CS 4530: Fundamentals of Software Engineering

Module 12: Software Engineering & Security

Adeel Bhutta, Joydeep Mitra 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 why it is important to consider non-functional requirements like security during design
  - Explain 5 common vulnerabilities in web applications and similar software systems, and describe some common mitigations for each of them.
  - Explain the STRIDE Framework for Security by Design

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 event (or attack) 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 threat

Security: Basic Vocabulary (3)

- Exploit: a technique or method for exploiting a vulnerability

- Attack: realization of a threat

- Mitigation: a technique for making an attack less likely, more expensive, or less valuable to an attacker.

- Security architecture: a set of mechanisms and policies that we build into our system to mitigate risks from threats

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

Security is about managing risk

- Increasing security might:
    - Increase development & maintenance cost
    - Increase infrastructure requirements
    - Degrade performance
- But, if we are attacked, increasing security might also:
    - Decrease financial and intangible losses

Secure by Design is mantra of modern software development

- Traditional Software Process Models do not bake security into the software development lifecycle.
  - Security issues are usually an after-thought.
- The modern approach to secure software engineering is to **consider security during the design phase**.
- How? By starting with **threat modeling**.

Threat modeling can help us analyze the issues

- What is being defended? Who do we trust? What parts of the system do we trust?

- What malicious actors exist and what attacks might they employ? (**Identification**)

- What is the likelihood and impact of each threat? (**Assessment**)

- What can we do in case of attack? (**Countermeasures**)

- How do we adapt to evolving risks? How do we update threat models regularly? (**Review**)
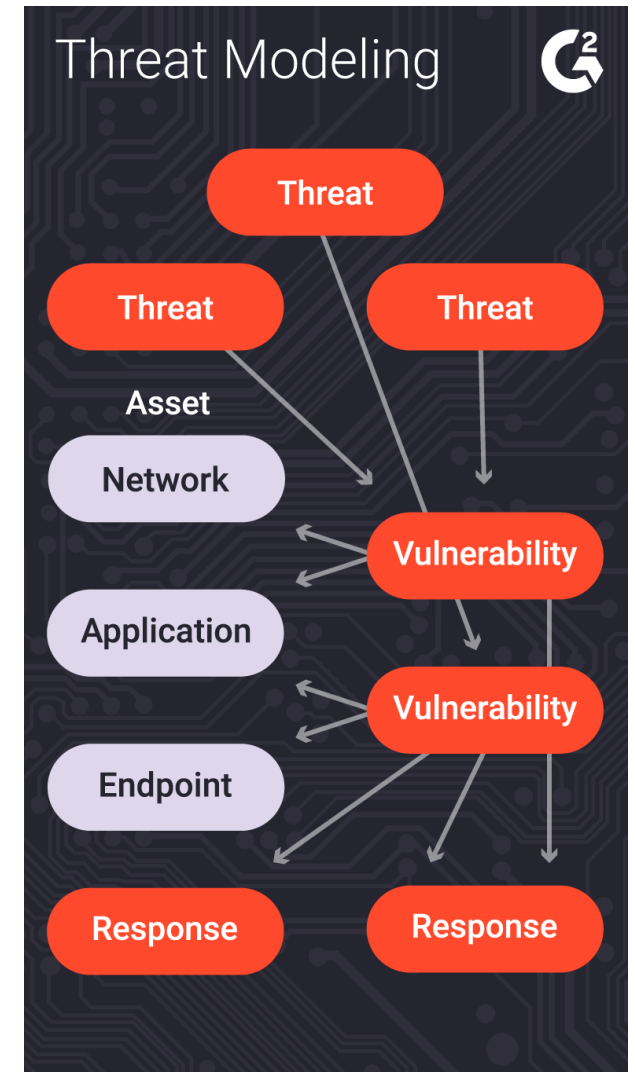
Two views of Security

1. **Let's look at Security from High Level first (call it Project-Level Security)**

2. Then we will look at Security from the App level.

A Baseline Threat Model {at Project Level}

- Trust:
  - Developers writing our code (at least for the code they touch)
  - Server running our code
  - Popular dependencies that we use and update
- Don't trust:
  - Code running in browser
  - Inputs from users
  - Other employees (employees should have access only to the resources they need)

Man vs. Musk: A Whistleblower Creates Headaches for Tesla

An employee who was fired after expressing safety concerns leaked personnel records and sensitive data about driver-assistance software.
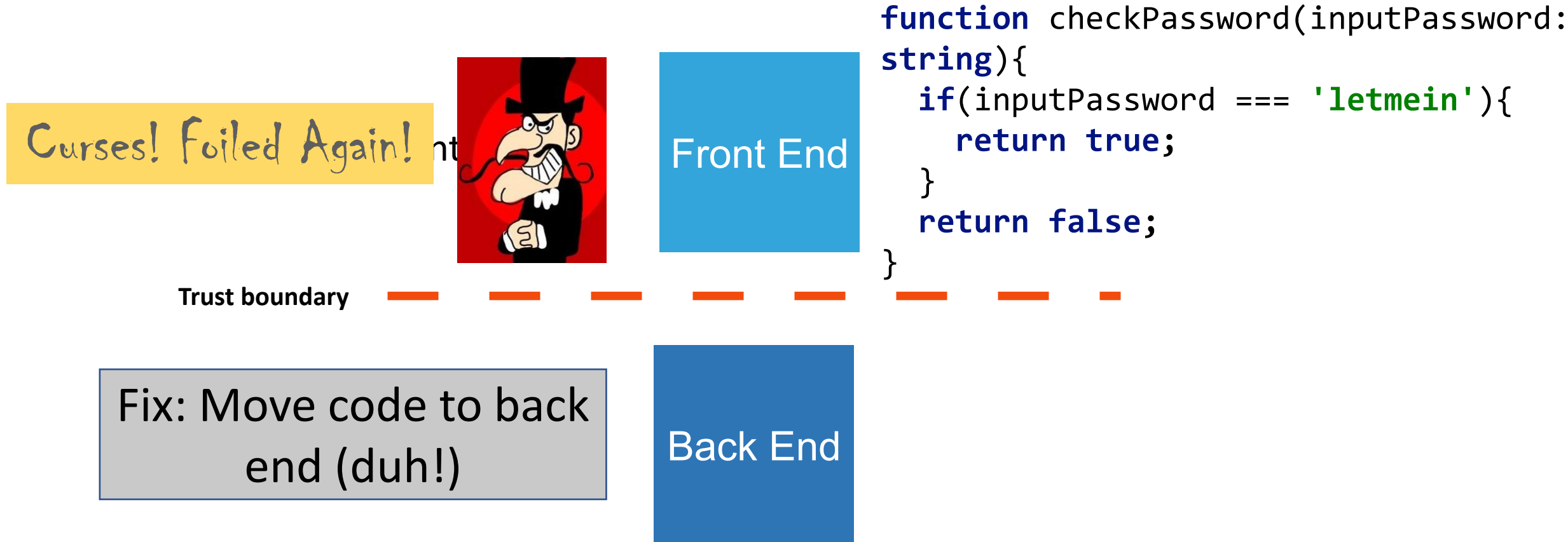
A Baseline Security Policy {at Project Level}

- Encrypt all data in transit, sensitive data at rest

- Use multi-factor authentication

- Use encapsulated zones/layers of security
  - Different people have access to different resources
  - Principle of Least Privilege

- Log everything! (employee data accesses/modifications) (maybe)

- Do regular, automatic, off-site backups

- Bring in security experts early for riskier situations
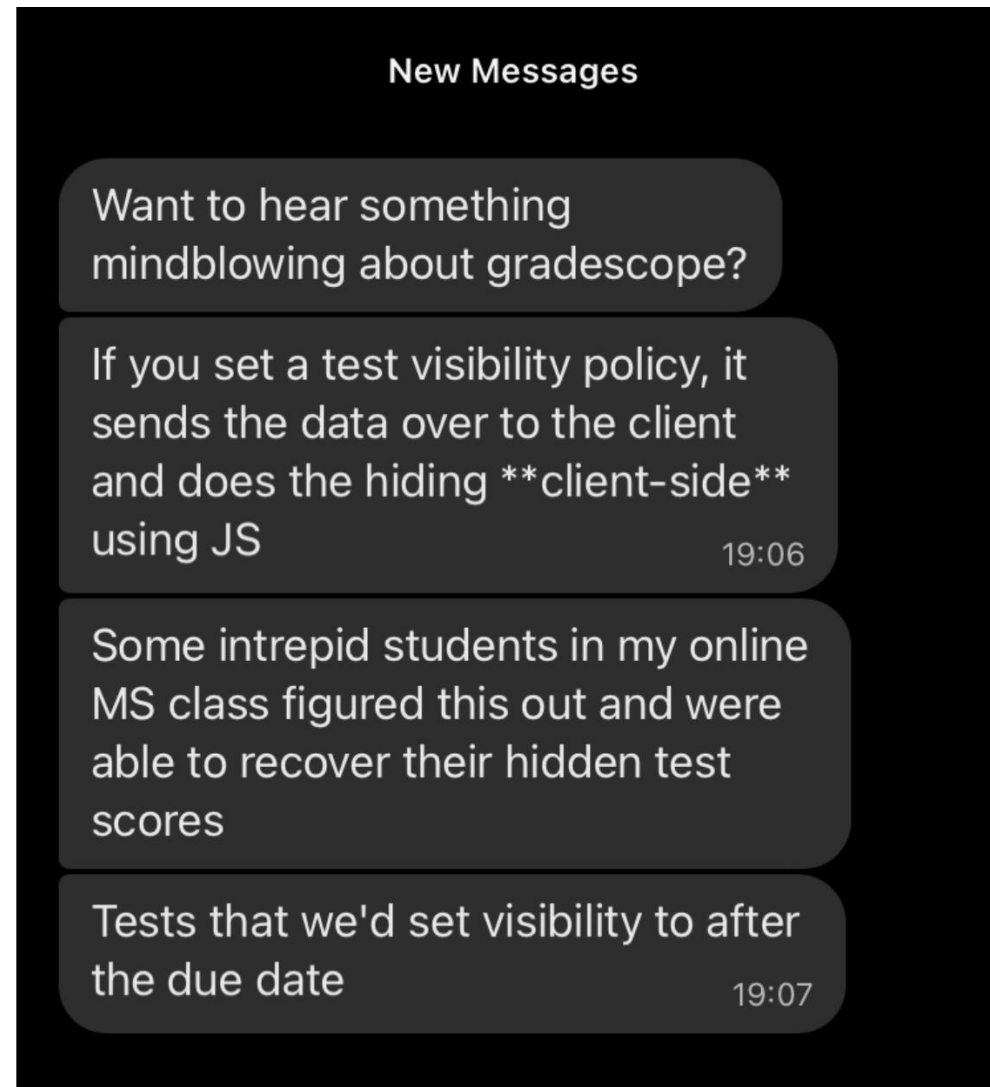
5 major classes of vulnerabilities

- 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/www-project-top-ten/

Vulnerability 1 Example: authentication code in a web application

Curses! Foiled Again!

Front End

Trust boundary

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

Fix: Move code to back end (duh!)

Back End

Who would do such a silly thing?

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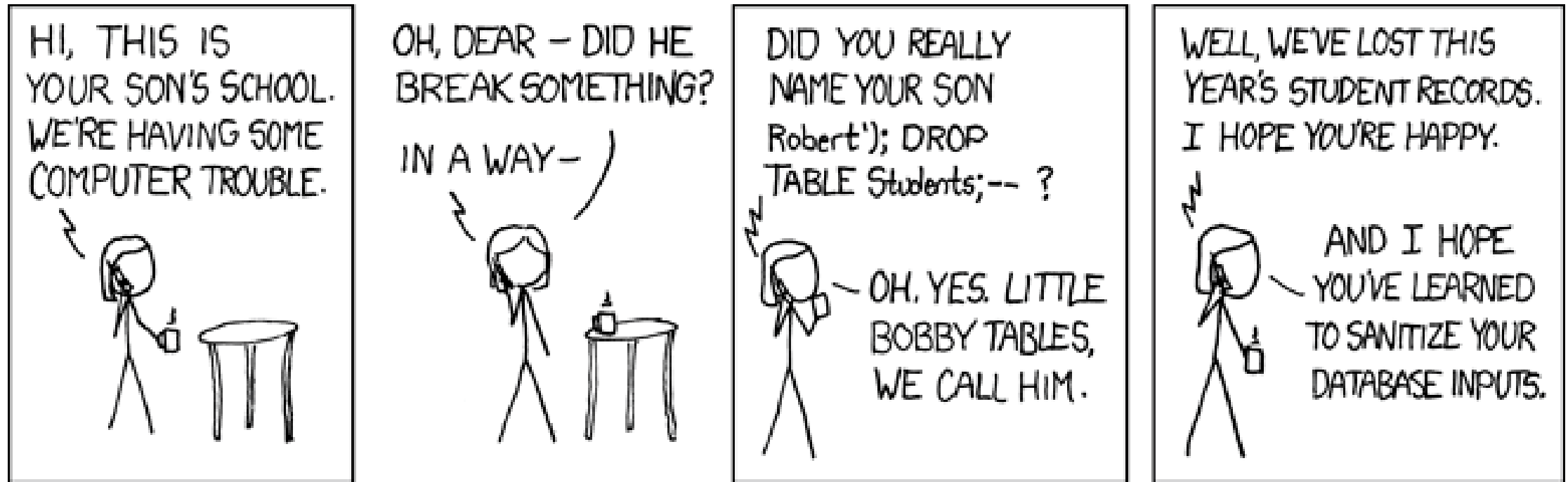
Example: code injection

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

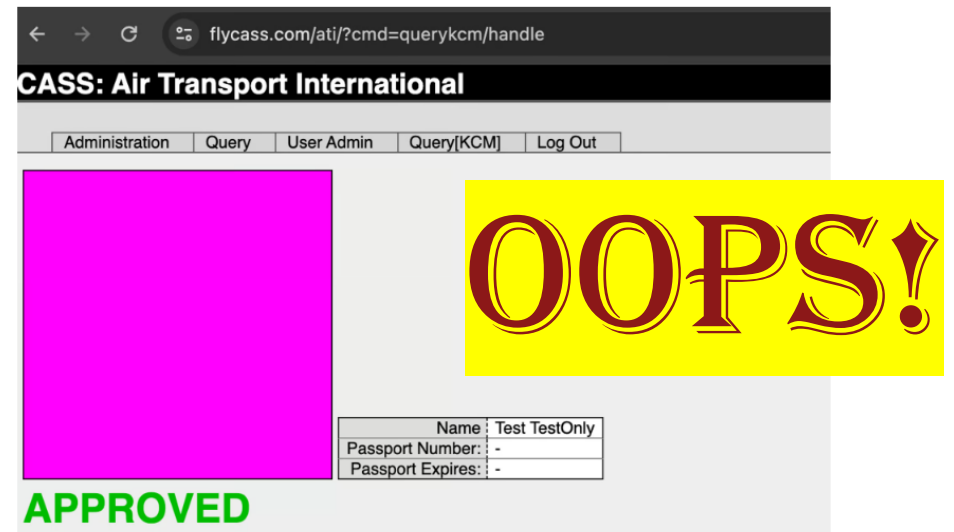| Parameter name	Constructed Query	Effect
Alice	SELECT * FROM accounts WHERE name='Alice';	Select a single account
Alice O'Neal	SELECT * FROM accounts WHERE name='Alice O'Neal';	SQL Error
5' OR '1'='1	SELECT * FROM accounts WHERE name='5' OR '1'='1';	**Select all accounts**

OOPS!

- OWASP A03:2021-Injection

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!

To test that it was possible to add new employees, we created an employee named Test TestOnly with a test photo of our choice and authorized it for KCM and CASS access. We then used the Query features to check if our new employee was authorized. Unfortunately, **our test user was now approved to use both KCM and CASS:**

https://ian.sh/tsa

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

EQUIFAX
English ▲▼   Return to equifax.com ▸

2017 Cybersecurity Incident & Important Consumer Information

Need help? **Contact Us**

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.

Project-level mitigations for code injection attacks

- Use tools like TSOA or swagger-codegen to automatically generate safe code.

- Manually sanitize inputs to prevent them from being executable

- 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

- eval() in JS – executes a string as JS code

Vulnerability 3: Bad Authentication

HTTP Request

HTTP Response

Server

client page
("Avery")

Server
("Amazon")
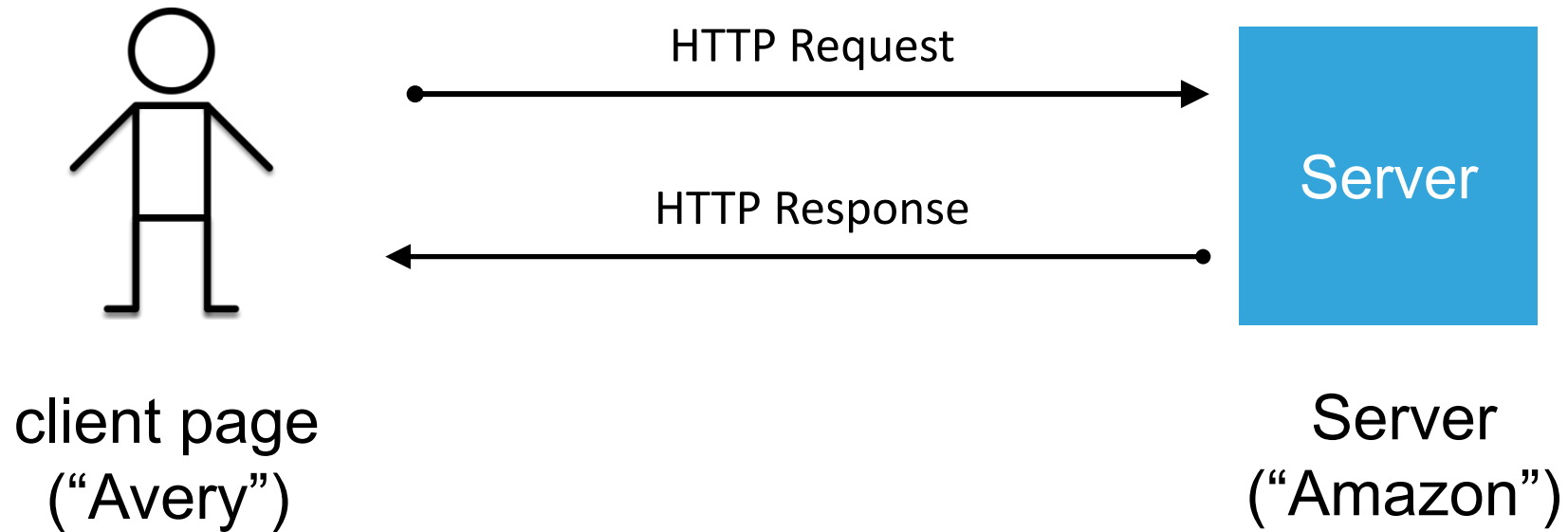
- How does Amazon know that this request is coming from Avery?

- How does Avery know that this response is coming from Amazon?

How does Amazon, Inc. know that this request is coming from Avery?

- Password
  - Establishes that the request is coming from someone who knows Avery's password

- 2-factor authentication is a way of linking Avery's password to the real Avery.
  - Something the real Avery has (physical key, bank card, device token)
  - Something the real Avery knows (name of first pet, etc.)
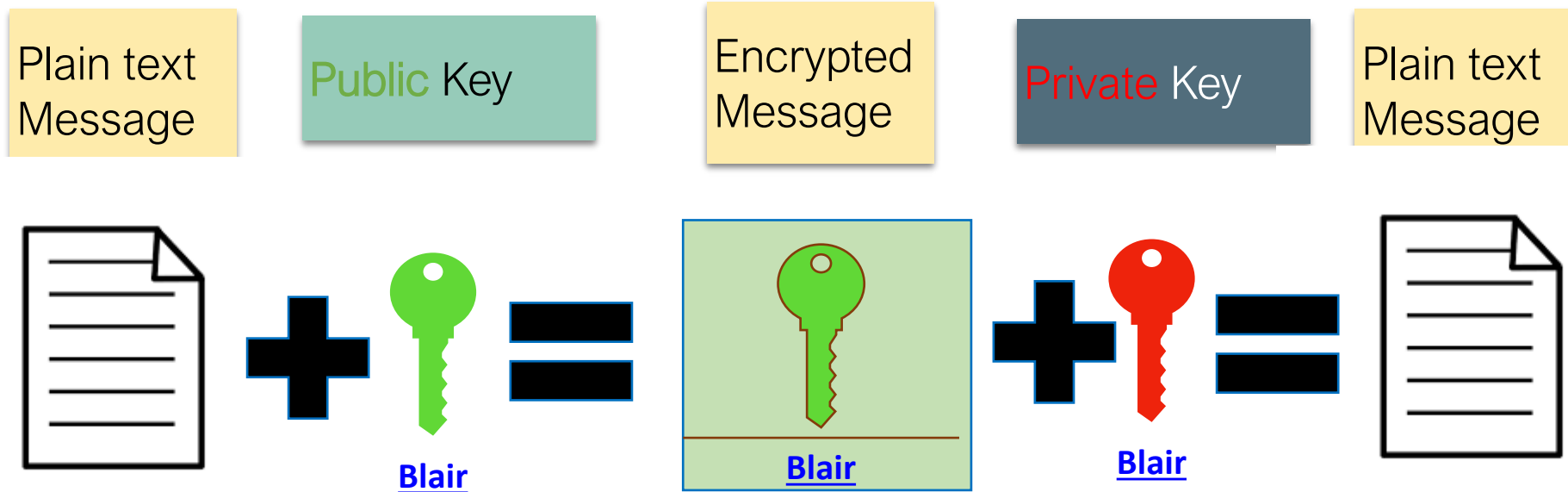 - Something the real Avery is (biometrics, address history, etc.)

How does Avery know that this request is coming from Amazon, Inc.?

- The answer depends on public-key cryptography (PKI)
  - also called "asymmetric cryptography"

- "Cryptographic Failures" is #1 on the 2021 OWASP list of Top 10 Web Application Security Risks

- PKI uses two keys: Public and Private

Encrypted with:	Who can encrypt?	Who can decrypt?
Private Key	Only the owner of the private key	Anyone with the public key, i.e. everyone
Public Key	Anyone	Only the owner of the corresponding private key

A message encrypted with a public key can only be decrypted with the matching private key

Avery wants to send a confidential message to Blair, so they encrypt it with Blair's public key

| Plain text Message | Public Key | Encrypted Message | Private Key | Plain text Message |

Blair + **Blair** = **Blair**

Only Blair has the key to decrypt the message!

Confidentiality achieved!

Encrypt messages with the sender's private key to ensure integrity

Avery wants to send a message, and they want whoever reads to be confident that it was Avery who sent the message.

| Plain text Message | Private Key | Encrypted Message | Public Key | Plain text Message |
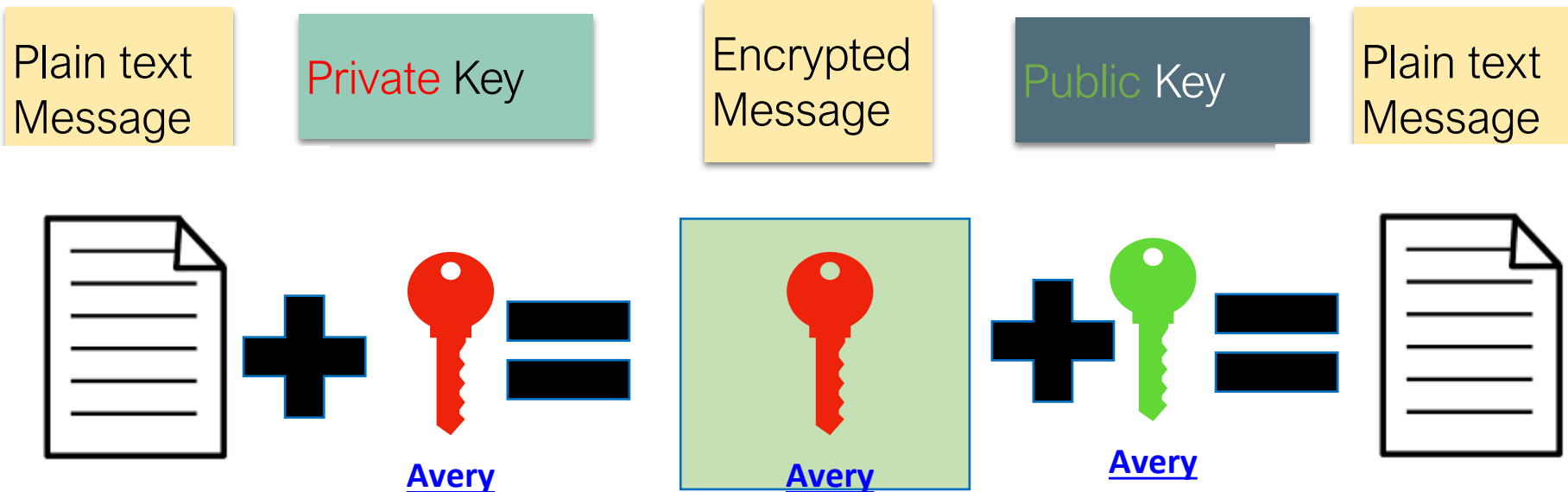
Avery

Avery

Avery

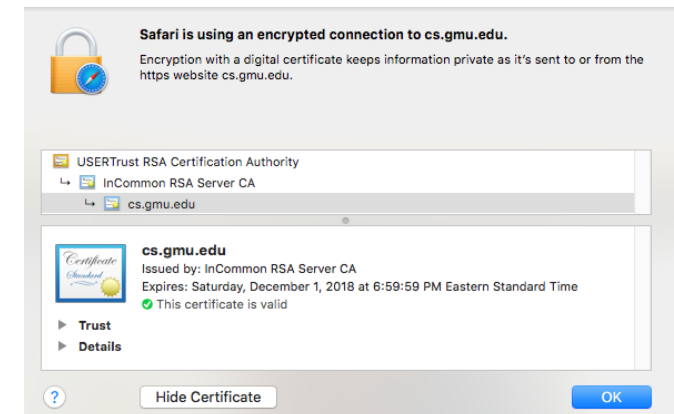Only Avery had the key this was locked with!

Integrity achieved!

How does Avery get public key from Amazon, Inc.?

- Avery can rely on a third party, called a "certificate authority" (CA).

- The CA can *endorse* that a public key is held by a certain real-world entity.

- The third party issues a *certificate* containing the Amazon's public key, encrypted with the CA's own private key.

- When our browser visits amazon.com, amazon.com sends its certificate to our browser.

- Avery decrypts the certificate, using the CA's public key. Avery now has the real public key of "Amazon Inc".

- Every computer/browser is shipped with these root CA public keys

To acquire a certificate, Amazon, Inc. must have shared their public key and some real-world proof that they are amazon.com to the CA.

Safari is using an encrypted connection to cs.gmu.edu.
Encryption with a digital certificate keeps information private as it's sent to or from the https website cs.gmu.edu.

USERTrust RSA Certification Authority
  └ InCommon RSA Server CA
      └ cs.gmu.edu

cs.gmu.edu
Issued by: InCommon RSA Server CA
Expires: Saturday, December 1, 2018 at 6:59:59 PM Eastern Standard Time
This certificate is valid

▶ Trust
▶ Details

Hide Certificate   OK

What happens if a CA is compromised, and issues invalid certificates?

Security

Comodo-gate hacker brags about forged certificate exploit

Tiger-blooded Persian cracker boasts of mighty exploits

OOPS!

Security

Fuming Google tears Symantec a new one over rogue SSL certs

We've got just the thing for you, Symantec ...

By Iain Thomson in San Francisco 29 Oct 2015 at 21:32   36   SHARE ▼

Google has read the riot act to Symantec, scolding the security biz for its

You can do this for your website for free
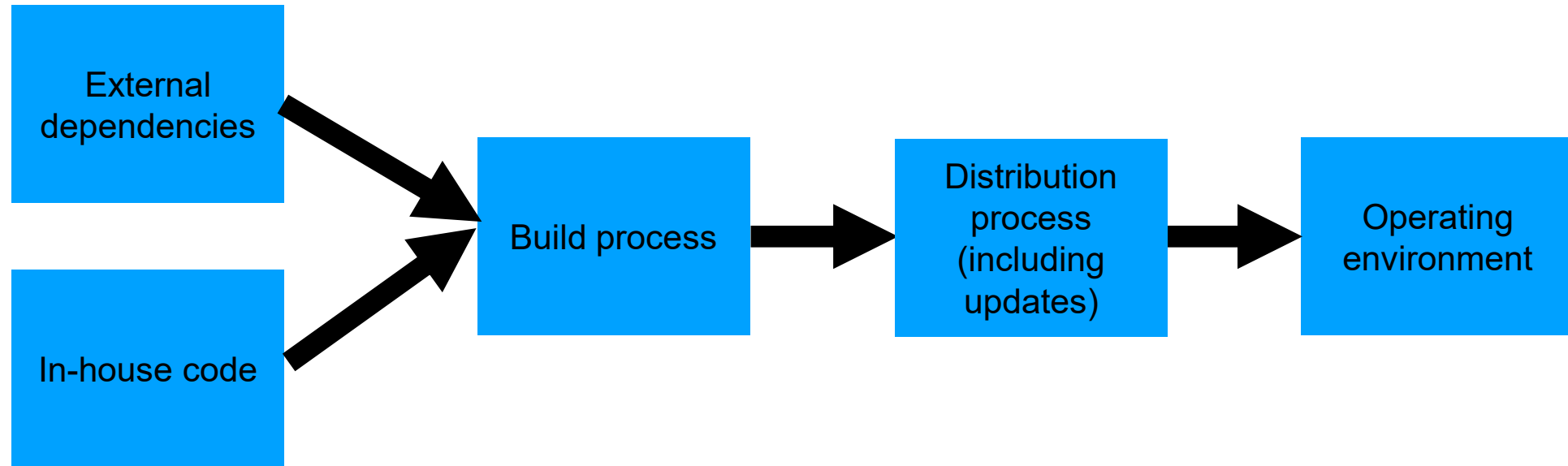
- letsencrypt.com

Project-level 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 pathways for account creation, password reset.

- Use an SSO to handle login
  - They might do it better than you can.
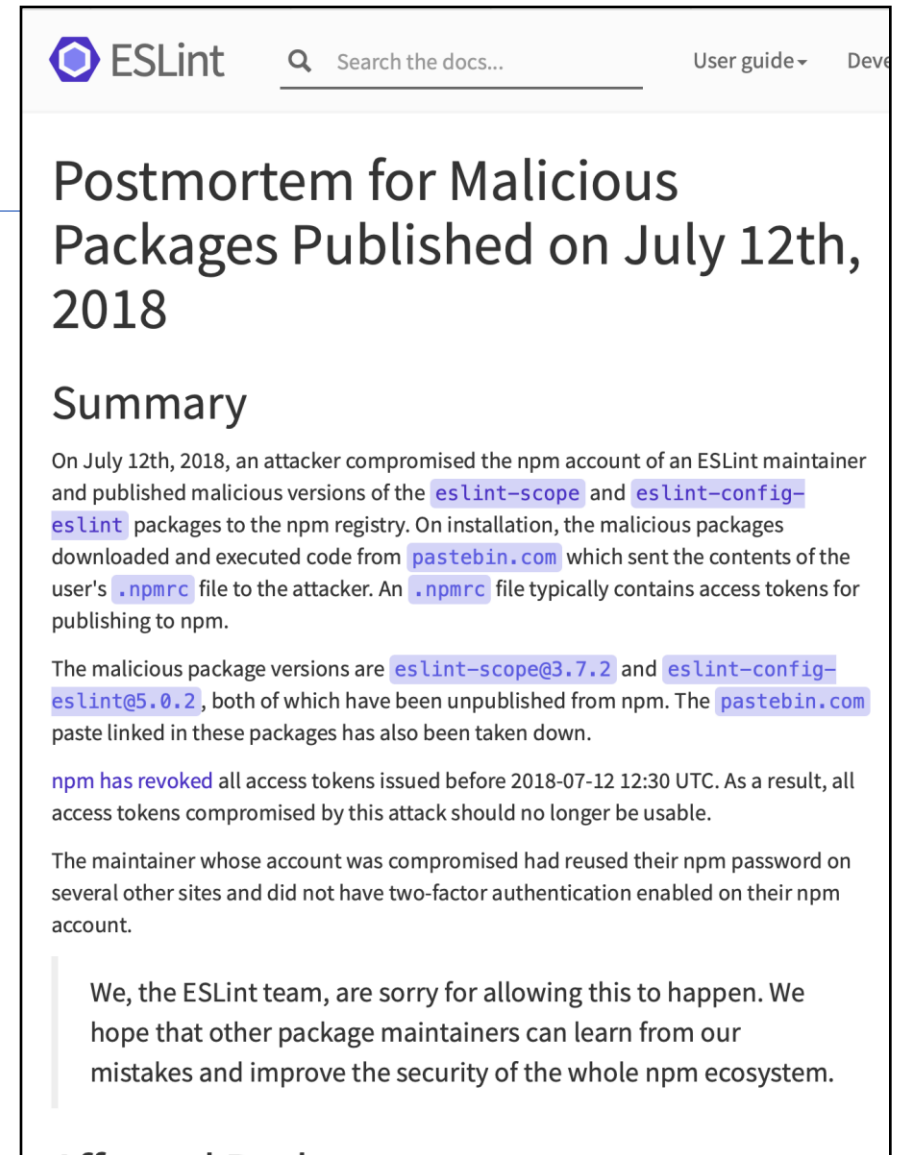
Vulnerability 4: Supply-Chain Attacks

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

The software supply chain has many points of weakness

External dependencies → Build process

In-house code → Build process

Build process → Distribution process (including updates) → Operating environment

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...

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/

33

This incident leveraged several small security failures

- An eslint-scope developer used their same password on another site.

- The other site did not use 2FA

- Password was leaked from the other site.

- Attacker created malicious version of eslint-scope

- Many users did not use package-lock.json, so their packages automatically installed the new (evil) version.

- The malicious version sent copies of the user's .npmrc to the attacker.   This file typically contains user tokens.

- Estimated 4500 tokens were leaked and needed to be revoked.

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
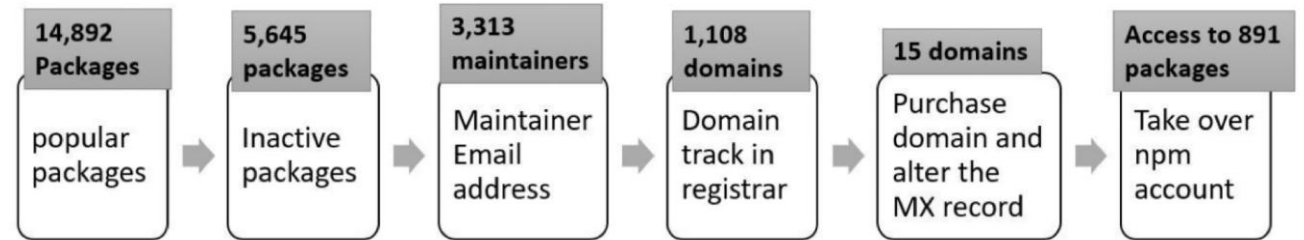
Your suppliers' risks are your risks

- "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!

To test that it was possible to add new employees, we created an employee named `Test TestOnly` with a test photo of our choice and authorized it for KCM and CASS access. We then used the Query features to check if our new employee was authorized. Unfortunately, **our test user was now approved to use both KCM and CASS:**

https://ian.sh/tsa

Project-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: A security architecture must include a security culture

Getting security right is about people as well as software

- Question: How do you get your developers to do all this?
- Answer: Security architecture is a set of ==mechanisms and policies== that we build into our system to mitigate risks from threats

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
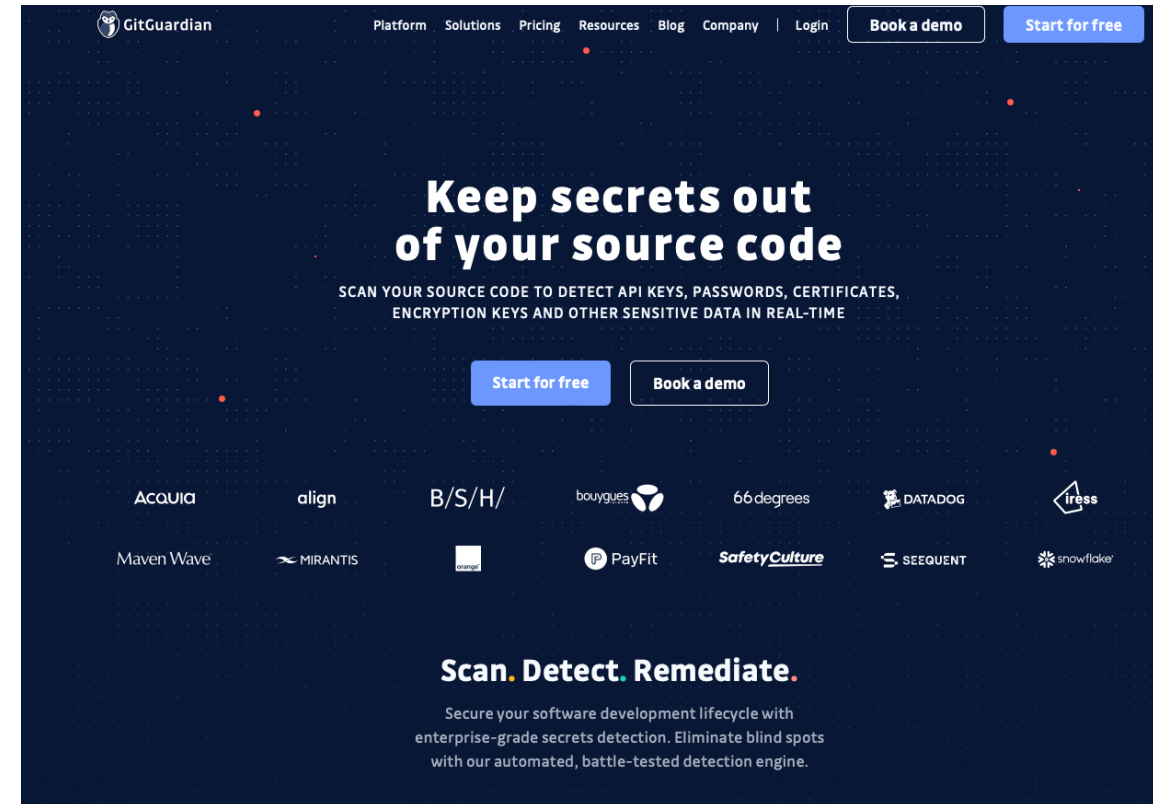
Mechanisms aren't enough: Do developers keep secret keys secret?

- 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 management problem or a tool 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

Elements of a security culture

- Make security a regular part of the process.
  - Include security tools as part of the build/release process
  - Tools may have false positives and false negatives
  - Educate developers about when how to recognize positives that look false, but aren't
  - Include security review as regular part of code review

Wow! That's a lot of High-level stuff

Let's pause a minute …..

Two Views of Security

1. Let's call it Security from High Level first (call it Project-Level Security)
2. **Now, let's look at Security from the App level.**

A Baseline Threat Model {at App Level}

- Concentrate on **technical risks** rather than broad threats.
  - Are there any missing controls?
  - Is there a data flow that can be abused?

- Many technical threats combine to create broad threats.
  - Focusing on vague nation-state attacks and zero-day exploits can overshadow essential application security details.

A Baseline Security Policy {at App Level}

- 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.

- Defining a threat model upfront for the entire system is counter productive.

Basic Structure of Threat Modeling in Agile

An effective threat modeling session must deal with the three primary questions

Activity	Question	Outcome
Explain and explore	What are you building?	A technical diagram
Brainstorm threats	What can go wrong?	A list of technical threats
Prioritize and fix	What are you going to do?	Add prioritized fixes to backlog (todo list)

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

You will learn more about this in the activity posted on the module page!

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?
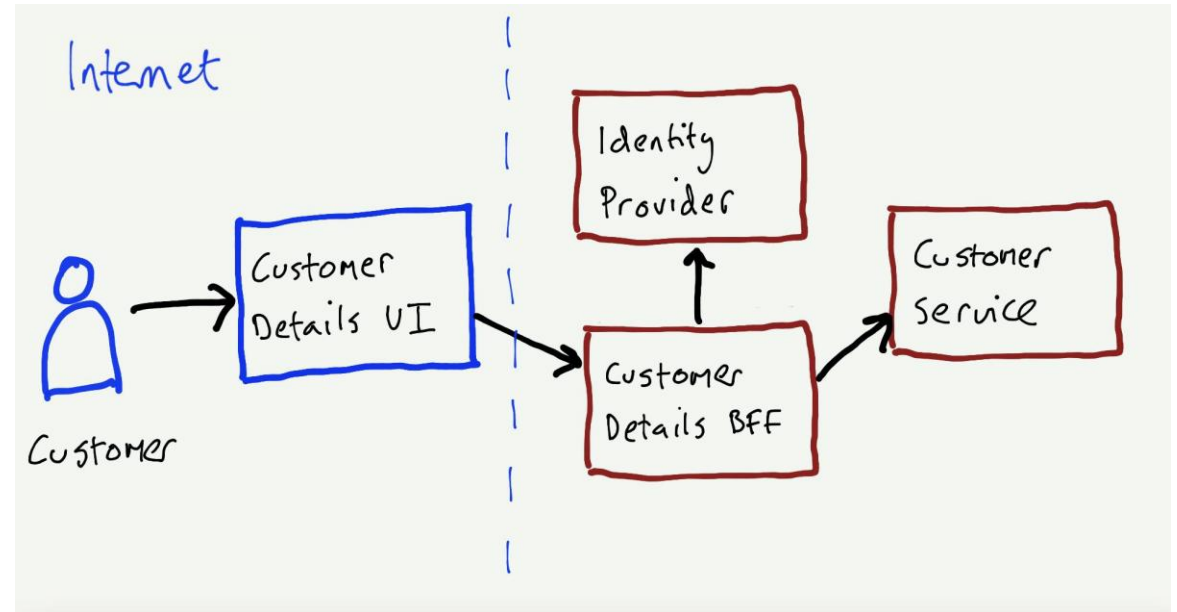 - What are you going to do?

What are we building?

- Use a sketch to represent
  - relevant components
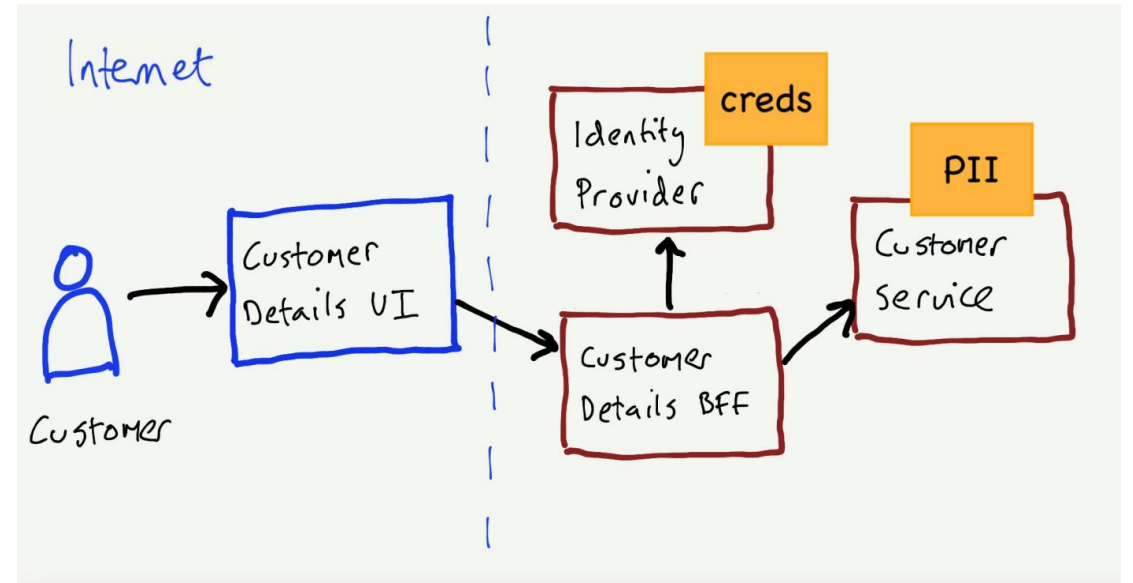 - users that interact with a component
  - collaborative components
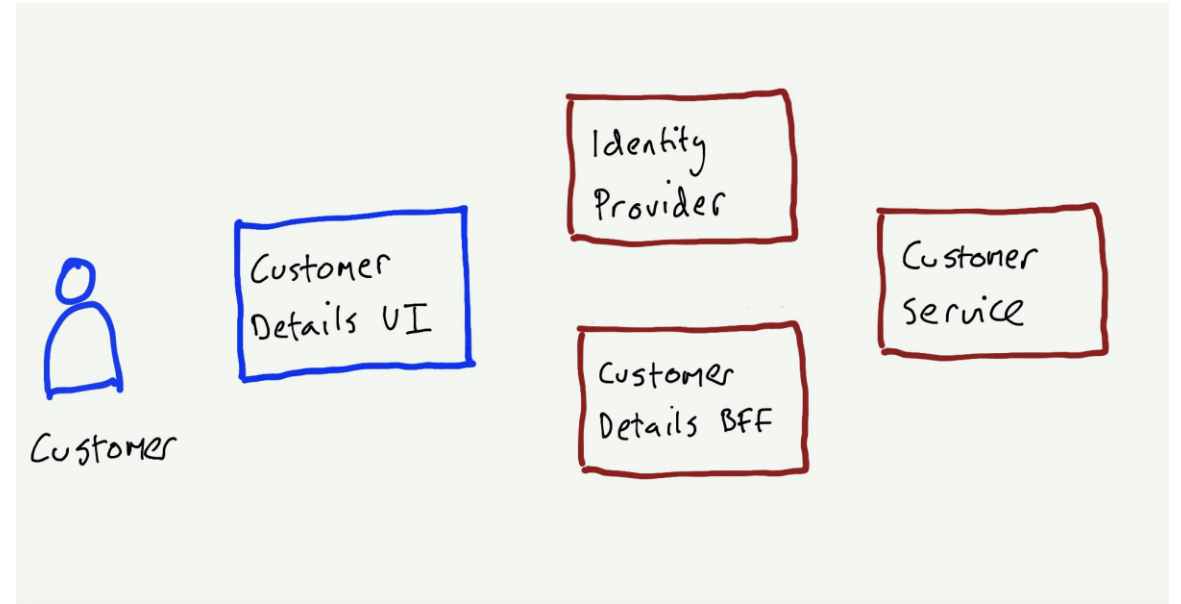
What are we building?

- Explicitly model data flows in the sketch.

- Data flows help show where requests originate (source).

- Label networks and show boundaries between them.
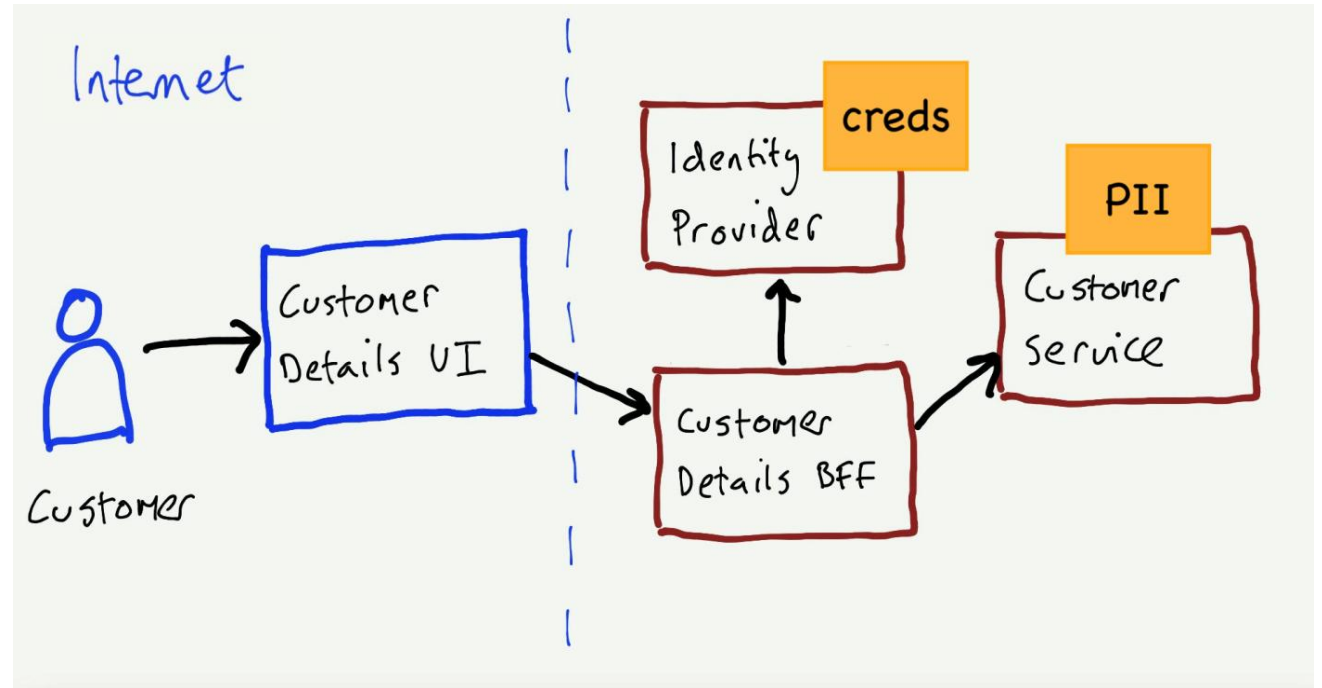
What are we building?

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

What can go wrong?

- Using the STRIDE model, we should identify possible threats that can happen on each flow.
  - Customer -> UI
  - Customer -> Identity Service
  - UI -> BFF
  - BFF -> Customer Service
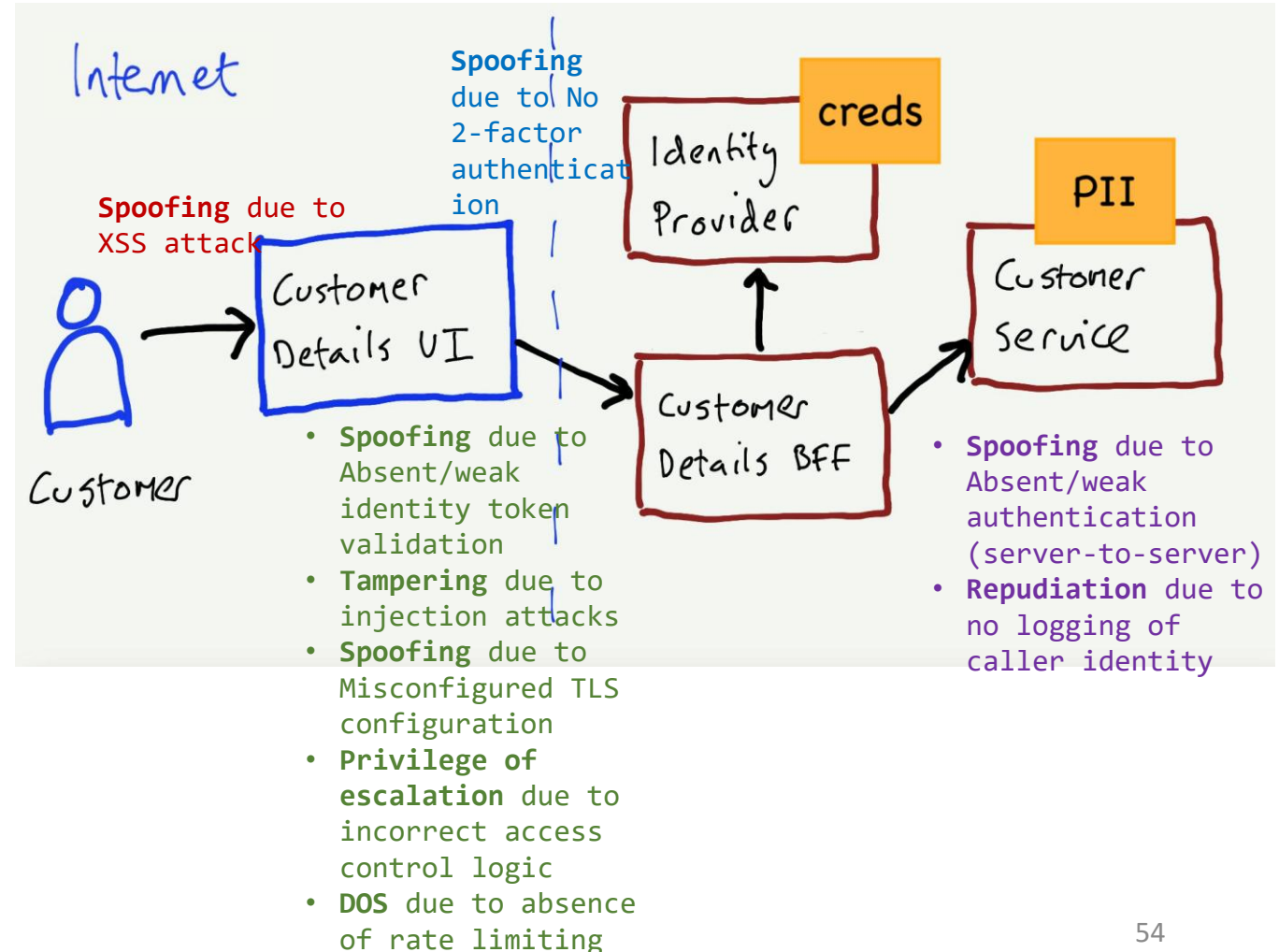
What can go wrong?

- Using the STRIDE model, we should identify possible threats that can happen on each flow.
  - Customer -> UI
  - Customer -> Identity Service
  - UI -> BFF
  - BFF -> Customer Service

Internet

Spoofing due to XSS attack

Spoofing due to No 2-factor authentication

Identity Provider

creds

PII

Customer Service

- **Spoofing** due to Absent/weak identity token validation
- **Tampering** due to injection attacks
- **Spoofing** due to Misconfigured TLS configuration
- **Privilege of escalation** due to incorrect access control logic
- **DOS** due to absence of rate limiting

- **Spoofing** due to Absent/weak authentication (server-to-server)
- **Repudiation** due to no logging of caller identity

What are you going to do?

- 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 of Work Breakdown:
  - Prevent authorization bypass when accessing an API by creating session tokens for the server
  - Prevent XSS attack via user input by sanitizing all inputs
  - Prevent Denial of service for server from the internet by using a rate limit

- **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 why it is important to consider non-functional requirements like security during design
    - Explain 5 common vulnerabilities in web applications and similar software systems, and describe some common mitigations for each of them.
    - Explain the STRIDE Framework for Security by Design