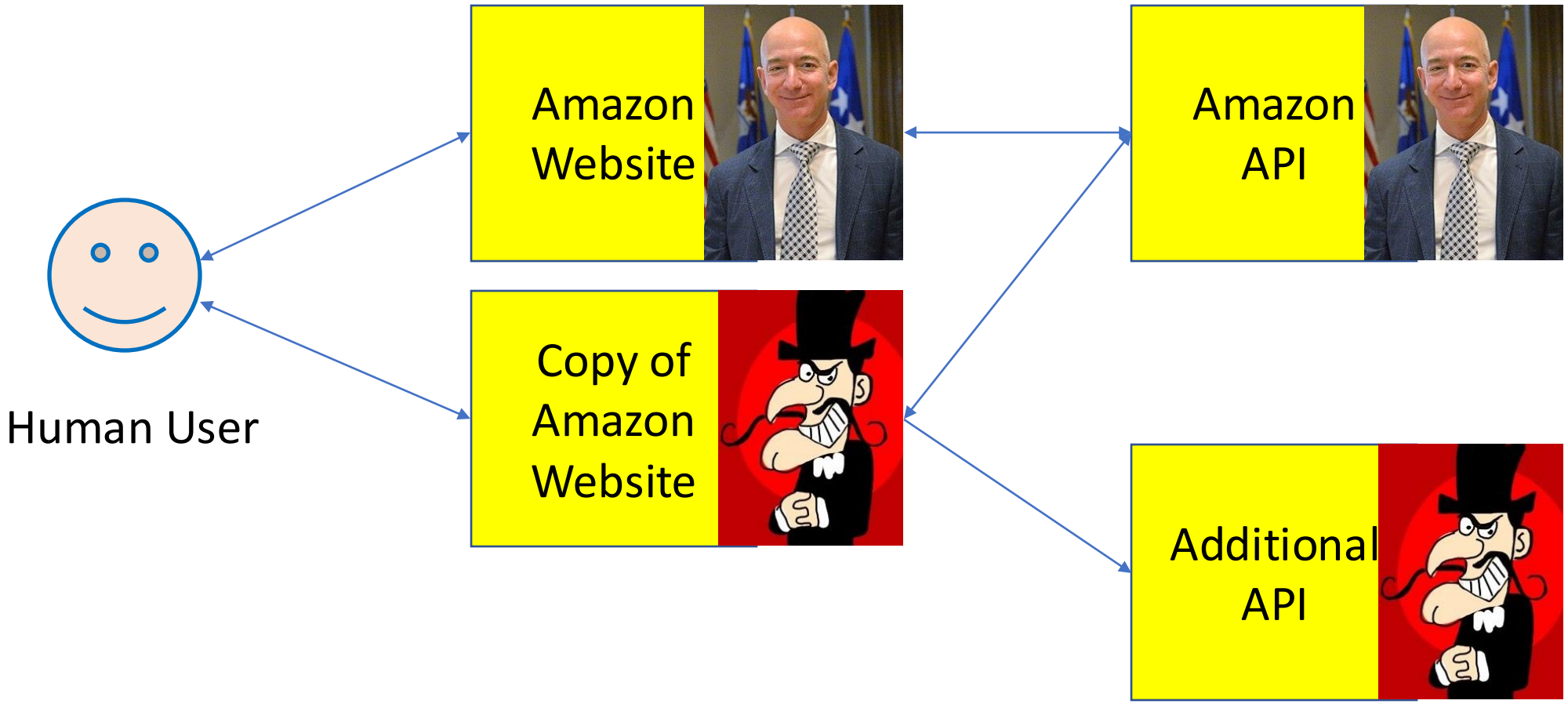
Current Description

Apache Log4j2 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) JNDI features used in configuration, log messages, and parameters do not protect against attacker controlled LDAP and other JNDI related endpoints. An attacker who can control log messages or log message parameters can execute arbitrary code loaded from LDAP servers when message lookup substitution is enabled. From log4j 2.15.0, this behavior has been disabled by default. From version 2.16.0 (along with 2.12.2, 2.12.3, and 2.3.1), this functionality has been completely removed. Notethat this vulnerability is specific to log4j-core and does not affect log4net, log4cxx, or other Apache Logging Services projects.
https://nvd.nist.gov/vuln/detail/CVE-2021-44228

https://thehackernews.com/2021/12/extremely-critical-log4j-vulnerability.html

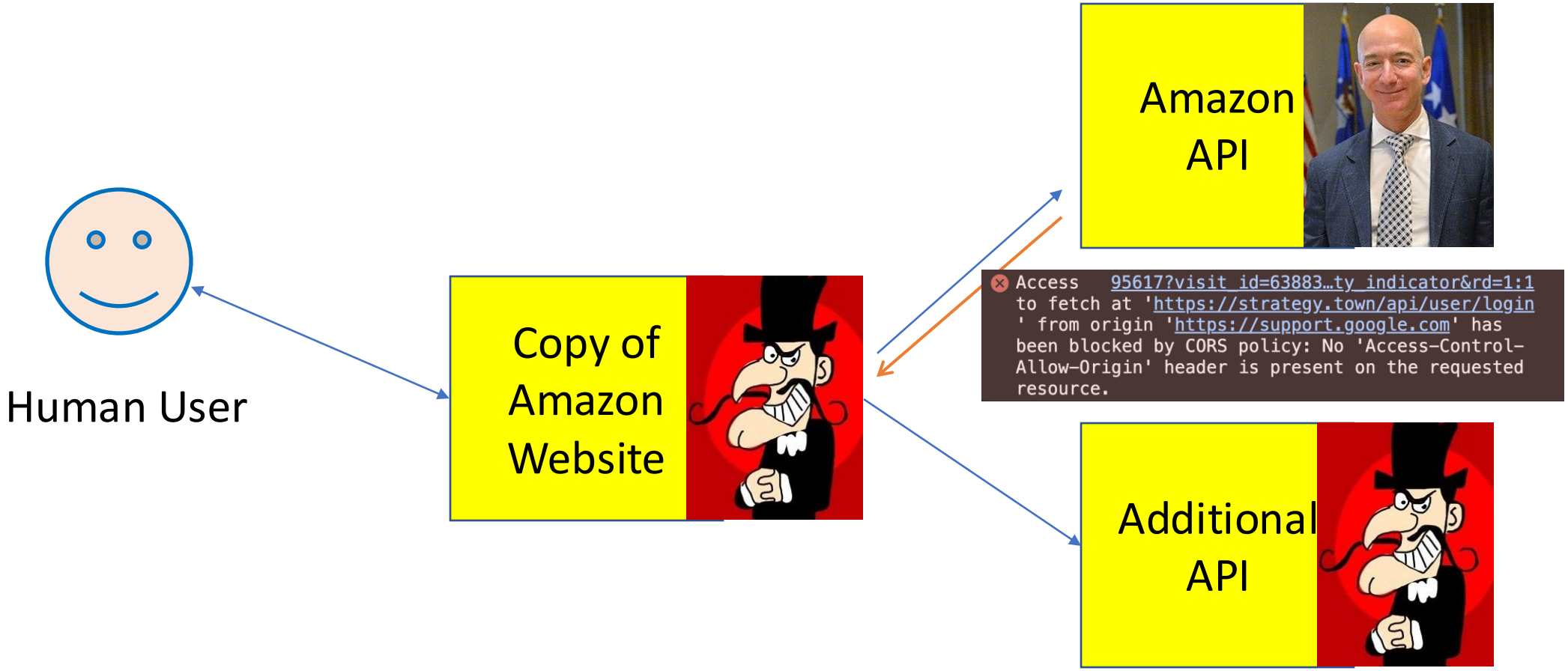
Security Principle #3: Beware the man in the middle

You and Amazon can't actually see each other. How do you know you're interacting with Amazon?



Security Principle #3: Beware the man in the middle

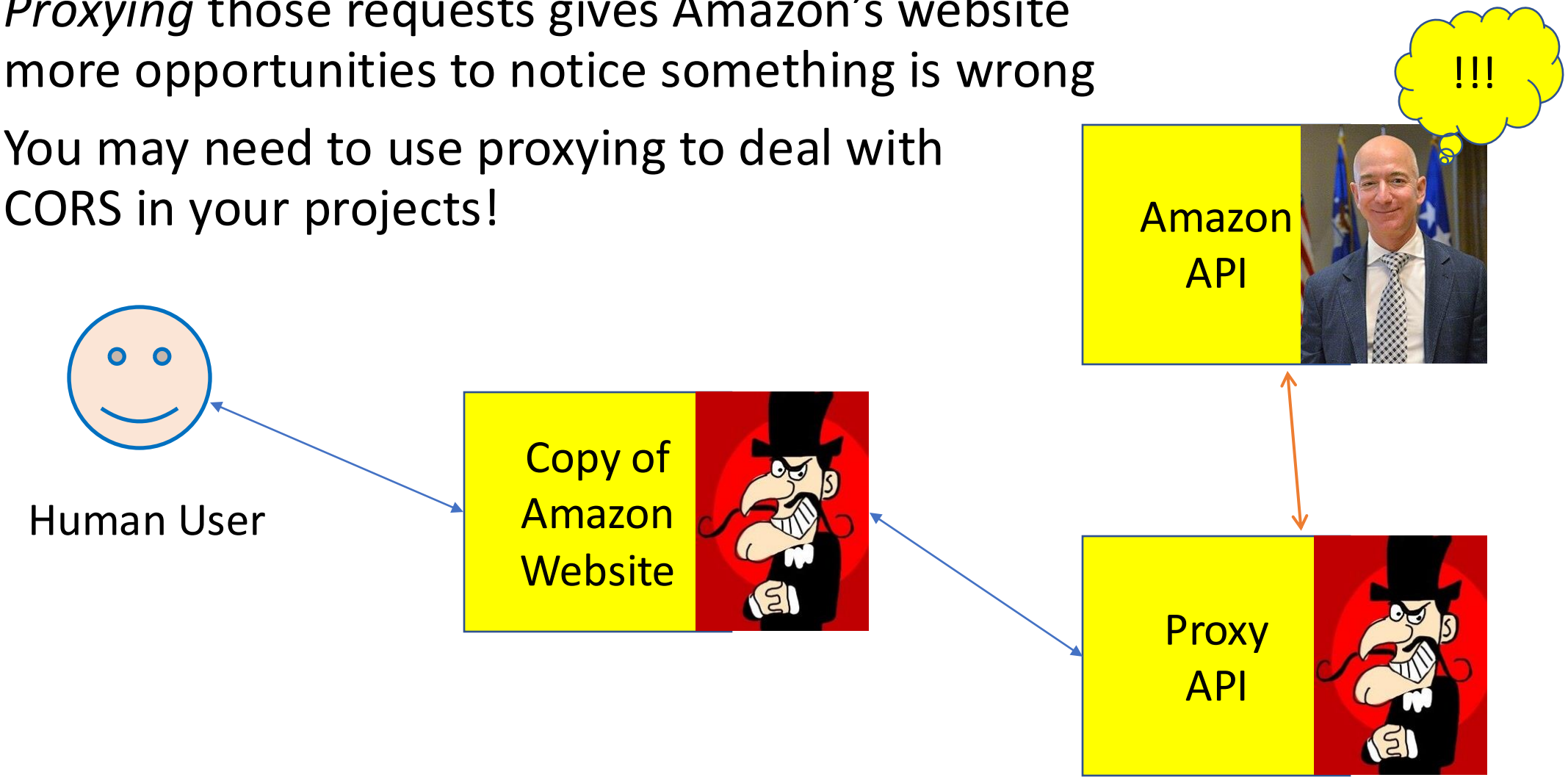
With CORS, Amazon's API tells the web browser:
ignore my response unless the user is on amazon.com



Security Principle #3: Beware the man in the middle

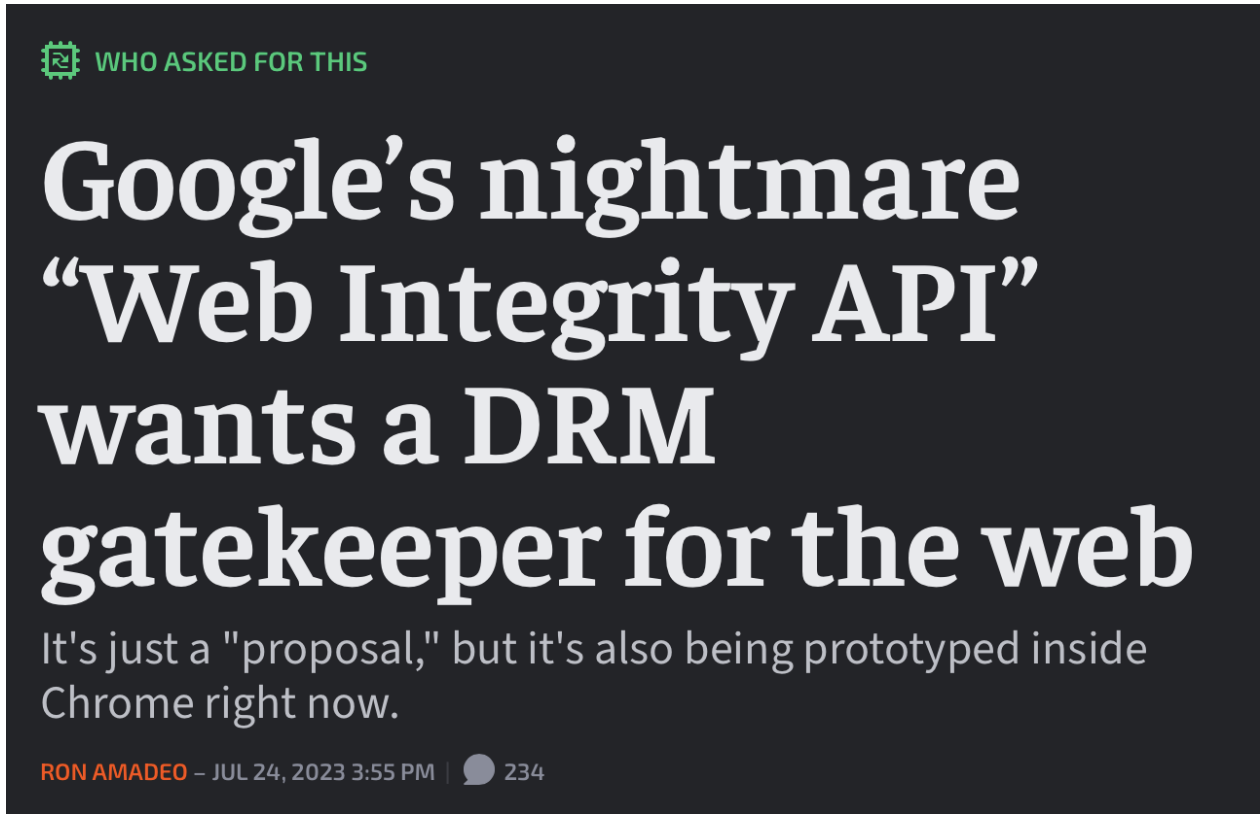
Proxying those requests gives Amazon's website more opportunities to notice something is wrong

You may need to use proxying to deal with CORS in your projects!



Browsers are *Fascinating*

- Influenced by browser makers, consumer choices, standards bodies, online businesses…
- CORS works because ~everyone would rather have CORS in their browser (but remember your threat model!)
- Browser monopolies arise and change the balance of power



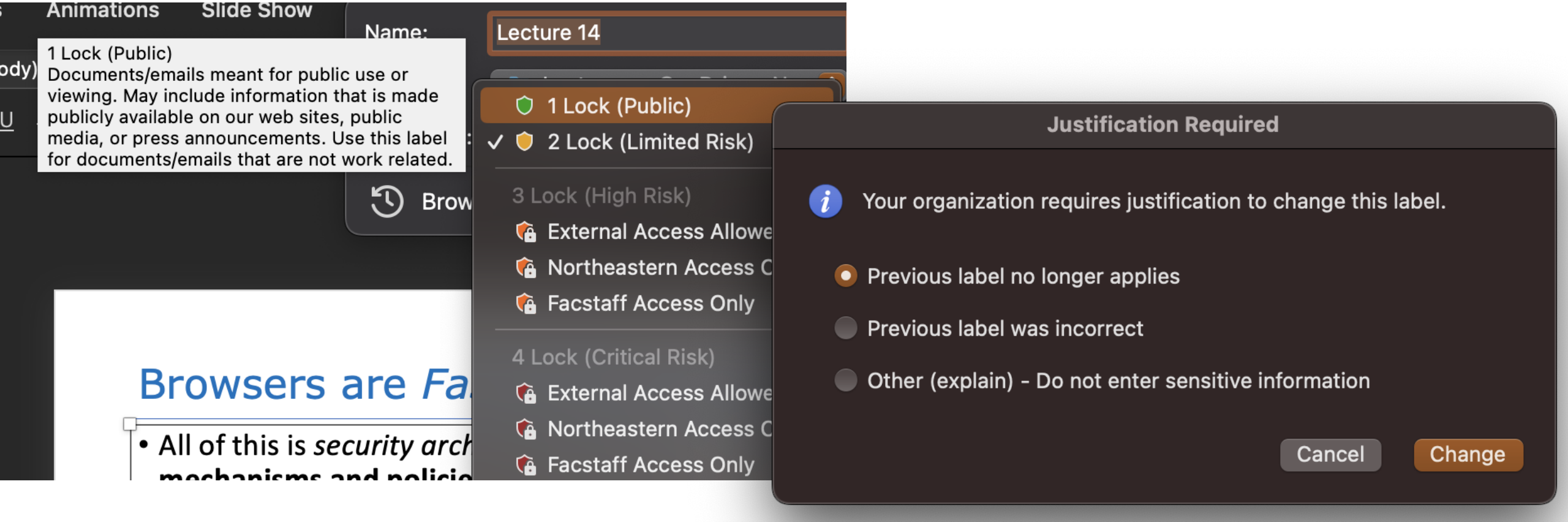
https://arstechnica.com/gadgets/2023/07/googles-web-integrity-api-sounds-like-drm-for-the-web/

Browsers are *Fascinating*: Cookies

- Cookies add *sessions* to HTTP requests — the login API end point can set a cookie and subsequent HTTP requests will send it back.
- If JavaScript has no business viewing the cookie, it can be an HTTP ONLY cookie — code can't see it
  - BUT THE USER CAN! (Remember your threat model!!!)
- Makes it unnecessary to send (and verify!) the password every time.
- Of course, it's also how Facebook knows what jobs you applied for…

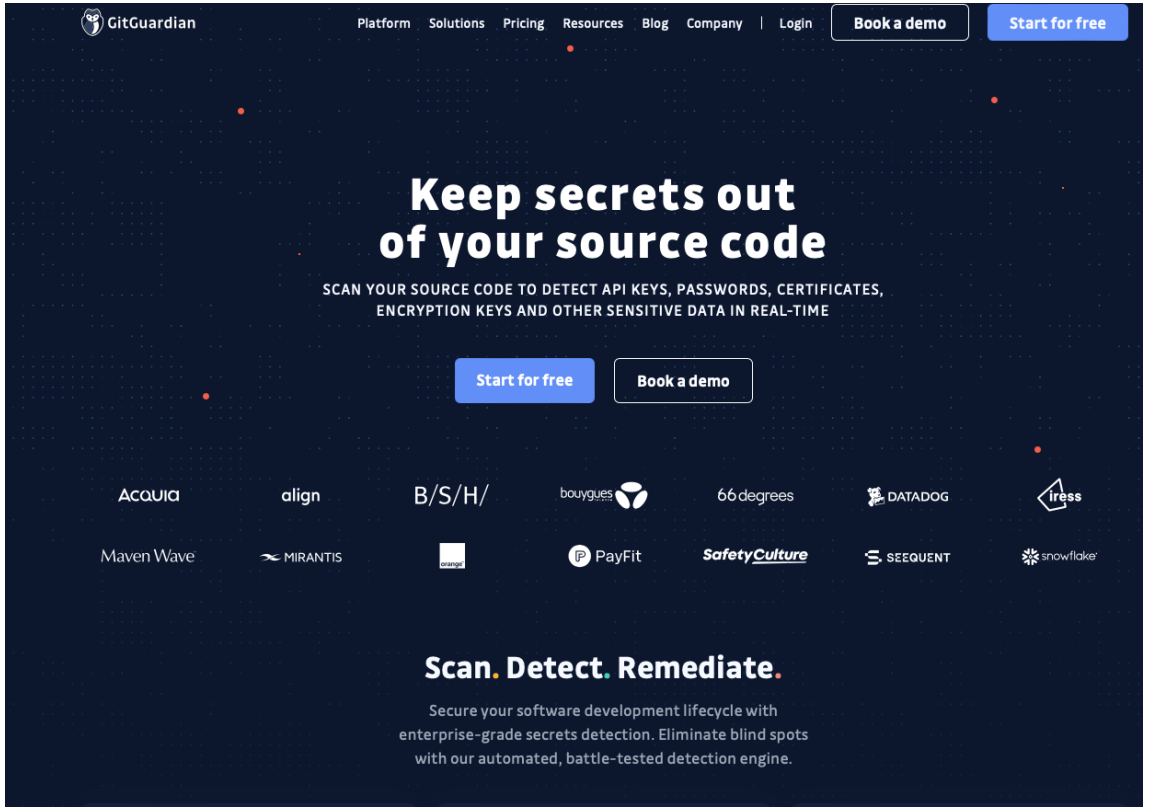
Security Architecture

- CORS and HTTP ONLY cookies are part of the *security architecture* of browsers— the **mechanisms and policies** that we build into our system to mitigate threats



Security Architecture and Security Culture

- Don’t check API keys (basically passwords someone else generates for you) into git, ever!
- Tools like *GitGuardian* automatically detect secrets in repositories



Security Architecture and Security Culture

- Industrial study of secret detection tool in a large software services company with over 1,000 developers, operating for over 10 years
- What do developers do when they get warnings of secrets in repository?
  - 49% remove the secrets; 51% bypass the warning
- Why do developers bypass warnings?
  - 44% report false positives, 6% are already exposed secrets, remaining are “development-related” reasons, e.g. “not a production credential” or “no significant security value”

Is this a problem?
Whose problem is it?

Cryptographic Primitive #2: Signing

- Your server can have a secret key (just a random one)
- If you compute sha256(secret + message) = hash, then give the message AND the hash to someone else, they can hand you back the message and the hash later, and you can believe "you" (someone who knew the secret) agreed to compute that hash — no one else could!
- The hash is an HMAC: a Hash Message Authentication Code
- Sign message "this is user2": that plus the HMAC is your cookie!

```
import { createHmac } from "node:crypto";
```

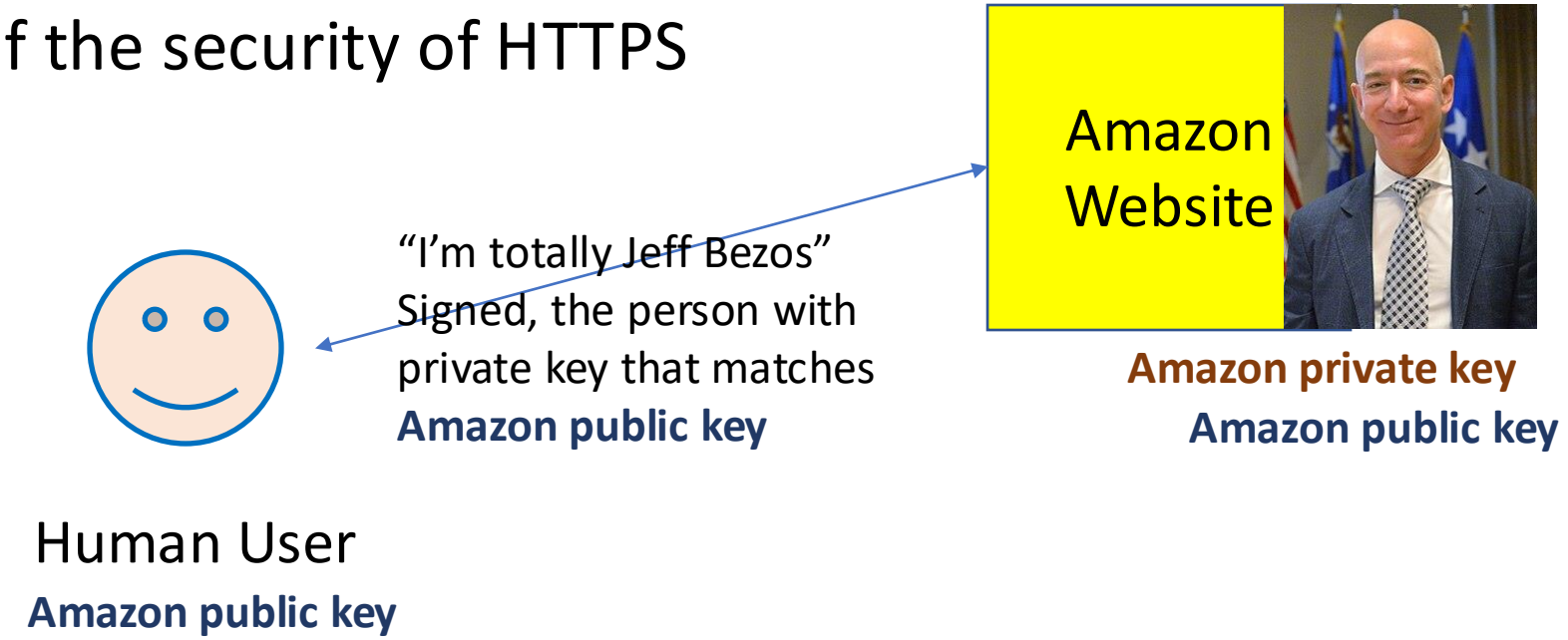
```
(alias) function createHmac(algorithm: string, key: BinaryLike | KeyObject, options?:
Stream.TransformOptions): Hmac
import createHmac
```

Creates and returns an `Hmac` object that uses the given `algorithm` and `key`. Optional `options` argument controls stream behavior.

The `algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha256'`, `'sha512'`, etc. On recent releases of OpenSSL, `openssl list -digest-algorithms` will display the available digest algorithms.

Cryptographic Primitive #2: Signing

- Public key encryption allows *asymmetric* signing
  - Paired **public key** and **private key**
  - Anyone with the **public key** can verify that a message was signed only by someone with the **private key**
- One basis of the security of HTTPS

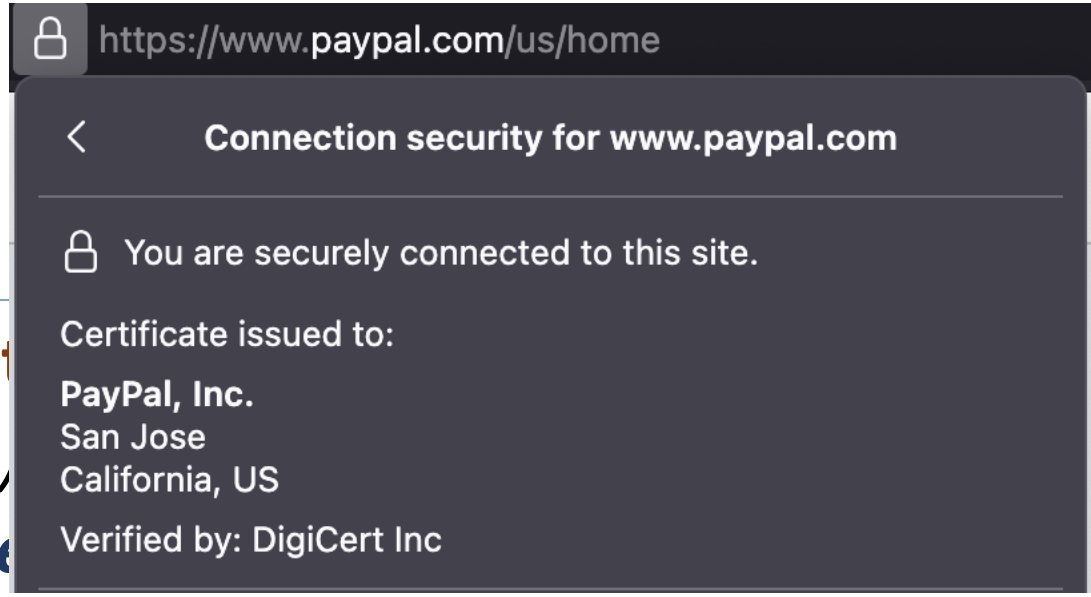


Security Principle #4: Chains and webs of trust

Your browser or computer shipped with some public keys:

- **Google Trust Services LLC (GTS Root R4) public key**
- **Internet Security Research Group (ISRG Root X1) public key**
- **DigiCert High Assurance EV Root CA public key**

Security Principle #4: Chains and webs of trust



DigiCert High Assurance EV Root CA privat

- Signs the message "*Whoever has the priv* ***DigiCert SHA2 Extended Validation Serve*** *legit and is a certificate authority*"

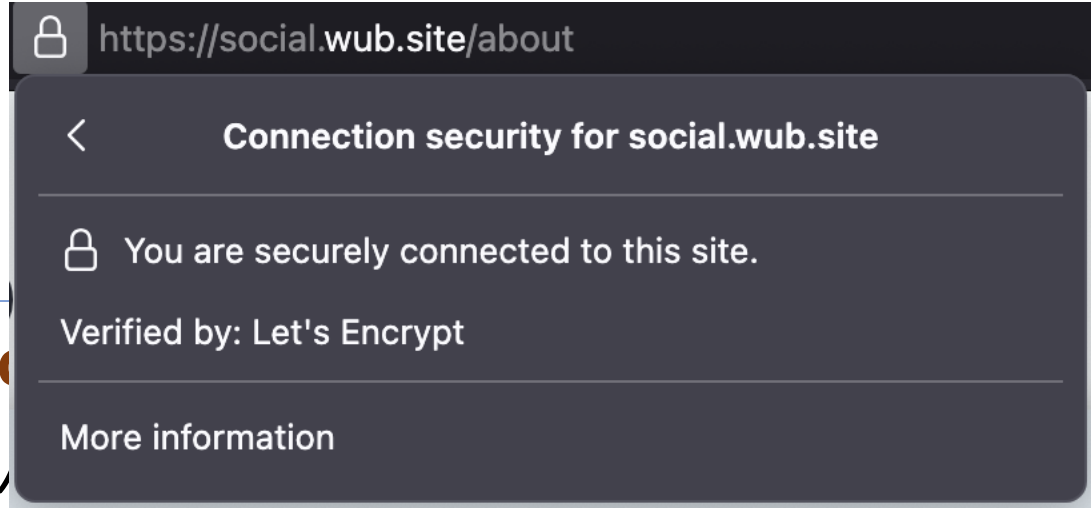
DigiCert SHA2 Extended Validation Server CA private key holder

- Signs the message *"Whoever has the private key matching* ***paypal.com public key*** *is* ***definitely associated with the legal entity PayPal.com****, but is not a certificate authority"*

www.paypal.com private key holder

- Signs the message *"Hi, if you think you're connecting to https://www.paypal.com/, would you like to give me your password?"*

Security Principle #4: Chains and webs of trust



Internet Security Research Group (ISRG R

- Signs the message "*Whoever has the priv* ***Encrypt (R10) private key*** *seems legit and is a certificate authority*"

Let's Encrypt (R10) private key holder

- Signs the message *"Whoever has the private key matching* ***social.wub.site public key*** *seems legit associated with that domain, but not as a certificate authority"*

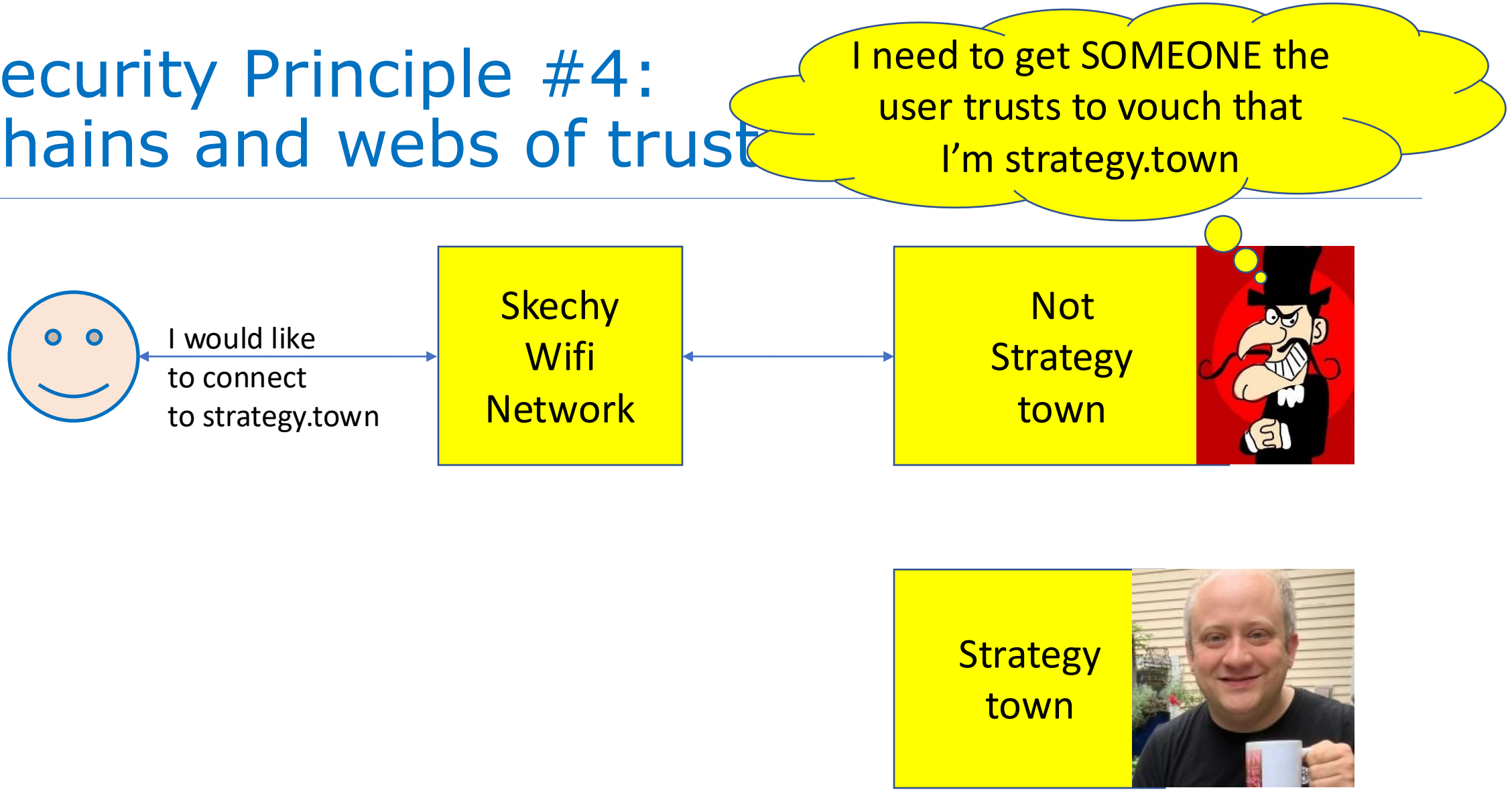
social.wub.site private key holder (that is, Rob)

- Signs the message *"Hi, if you think you're connecting to https://social.wub.site/, would you like to give me your password?"*

Security Principle #4: Chains and webs of trust

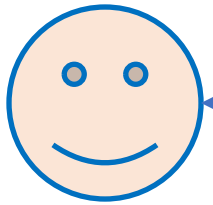
- You can do this on your own server for free
- This wasn't the case before the Let's Encrypt nonprofit!

Security Principle #4: Chains and webs of trust



Security Principle #4: Chains and webs of trust

I need to get SOMEONE the user trusts to vouch that I'm strategy.town



I would like to connect to strategy.town

Skechy Wifi Network

Not Strategy town

Let's Encrypt

Hopefully Less Sketchy Internet

Strategy town

Let's Encrypt is vouching for anyone IT SEES as the strategy.town owner

Security Principle #4: Chains and webs of trust

- Modern TLS/HTTPS security relies on the security of the *domain name system*
- The DNS system has its own security issues and threat models!

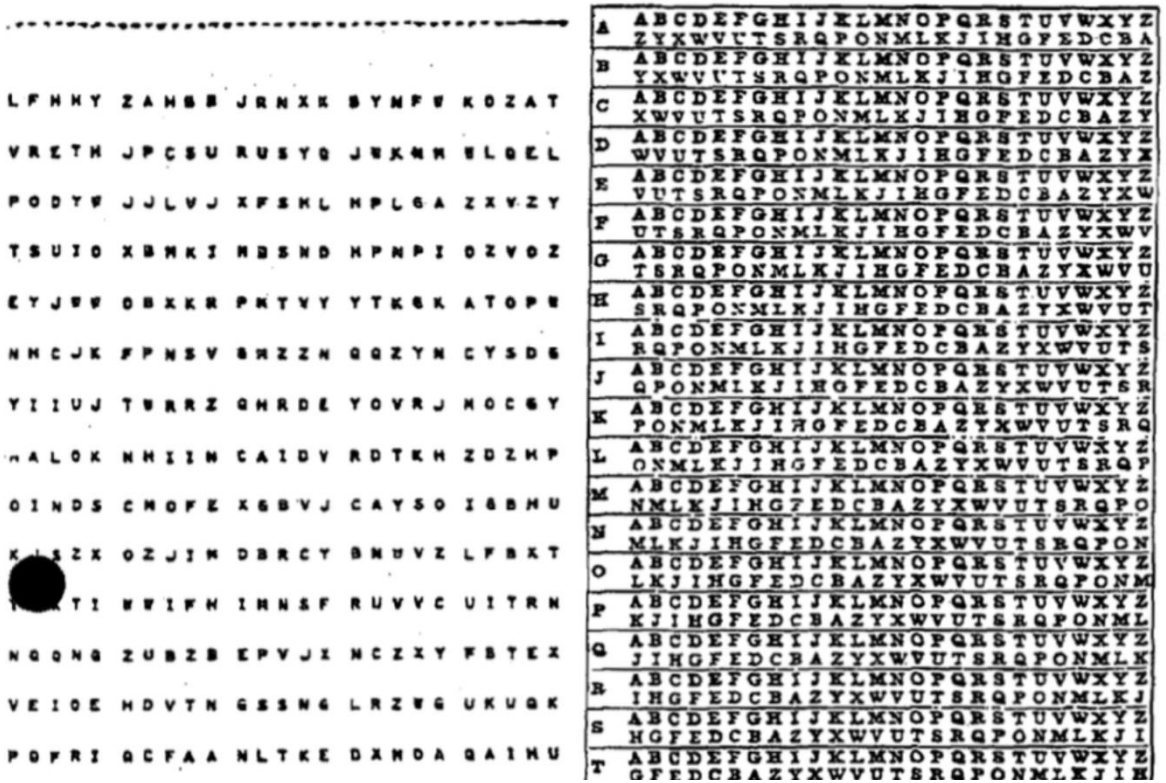
The .org controversy

In November 2019, the Internet Society (ISOC) announced its intention to sell Public Interest Registry (PIR) – the registry for the .org top-level domain – to investment firm Ethos Capital for US$1.135 billion. The announcement has generated controversy, with supporters and opponents of the deal strongly voicing their views.

https://dig.watch/trends/dotorg

Cryptographic Primitive #3: Shared secret encryption

- Absolutely perfect security via one-time pads
  - You and I both have n TRULY RANDOM bytes X
  - You send me X xor Message
  - I decode message by computing (X xor Message) xor X = Message
- ONLY FOR ONE TIME!!!



https://en.wikipedia.org/wiki/One-time_pad

Cryptographic Primitive #3: Shared secret encryption

- There are a couple of reasonably secure *symmetric encryption standards*
- Use a *small* shared secret to encrypt *lots* of data
- We *think* most of the ones we currently use are pretty secure, even against quantum computers
- Problem: how do share a secret with Amazon?

Human User

Amazon public key

Amazon private key

Amazon public key

Cryptographic Primitive #3: Shared secret encryption

- Problem: how do share a secret with Amazon?
- If two people know two public keys and either one of the corresponding private keys, they can do math to come up with a shared secret, and use that for symmetric encryption!
- Textbook ways of doing this are very vulnerable to quantum computers

Review of this whirlwind tour

Four big ideas Rob thought were worth emphasizing:

1. Use an established solution
2. Watch for new attack surfaces
3. Beware the man in the middle
4. It’s all chains and webs of trust

Three cryptographic primitives you should be aware of:

1. Hash
2. Signing/message authentication codes
3. Shared secret encryption

Learning Objectives for this Lesson

- Now that it's the end of the lesson, you should be able to:
  - Explain why you should always hash and salt your passwords
  - Have basic literacy in some key cryptographic primitives (hashes, message authentication codes, and encryption)
  - Define key terms (attack surface, threat model) relating to software/system security
  - Explain why all aspects of software engineering are necessary to think about in order to think about security