CS 4530: Fundamentals of Software Engineering

Module 13: Software Engineering & Security

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Module

- By the end of this module, you should be able to:
  - Define key terms relating to software/system security
  - Explain 5 common vulnerabilities in web applications and similar software systems, and describe some common mitigations for each of them.
  - Explain why software alone isn’t enough to assure security
  - Explain at least one way in which security can be incorporated into an agile development process

Outline of this lecture

1. Definition of key vocabulary
2. Some common vulnerabilities, and possible mitigations
3. Getting security right is about people as well as software.

Security: Basic Vocabulary (1)

- Security is a set of non-functional requirements (sometimes called “CIA”):
- Confidentiality: is information disclosed to unauthorized individuals?
- Integrity: is code or data tampered with?
- Availability: is the system accessible and usable?

Security: Basic Vocabulary (2)

- Asset: something of value that is the subject of a security requirement
- Threat: potential condition or event that could compromise a security requirement
- Vulnerability: a characteristic or flaw in system design or implementation, or in the security procedures, that, if exploited, could result in a security compromise

Security: Basic Vocabulary (3)

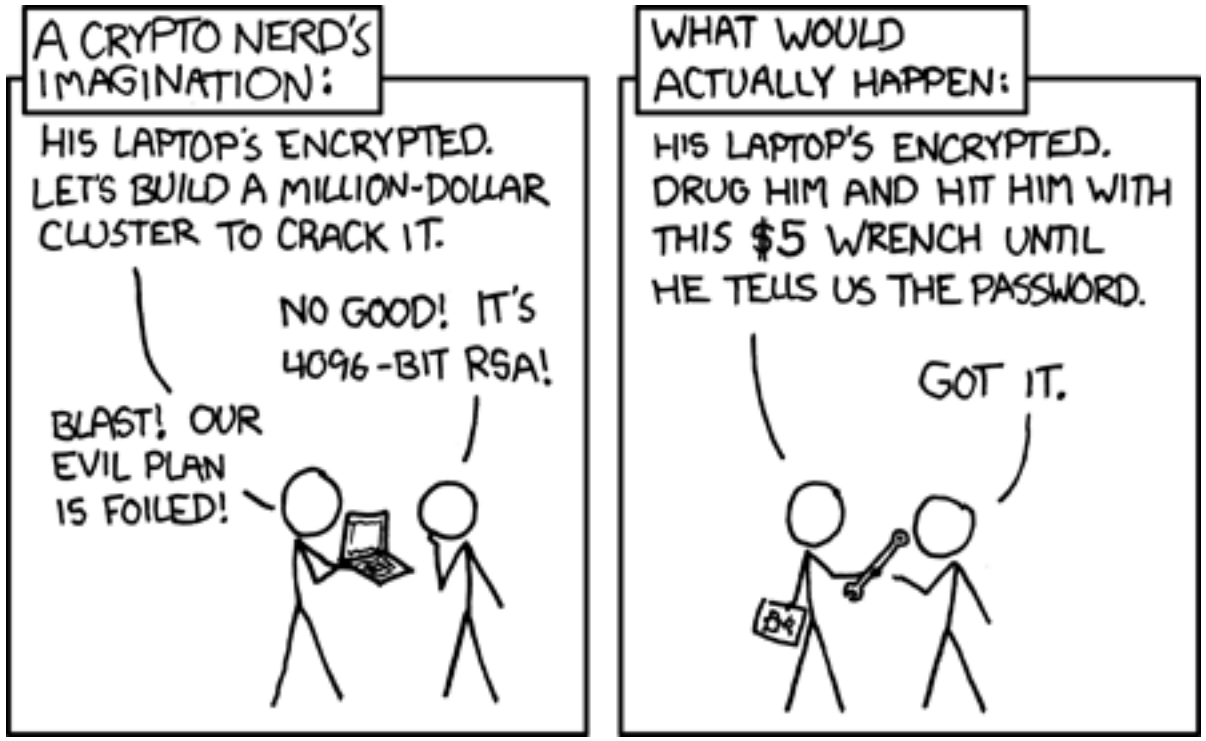
- Attack Surface: a portion of the system that might contain a vulnerability
- Exploit: a technique or method for exploiting a vulnerability
- Attack: an event that results in the realization of a threat
- Mitigation: a technique for making an attack less likely, more expensive, or less valuable to an attacker.

Security is a multiplayer game

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

- GameNite has radically different applicable threat models than, say, the State Department.

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 endpoint, send arbitrary messages over websockets, and can directly interpret every message that comes from the server.

Mitigations:

1. Don't put secrets in client code
2. Agressively validate API requests
   - more on this later.

Security isn't always free

- In software, as in the real world…
- You just moved to a new house, someone just moved out of it. What do you do to protect you belongings/property?
- What are the assets that need protection?
  - residents, furniture, cash, your stuff
- What are the vulnerabilities?
  - doors, windows

What are the threats? What are possible mitigations?

Threat/Exploit	Mitigation
Fire	Smoke alarm, fire extinguisher
Hurricane	Strengthen windows; raise house; move away
Burglary	Burglar alarm? Better locks? Strengthen windows?
Former owner has keys	Change locks
Terrorism	???
Nuclear war	????
???	

STRIDE Framework can help identify Common Threats

- The **STRIDE** framework is useful to reason about potential threats.
  - **S**poofing
  - **T**ampering
  - **R**epudiation
  - **I**nformation Disclosure
  - **D**enial of Service
  - **E**levation of Privilege

Five major classes of vulnerabilities

- Vulnerability 1: Code that runs in an untrusted environment
- Vulnerability 2: Untrusted Inputs
- Vulnerability 3: Bad authentication (of both sender and receiver!)
- Vulnerability 4: Malicious software from the software supply chain
- Vulnerability 5: Failure to apply security policy.

https://owasp.org/Top10/2025/

Vulnerability 1 Example: authentication code in a web application

Curses! Foiled Again!

Front End

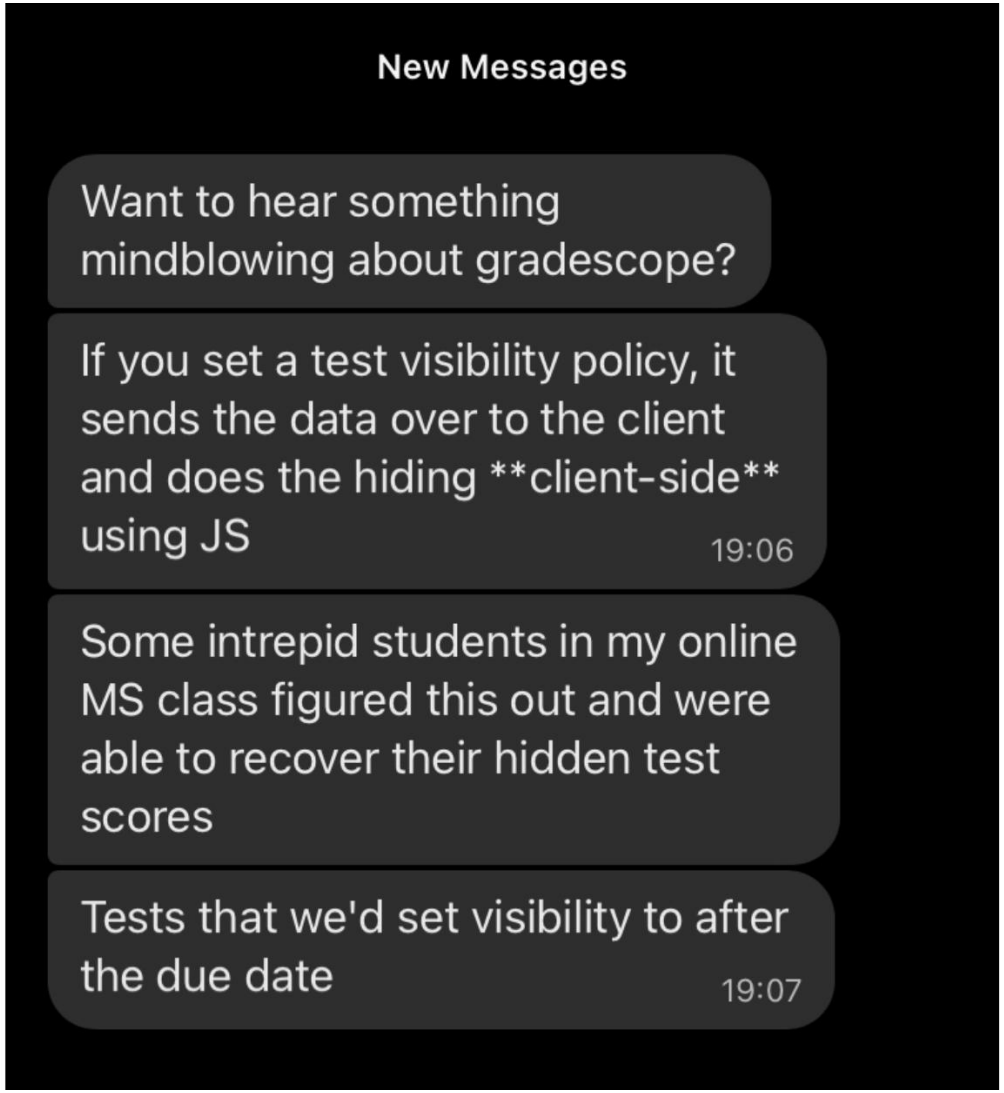
```
function checkPassword(inputPassword:
string){
  if(inputPassword === 'letmein'){
    return true;
  }
  return false;
}
```

Trust boundary

Fix: Move code to back end (duh!)

Back End

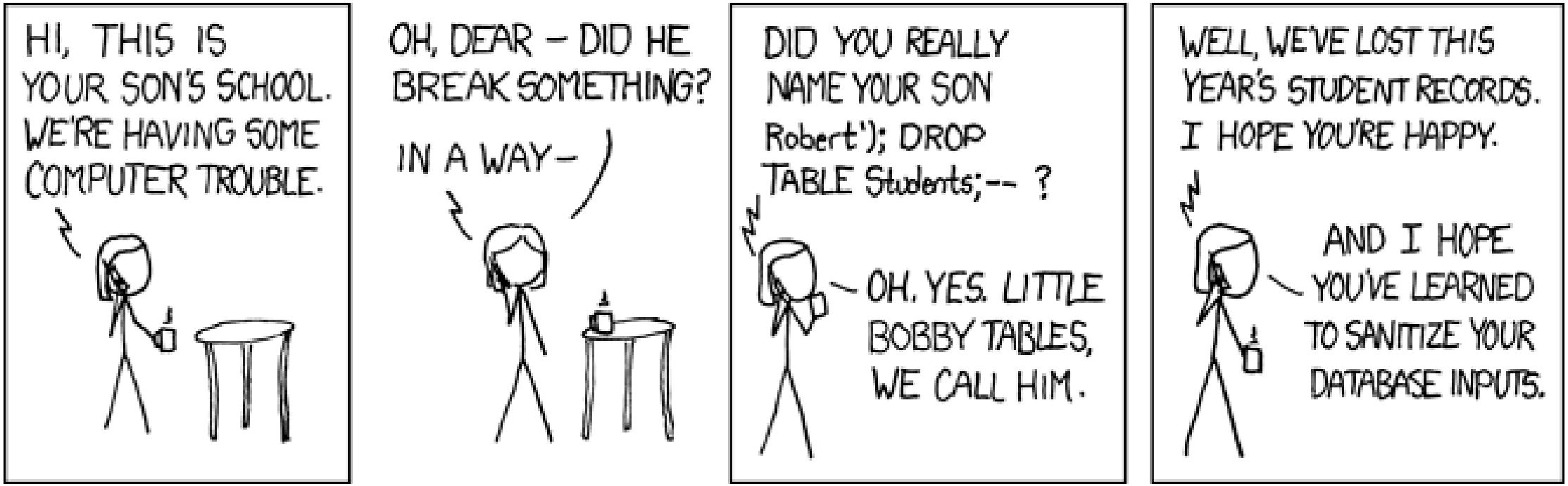
Who would do such a silly thing?



Security Principle #1: Minimize Attack Surfaces

- Mitigations:
  - Study new APIs and commands for new attack surfaces
  - Remove extraneous dependencies
  - Remove unneeded APIs and commands
  - Authenticate users of APIs

Vulnerability 2: Data controlled by a user flowing into our trusted codebase



https://xkcd.com/327/

Example: code injection

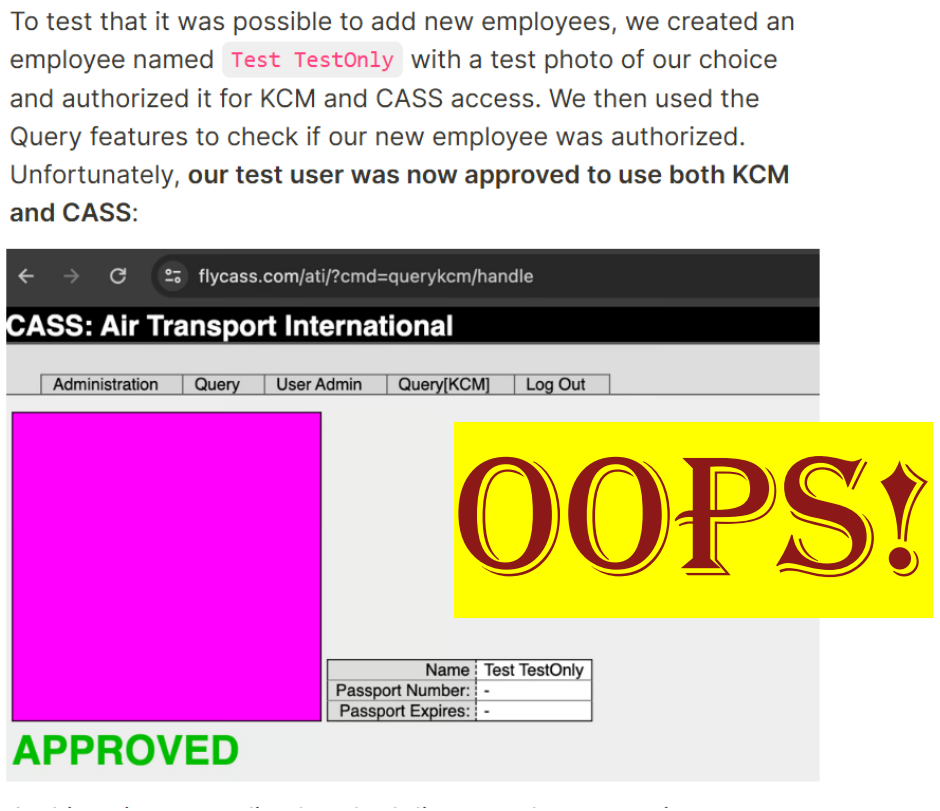
```
String query = "SELECT * FROM accounts WHERE
       name='" + request.getParameter("name") + "'";
```

Parameter name	Constructed Query	Effect
Alice	`SELECT * FROM accounts WHERE name='Alice';`	Select a single account
Alice O'Neal	`SELECT * FROM accounts WHERE name='Alice O'Neal';`	SQL Error
5' OR '1'='1	`SELECT * FROM accounts WHERE name='5' OR '1'='1';`	**Select all accounts**

- OWASP https://owasp.org/Top10/2025/A05_20

Bypassing airport security via SQL injection (2024!)

- "Known Crewmembers" can get to the cockpit without inspection.
- Large airlines: Each airline runs its own authorization system, but small airlines rely on a vendor
- The authors found one such vendor that had an SQL injection error
- Using the username of ' or '1'='1 and password of ') OR MD5('1')=MD5('1, we were able to login to FlyCASS as an administrator of Air Transport International!



https://ian.sh/tsa

A code injection attack (in Apache struts) cost Equifax $1.4 Billion

NEWS

Equifax Says Cybersecurity Breach Has Cost $1.4 Billion

EMMA HURT • MAY 10, 2019

CVE-2017-5638 Detail

Current Description

The Jakarta Multipart parser in Apache Struts 2 2.3.x before 2.3.32 and 2.5.x before 2.5.10.1 has incorrect exception handling and error-message generation during file-upload attempts, which allows remote attackers to execute arbitrary commands via a crafted Content-Type, Content-Disposition, or Content-Length HTTP header, as exploited in the wild in March 2017 with a Content-Type header containing a #cmd= string.

SE-level mitigations for code injection attacks

- Use packages like Zod to sanitize inputs
  - don't do this yourself!
- Use eslint rules to check that Zod is used on every input flow
- Avoid unsafe query languages (e.g. SQL, LDAP, language-specific languages like OGNL in java). Use “safe” subsets instead.
- Avoid use of languages (like C or C++) that allow code to construct arbitrary pointers or write beyond a valid array index
- Never use eval() in JS – executes a string as JS code

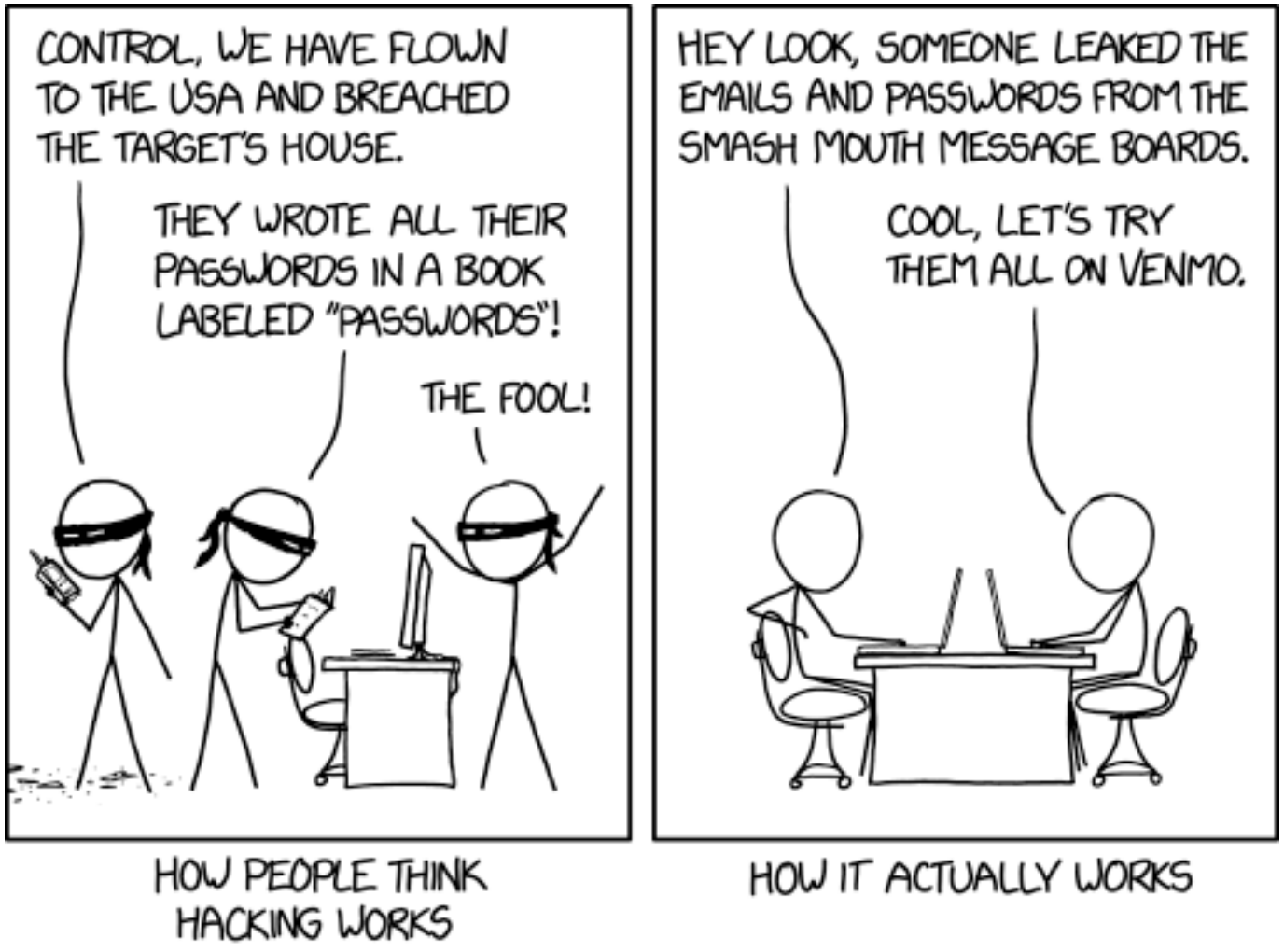
Vulnerability 3: Bad Authentication

- This is mostly about password management
- Attack surface:
  - your password system
  - the attacker may use lists of passwords stolen from other sites

(Inadequate) Mitigation: Don't keep passwords; keep their hashes instead

- Cryptographic hash:
  - Space of hash values is very large (NOT what you'd use for a hash table)
  - hash collisions should be so rare as to be negligible
  - can't go from hash code to original
- Your table consists of (username, savedHash) where savedHash is the hash of the password they created when they signed up for the site.
- Afterwards, the user logs in with (username, maybePassword).
- You compare hash(maybePassword) vs savedHash.

Is this good enough?



…people reuse passwords all the time. Could an attacker take advantage of that?

https://xkcd.com/2176/

Is it safe to store the hashes instead of the passwords themselves?

- Imagine that the attacker has stolen your password db, which looks like
  (username, hashedpassword)
- Next, imagine that the attacker has stolen a table that looks like
  (username, password)
  - this happens all the time!
- So they can easily construct a table that looks like
  (username, password, hash(password))
  - This is sometimes called a "rainbow table"
- And now they can run a join and retrieve a table that looks like
  (username, password)

  for each username that is in both tables

Actual Mitigation: always salt your passwords

- For each user, generate a random number called the "salt"
  - typically 128 bits (16 bytes)
- Your password db should now look like
  - (username, salt, hash(password+salt)
- There's no connection between hash(password+salt) and hash(password)
- So the rainbow table won't work.

Security Principle: Don't try this at home; use an established solution instead

- Use established, well-studied crypto packages
- This is a general security principle

```
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`.

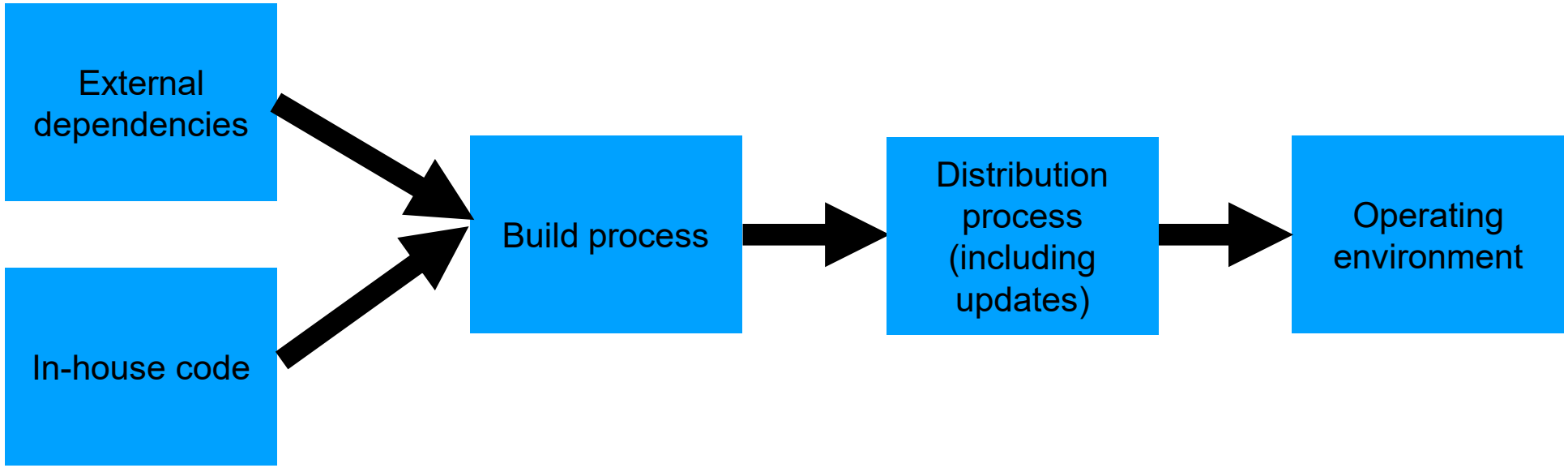
The `callback` function is called with two arguments: `err` and `derivedKey`. `err` is an exception object when key derivation fails, otherwise `err` is `null`. `derivedKey` is passed to the callback as a `Buffer`.

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

Vulnerability 4: Supply-Chain Attacks

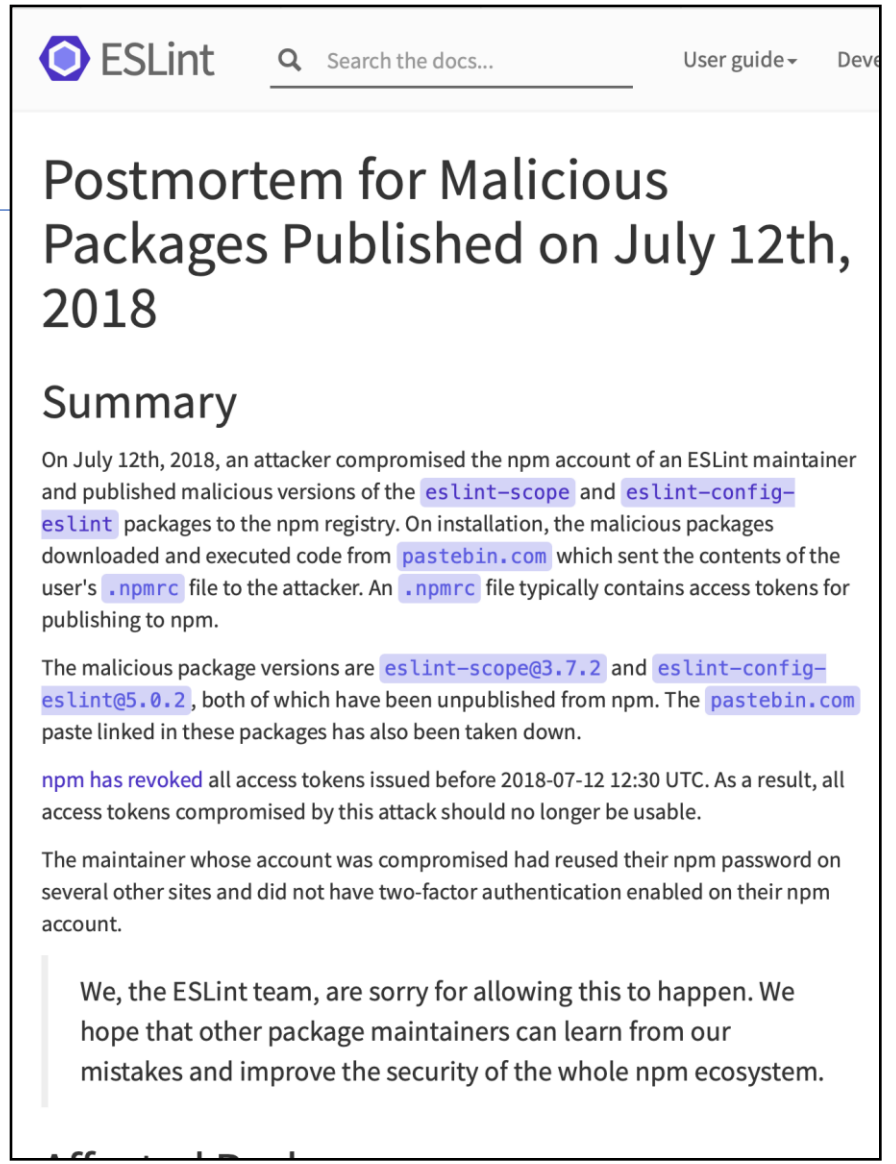
- Do we trust our own code?
- Third-party code provides an attack vector

The software supply chain has many points of weakness



Example: the eslint-scope attack (2018)

- On 7/12/2018, a malicious version of eslint-scope was published to npm.
- eslint-scope is a core element of eslint, so many many users were affected.
- Let’s analyze this…



ESLint Search the docs... User guide Deve

Postmortem for Malicious Packages Published on July 12th, 2018

Summary

On July 12th, 2018, an attacker compromised the npm account of an ESLint maintainer and published malicious versions of the `eslint-scope` and `eslint-config-eslint` packages to the npm registry. On installation, the malicious packages downloaded and executed code from `pastebin.com` which sent the contents of the user's `.npmrc` file to the attacker. An `.npmrc` file typically contains access tokens for publishing to npm.

The malicious package versions are `eslint-scope@3.7.2` and `eslint-config-eslint@5.0.2`, both of which have been unpublished from npm. The `pastebin.com` paste linked in these packages has also been taken down.

npm has revoked all access tokens issued before 2018-07-12 12:30 UTC. As a result, all access tokens compromised by this attack should no longer be usable.

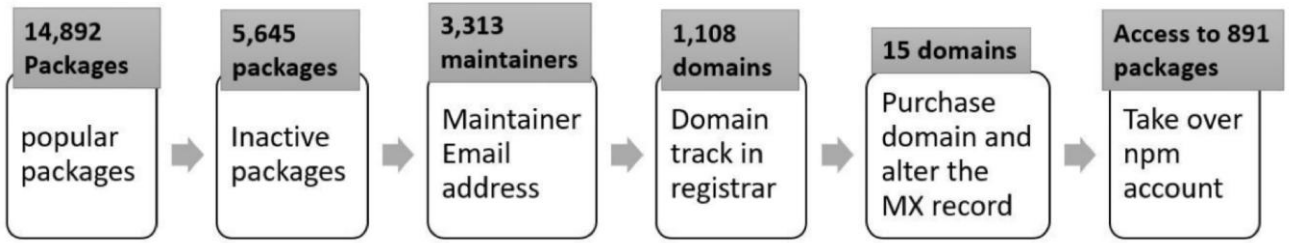
The maintainer whose account was compromised had reused their npm password on several other sites and did not have two-factor authentication enabled on their npm account.

> We, the ESLint team, are sorry for allowing this to happen. We hope that other package maintainers can learn from our mistakes and improve the security of the whole npm ecosystem.

https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes/

A 2021 NCSU/Microsoft found that many of the top 1% of npm packages had vulnerabilities

- Package inactive or deprecated, yet still in use
- No active maintainers



- At least one maintainer with an inactive (purchasable) email domain
- Too many maintainers or contributors to make effective maintenance or code control
- Maintainers are maintaining too many packages
- Many statistics/combinations: see the paper for details.

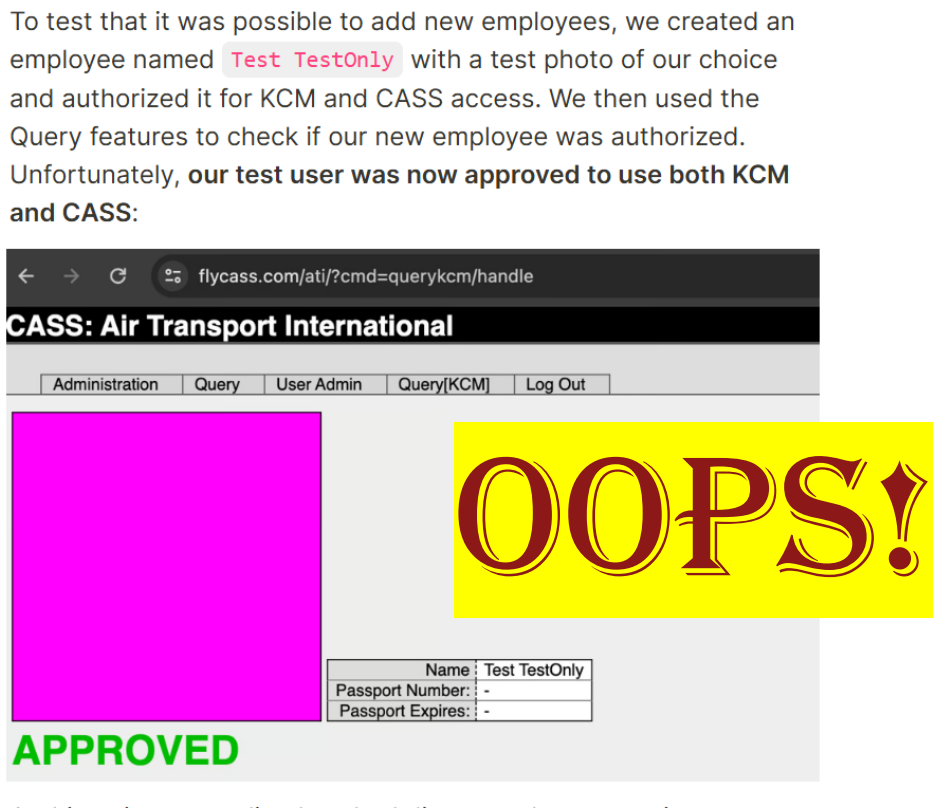
"What are Weak Links in the npm Supply Chain?" By: Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, Laurie Williams https://arxiv.org/abs/2112.10165

Curl is maintained by a single person!?!

- (slide added by mitch 3/3/26)
- (need to fill in bullets)
- Curl is maintained by a single person

Your suppliers' risks are your risks

- "Known Crewmembers" can get to the cockpit without inspection.
- Large airlines: Each airline runs its own authorization system, but small airlines rely on a vendor
- The authors found one such vendor that had an SQL injection error
- Using the username of ' or '1'='1 and password of ') OR MD5('1')=MD5('1, we were able to login to FlyCASS as an administrator of Air Transport International!



https://ian.sh/tsa

Process-level Threat Mitigations

- External dependencies
  - Audit all dependencies and their updates before applying them
- In-house code
  - Require developers to sign code before committing, require 2FA for signing keys, rotate signing keys regularly
- Build process
  - Audit build software, use trusted compilers and build chains
- Distribution process
  - Sign all packages, protect signing keys
- Operating environment
  - Isolate applications in containers or VMs

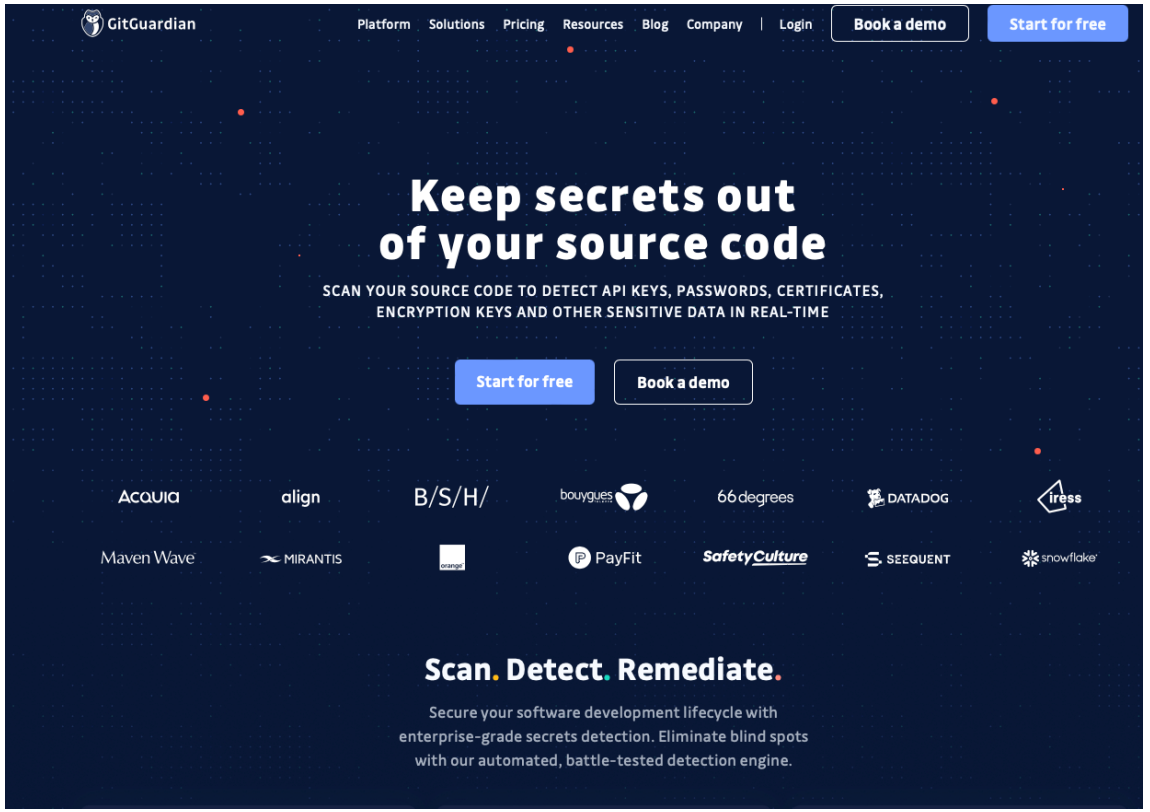
Vulnerability #5: Failure to Apply Security Policies

Remember the outline of our lecture:

1. Definition of key vocabulary
2. Some common vulnerabilities, and possible mitigations
3. Getting security right is about people as well as software.

Example mechanism: secret detection

- Recall: SSL only talks about public/private keys.
- Applications may have many other *secret* values (e.g. access tokens for other services)
- Tools like *GitGuardian* automatically detect secrets in repositories
- What happens when we put that in the toolchain?



Do developers keep secret keys secret, even when they are warned?

- 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 it a tool problem or a management problem?

“Why secret detection tools are not enough: It’s not just about false positives - An industrial case study”
Md Rayhanur Rahman, Nasif Imtiaz, Margaret-Anne Storey & Laurie Williams https://link.springer.com/article/10.1007/s10664-021-10109-y

Other mitigations for access-control threats

- Implement multi-factor authentication
- Make sure passwords are not weak, have not been compromised.
- Apply per-record access control
  - Principle of least privilege
- Harden account creation, password reset pathways
- Use an expert vendor, like Auth0, to handle login
  - They can do it better than you can.

But how do you get your developers to do all this?

David Blank-Edelman (former head of Systems at Khoury)

"The solution is in front of the screen, not behind it"

A security architecture must include a security culture

- Security architecture is a set of mechanisms and policies that we build into our system to mitigate risks from threats
- Vulnerability: a characteristic or flaw in system design or implementation, or in the security procedures, that, if exploited, could result in a security compromise
- Threat: potential event that could compromise a security requirement
- Attack: realization of a threat

Incorporating Security into an Agile Development Process

- Collaborate with stakeholders to understand security needs
- Frequent and small iterations
  - Start with the thinnest slice of the system. e.g.,
    - User registration flow
    - A microservice and it’s collaborating services
    - Current iteration
  - Repeat and refine them.

Let’s Walk through an Example

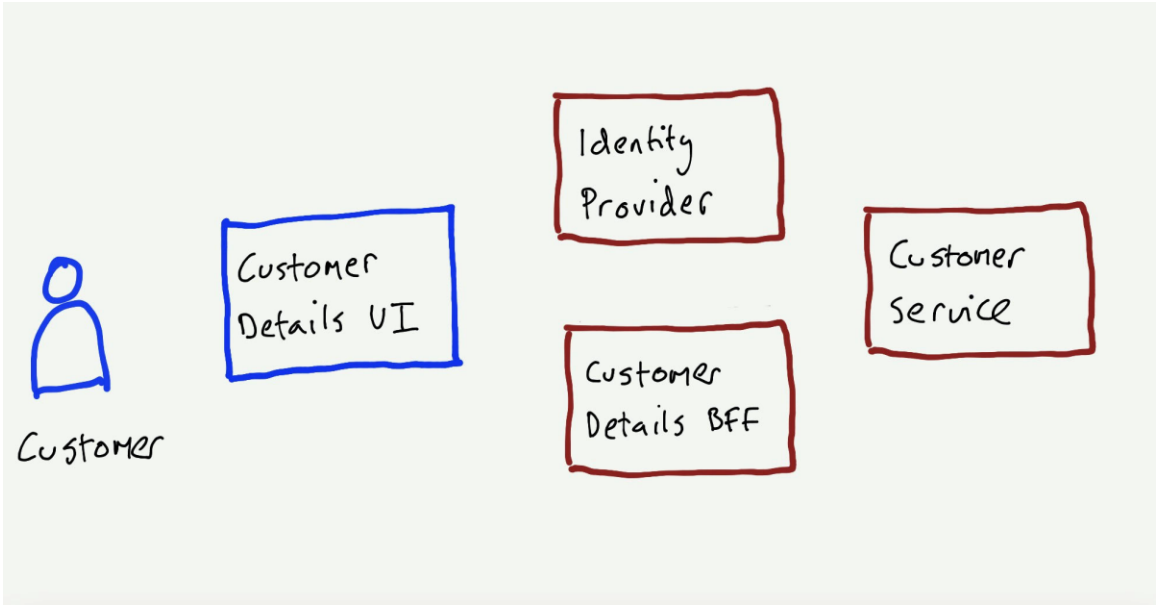
- Imagine working on this user story

> “As a customer, I need a page where I can see my customer details so I can confirm they are correct”

- Let’s work through our questions:
  - What are you building?
  - What can go wrong?
  - What are you going to do?

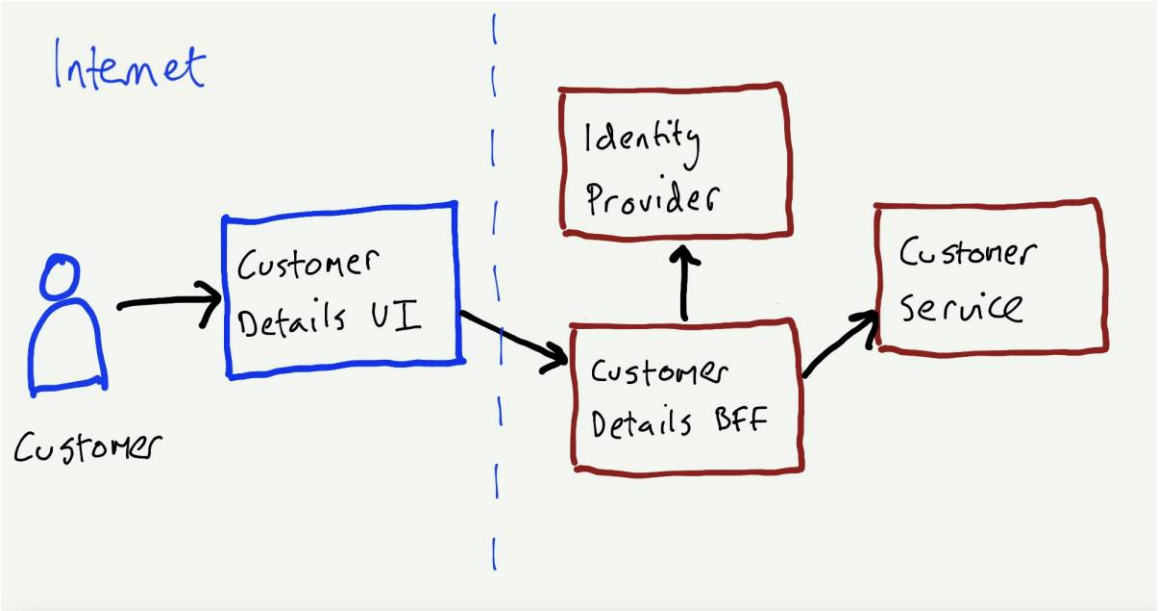
What are we building?

- Use a sketch to represent
  - relevant components
  - users who interact with a component
  - collaborative components



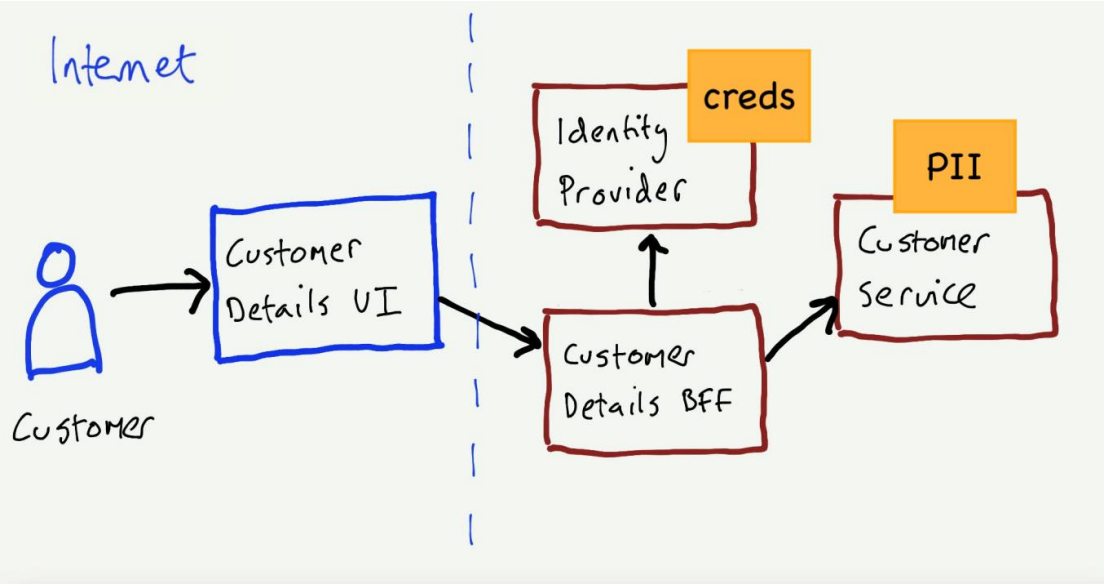
What are we building?

- Explicitly model data flows in the sketch.
- Data flows help show where requests originate (source).
- Label networks and show boundaries between them.



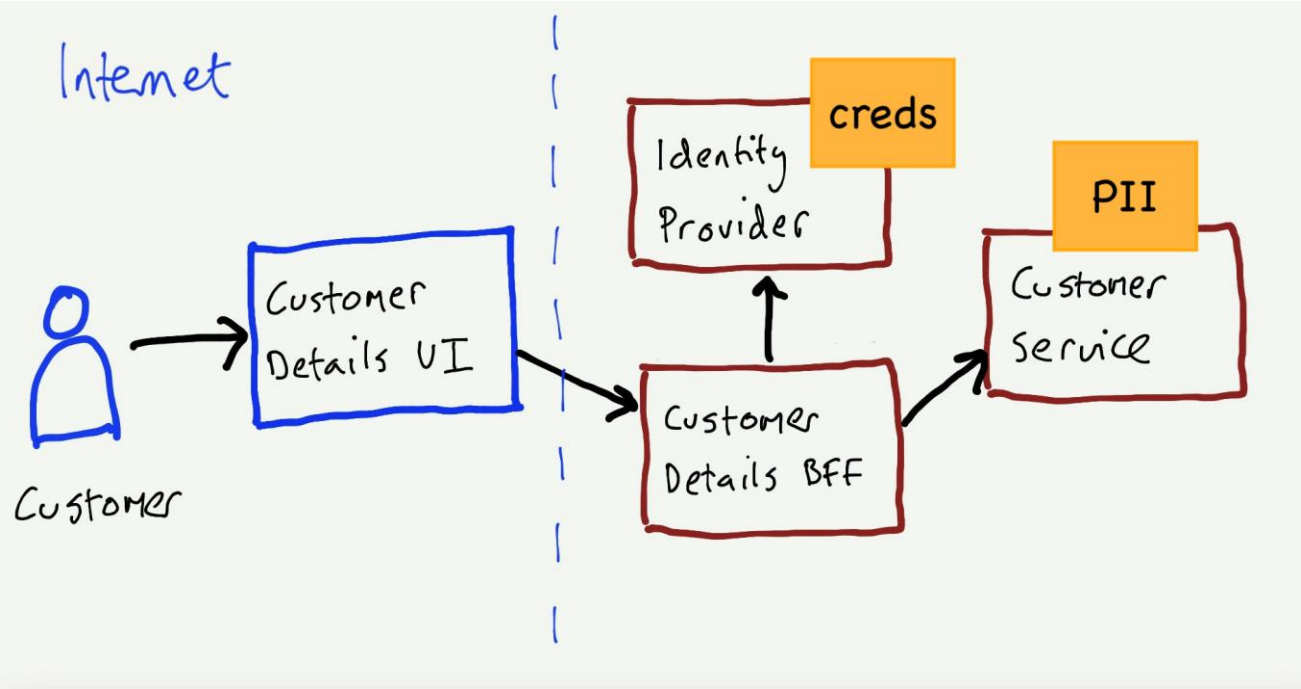
What are we building?

- Identify and show assets e.g., personally identifiable information (PII), your application has access to.



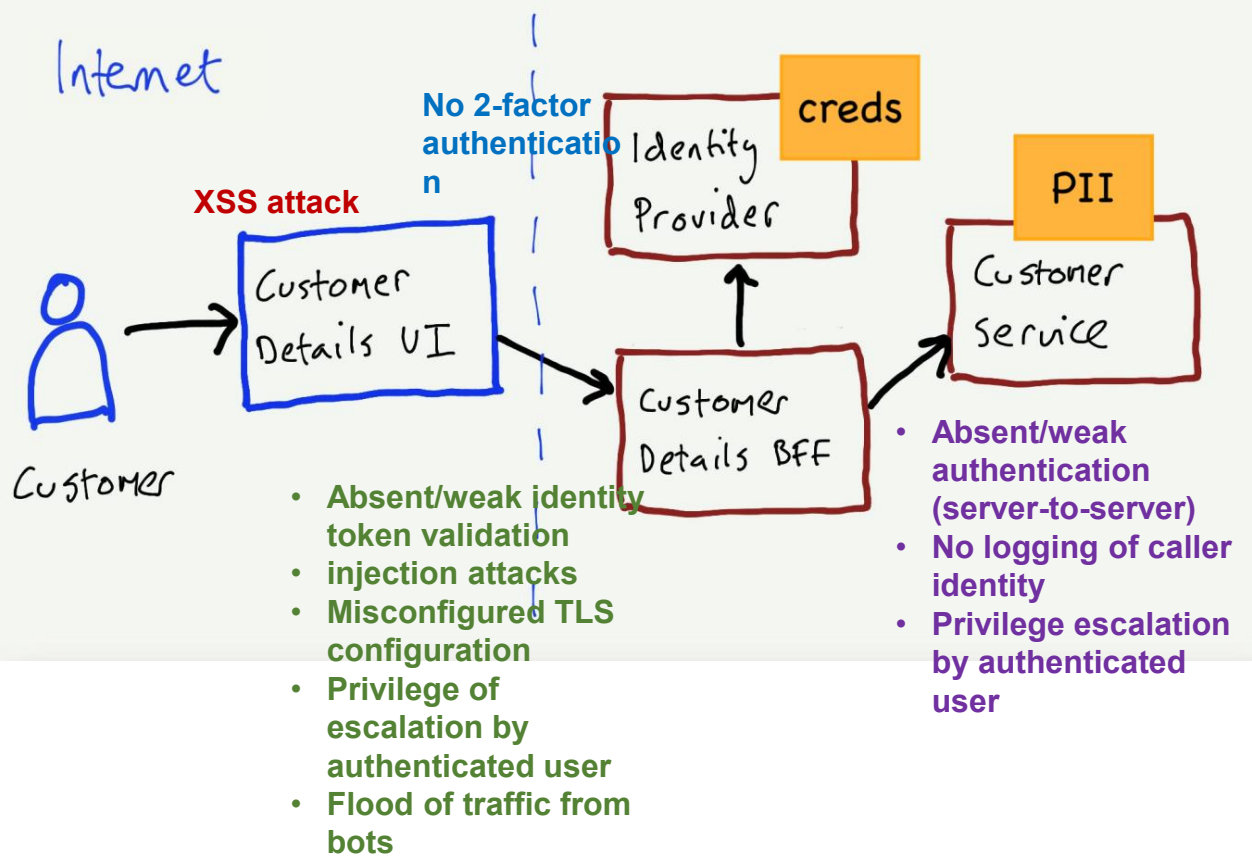
Every data flow is an attack surface

- There may be vulnerabilities on each flow.
  - Customer -> UI
  - Customer -> Identity Service
  - UI -> BFF
  - BFF -> Customer Service



Identify the vulnerabilities and threats on each flow

- Identify the possible vulnerabilities and threats on each flow.
  - Customer -> UI
  - Customer -> Identity Service
  - UI -> BFF
  - BFF -> Customer Service



Choose highest priority vulnerabilities

- Add Highest Priority threats to backlog (as user stories or conditions of satisfaction)

Given the user is logged in
When they request to view their profile page And they have a valid token
Then their profile page is displayed

Given the user is logged in
When they request to view their profile page
But they do not have a valid token
Then they are asked to login or signup

What are you going to do?

- Prioritize and Fix
  - Aim to address manageable number of threats (e.g., 3)
- Examples:
  - Authorization bypass when accessing an API.
  - XSS attack via user input.
  - Denial of service from the internet

- **Repeat and refine**!

A Guide to Threat Modeling by Jim Gumbley: https://martinfowler.com/articles/agile-threat-modelling.html

Learning Objectives for this Module

- You should now be able to:
  - Define key terms relating to software/system security
  - Explain 5 common vulnerabilities in web applications and similar software systems, and describe some common mitigations for each of them.
  - Explain why software alone isn’t enough to assure security
  - Explain at least one way in which security can be incorporated into an agile development process