CS 4530
Software Engineering

Lesson 10: Software Engineering & Security

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Lesson

By the end of this lesson, you should be able to…

- Describe that security is a spectrum, and be able to define a realistic threat model for a given system
- Evaluate the tradeoffs between security and costs in software engineering
- Recognize the causes of and common mitigations for common vulnerabilities in web applications
- Utilize static analysis tools to identify common weaknesses in code

Outline:

1. What is a threat model?
2. What are the primary categories of threats for software systems?
3. Techniques for mitigating threats
4. Costs and tradeoffs of mitigations

Security as non-functional requirements

CIA: An overview of security properties

- Confidentiality: is information disclosed to unauthorized individuals?
- Integrity: is code or data tampered with?
- Availability: is the system accessible and usable?

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 your belongings/property?
- Do you change the locks?
- Do you buy security cameras?
- Do you hire a security guard?
- Do you even bother locking the door?

Security is about managing risk

Vocabulary

- Security architecture is a set of mechanisms and policies that we build into our system to mitigate risks from threats
- Threat: potential event that could compromise a security requirement
- Attack: realization of a threat
- 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 is about managing risk

Cost of attack vs cost of defense?

- 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
- So: How likely do we think we are to be attacked in way **X**?

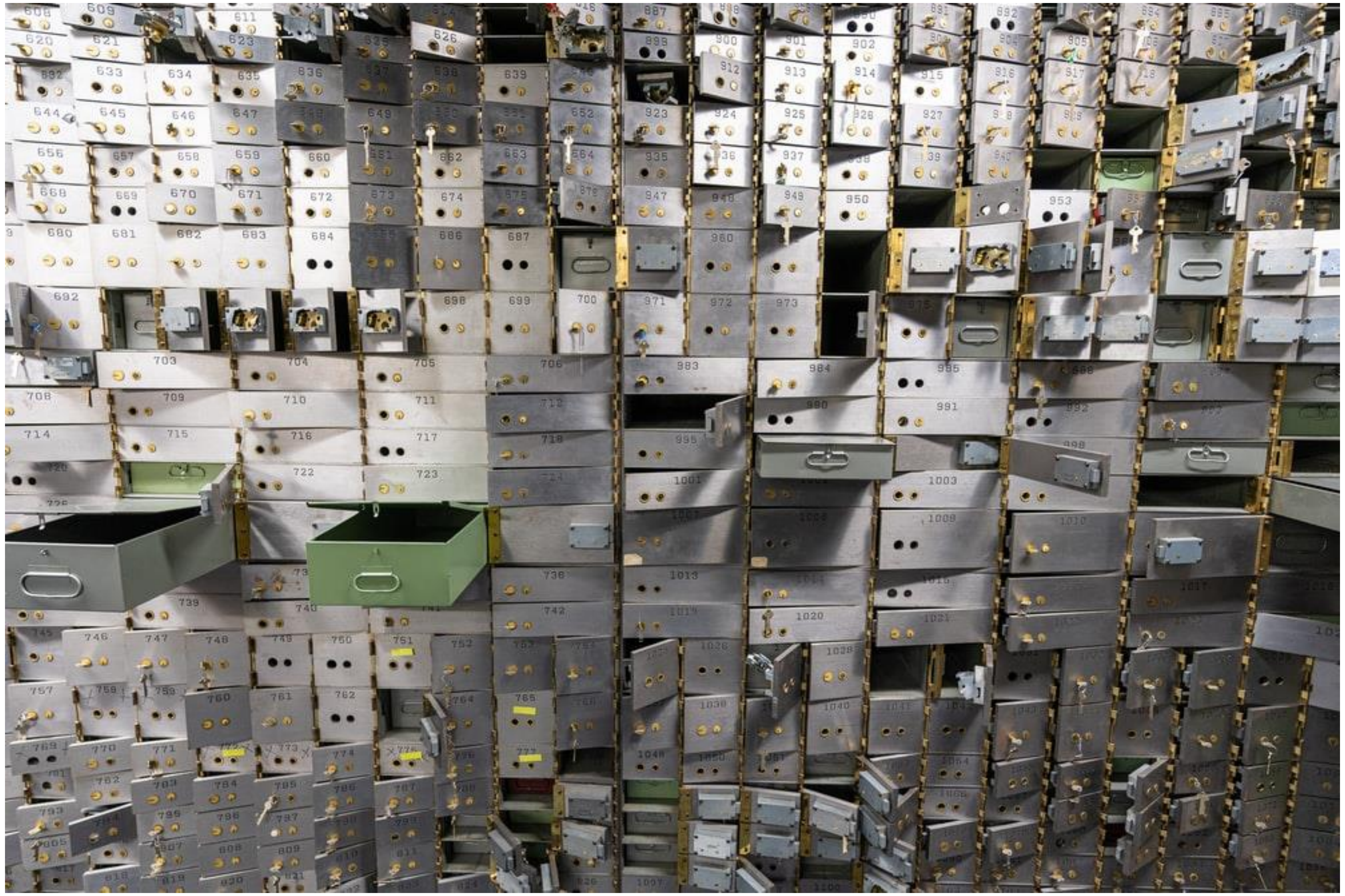
A Threat Model forces us to answer 3 key questions:

- What is important to defend?
- Who do we trust?
- What processes do we institute to protect our code and data?

Thread Model: What is important to defend?

What value can an attacker extract from a vulnerability?

- What is being defended?
  - What resources are important to defend?
- Does our code contain any sensitive data?
- What is the cost if that data is breached or tampered with?
  - Even if your code is not “sensitive”: does it expose other routes of attack?



Threat Model: Who do we trust?

- What entities or parts of system can be considered secure and trusted?
- Have to trust **something**!
- Never trust remote users (especially remote users!)

Threat Model: Processes

- What processes do we institute to protect our code and data?
  - How often do we review our code for security?
  - How often do we review our partners’ security practices?

Creating a Reasonable Threat Model

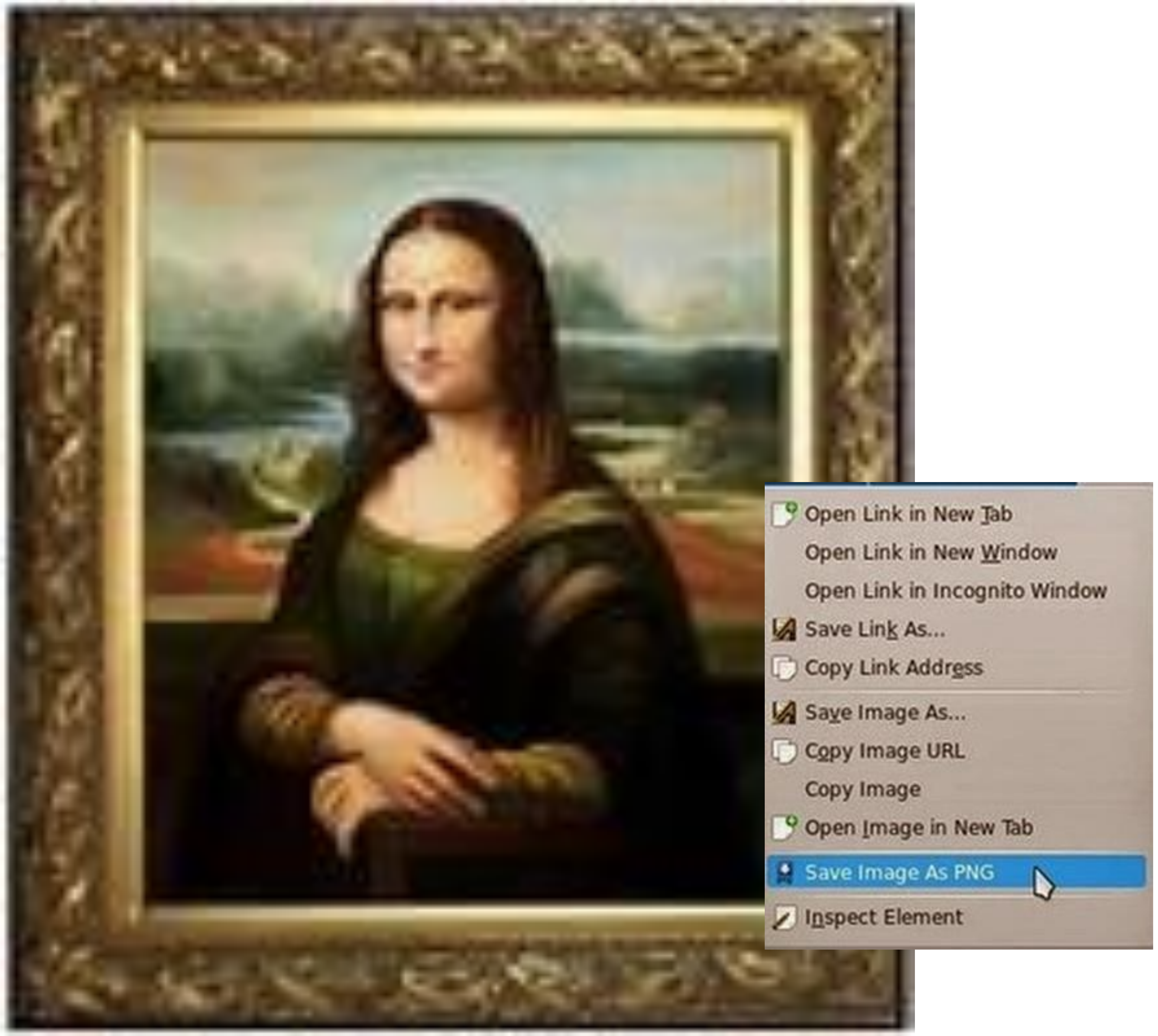
Best practices applicable in most situations

- Trust:
  - Developers writing our code
  - Server running our code
  - Popular dependencies that we use and update
- Don’t trust:
  - Code running in browser
  - Inputs from users
- Practice good security practices:
  - Encryption (all data in transit, sensitive data at rest)
  - Code signing, multi-factor authentication
- Bring in security experts early for riskier situations

Part 2: Categories of Threats

1. Code that runs in an untrusted environment
2. Untrusted data flowing into our trusted codebase
3. Threats coming from the software supply chain (dependency on untrusted code)

Threat: code that runs in an untrusted environment



Threat: Code that runs in an untrusted environment

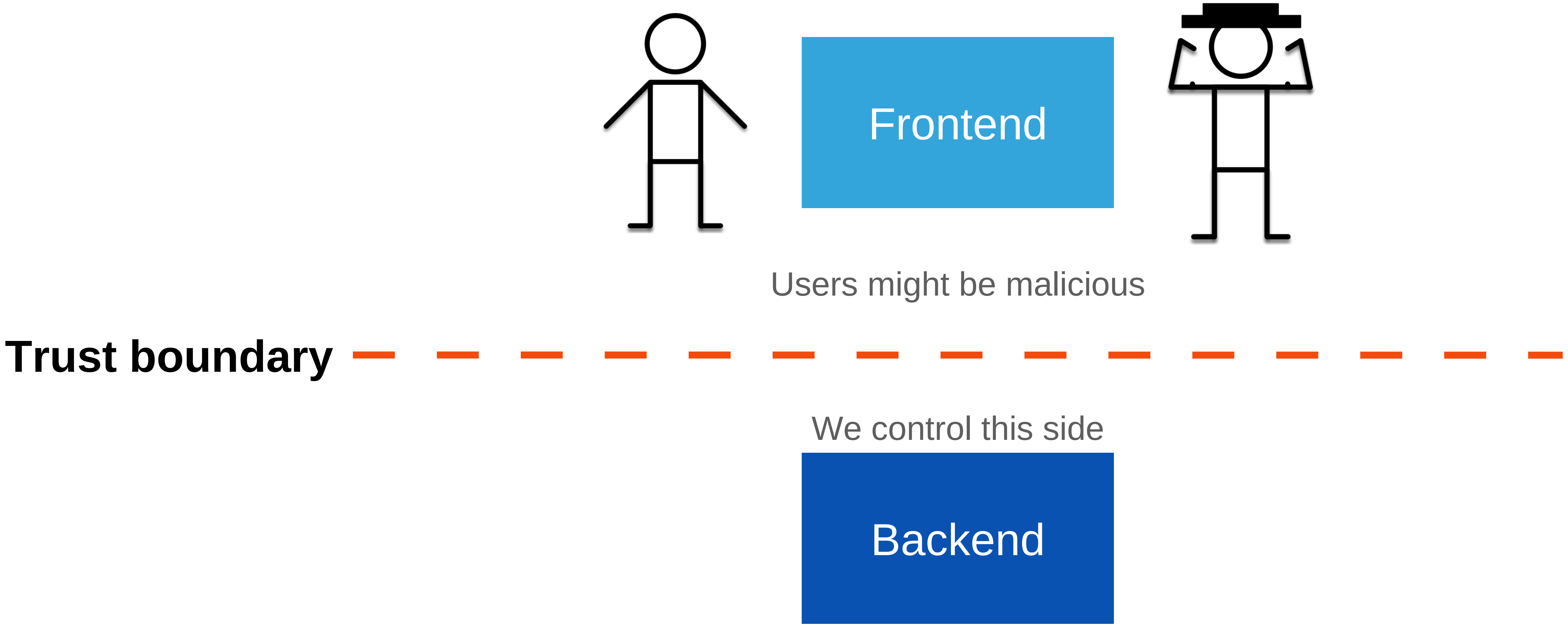
Authentication code in a web application

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

Should this go in our frontend code?

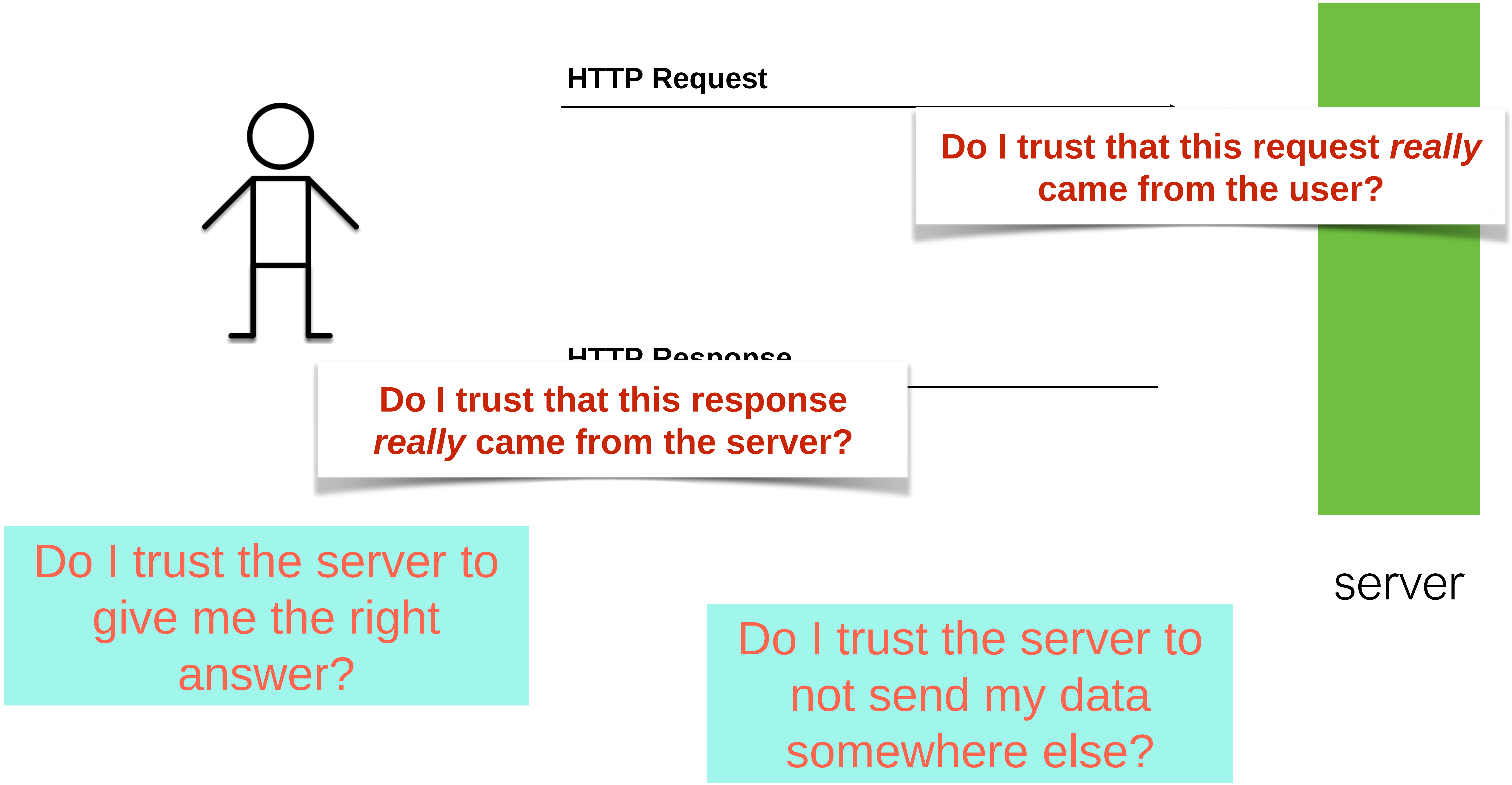
Threat: Code that runs in an untrusted environment

Authentication code in a web application

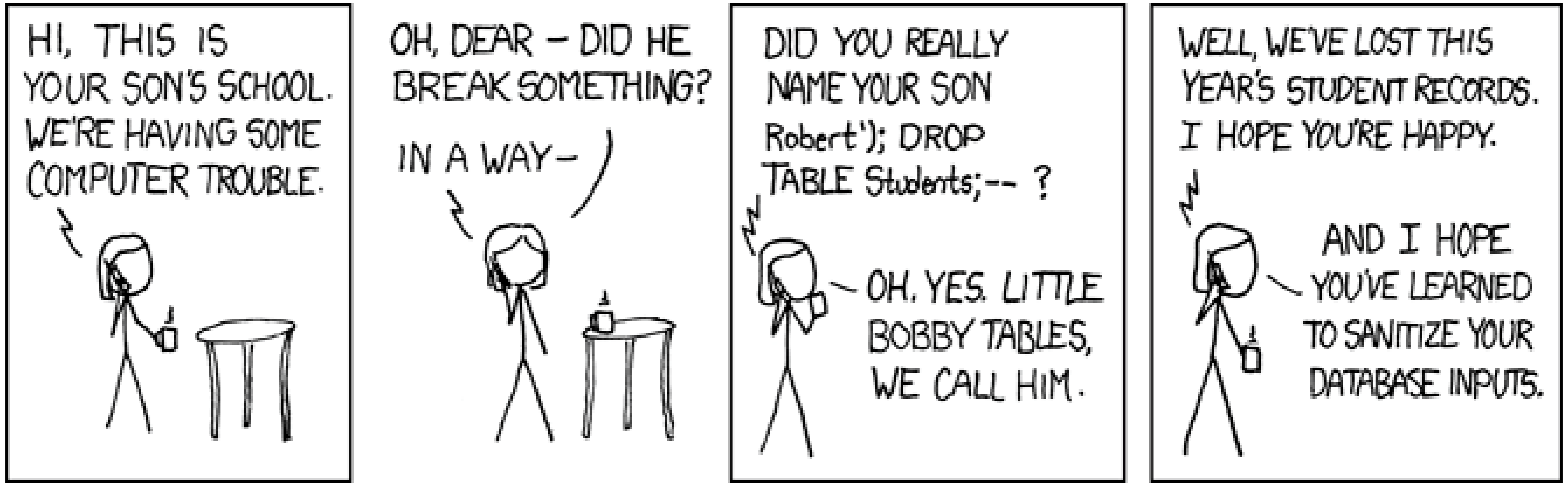



```
function checkPassword(inputPassword: string){
  if(inputPassword === 'letmein'){
    return true;
  }
  return false;
}
```

Threat: Code that runs in an untrusted environment



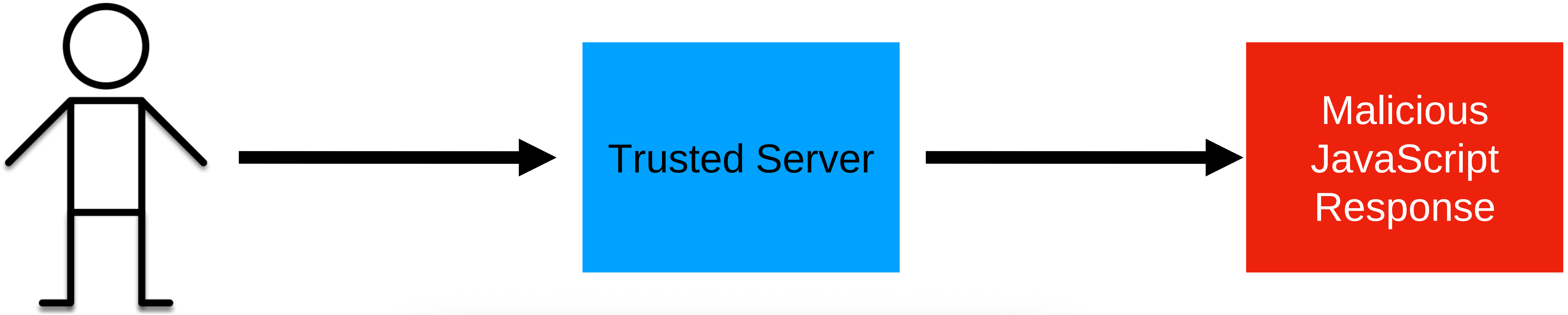
Threat: Data controlled by a user flowing into our trusted codebase

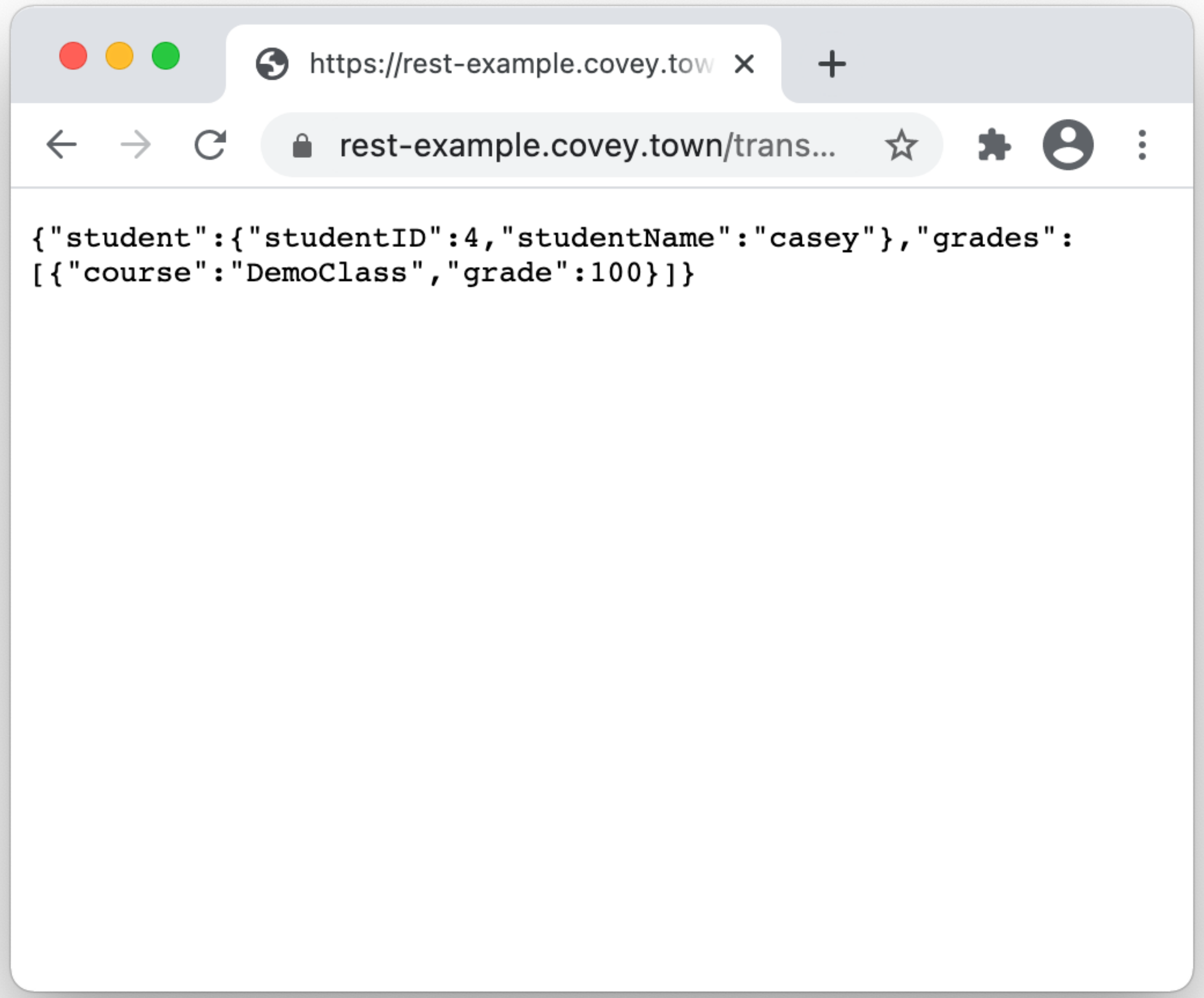


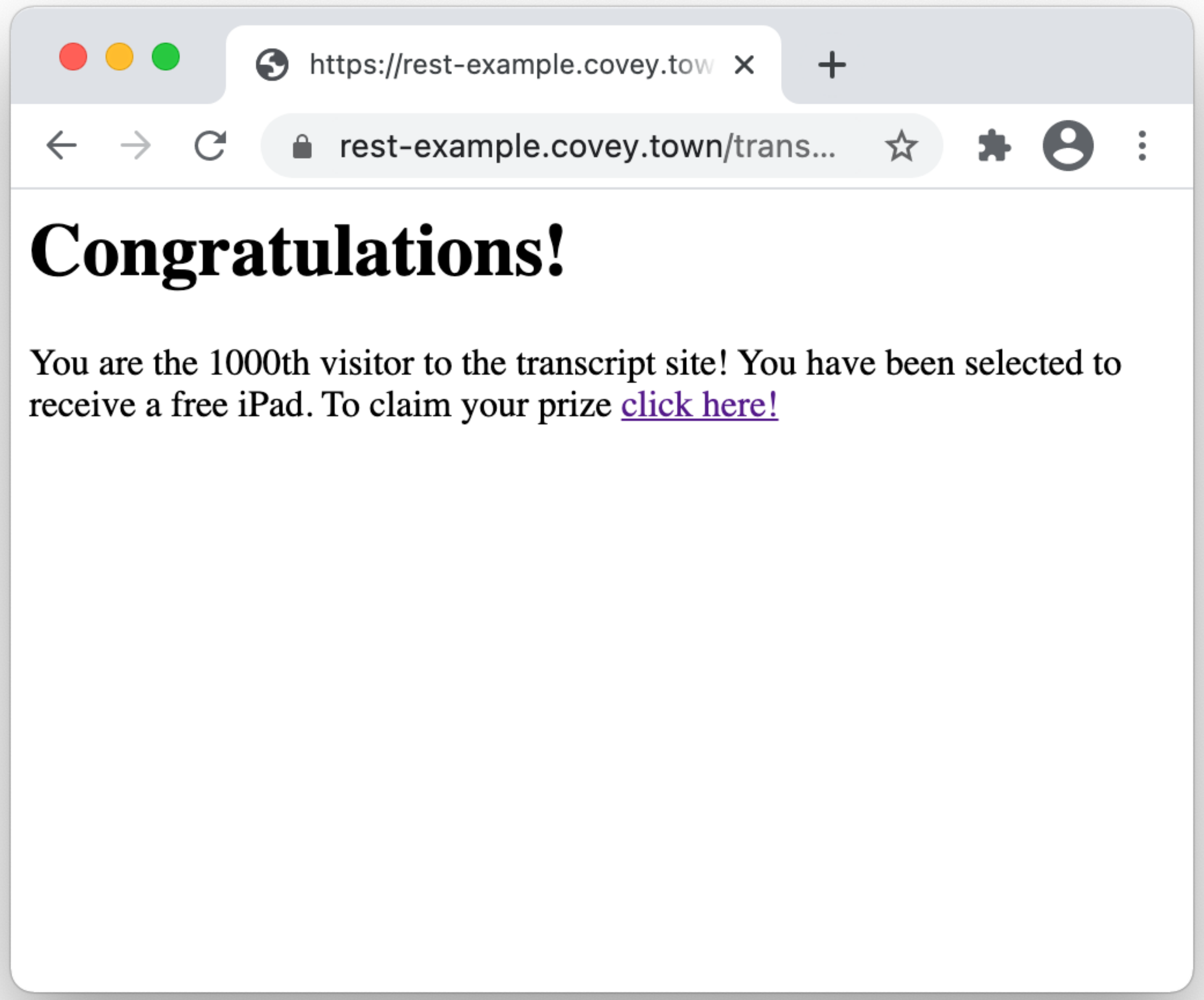
https://xkcd.com/327/

Threat: Data controlled by a user flowing into our trusted codebase

Cross-site scripting (XSS)

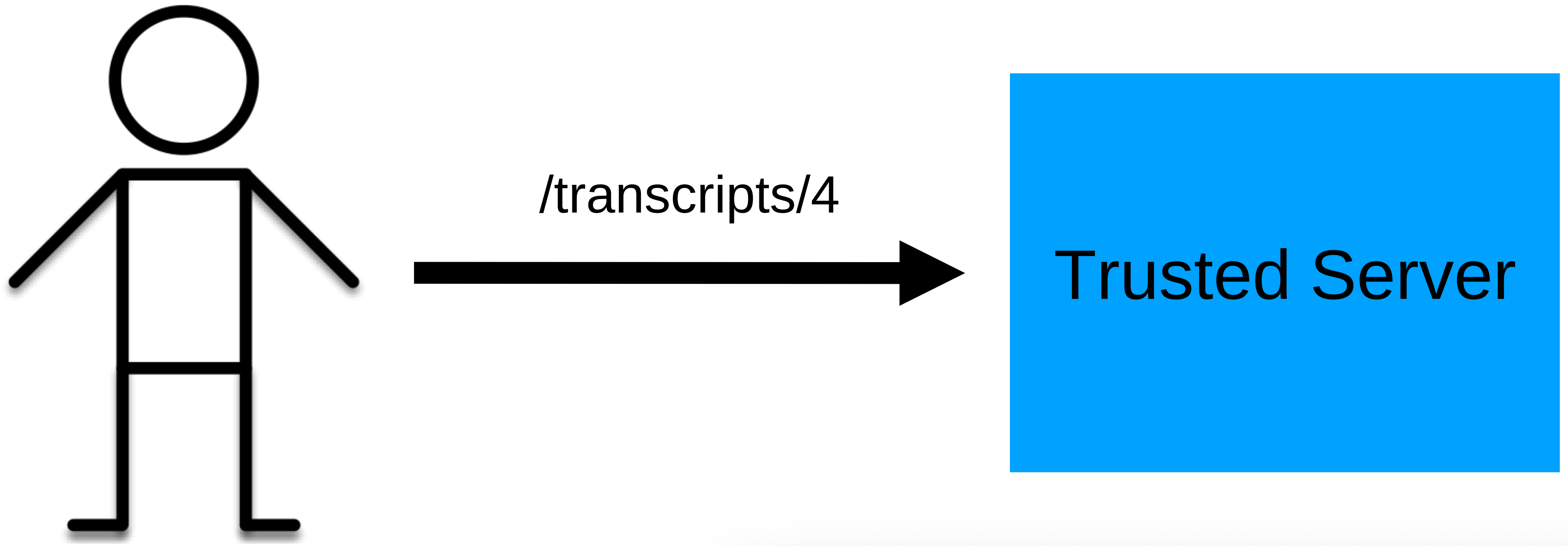




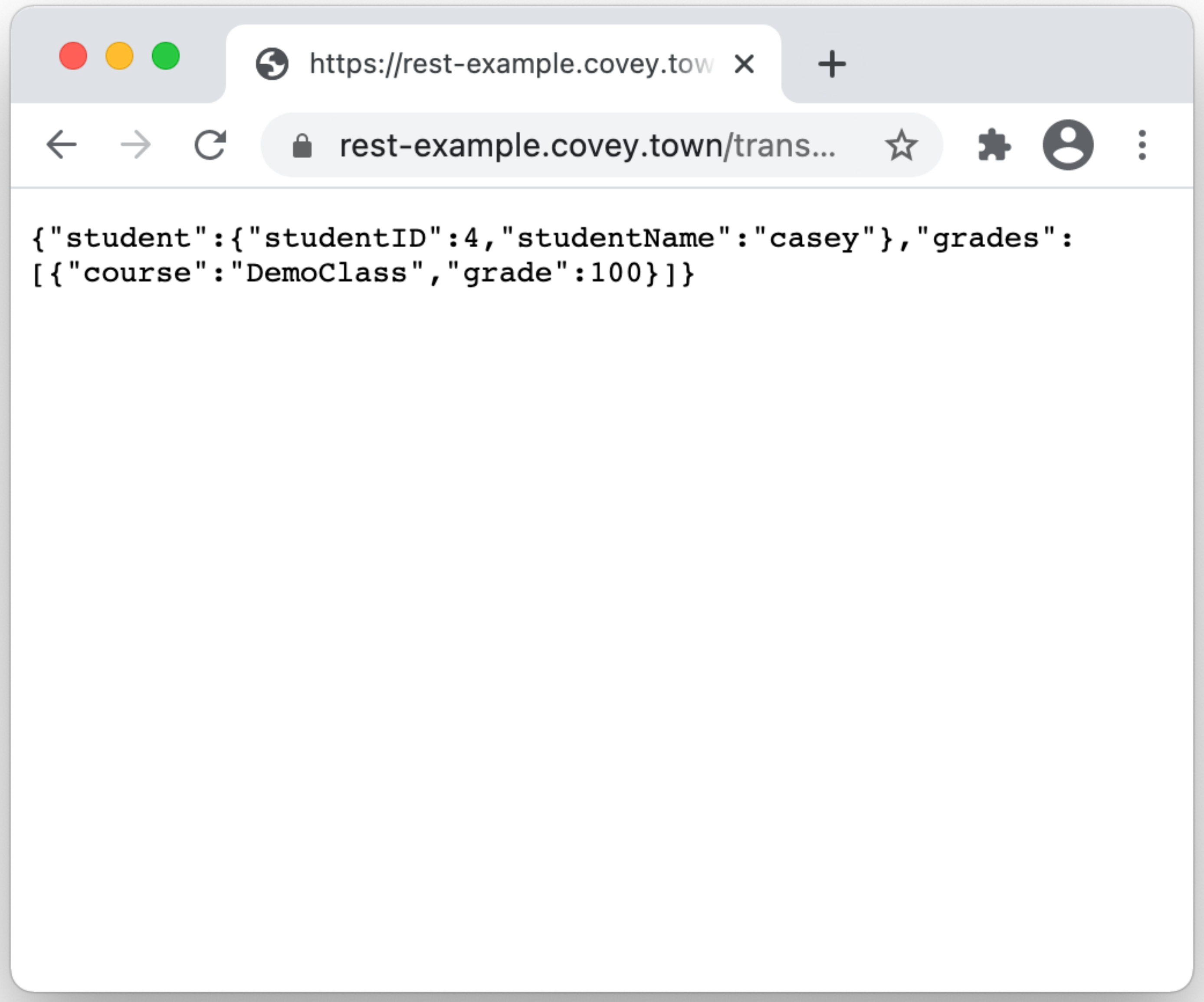


Threat: Data controlled by a user flowing into our trusted codebase

Cross-site scripting (XSS)

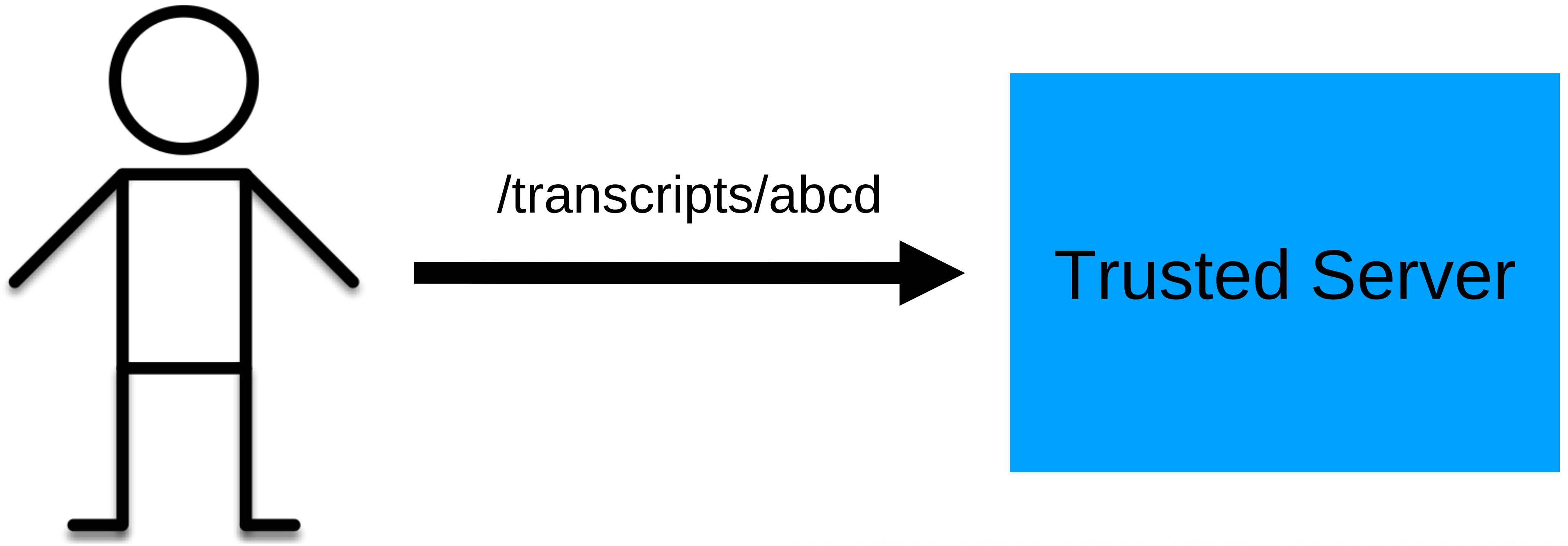



```
app.get('/transcripts/:id', (req, res) => {
  // req.params to get components of the path
  const {id} = req.params;
  const theTranscript = db.getTranscript(parseInt(id));
  if (theTranscript === undefined) {
    res.status(404).send(`No student with id = ${id}`);
  }
  {
    res.status(200).send(theTranscript);
  }
});
```

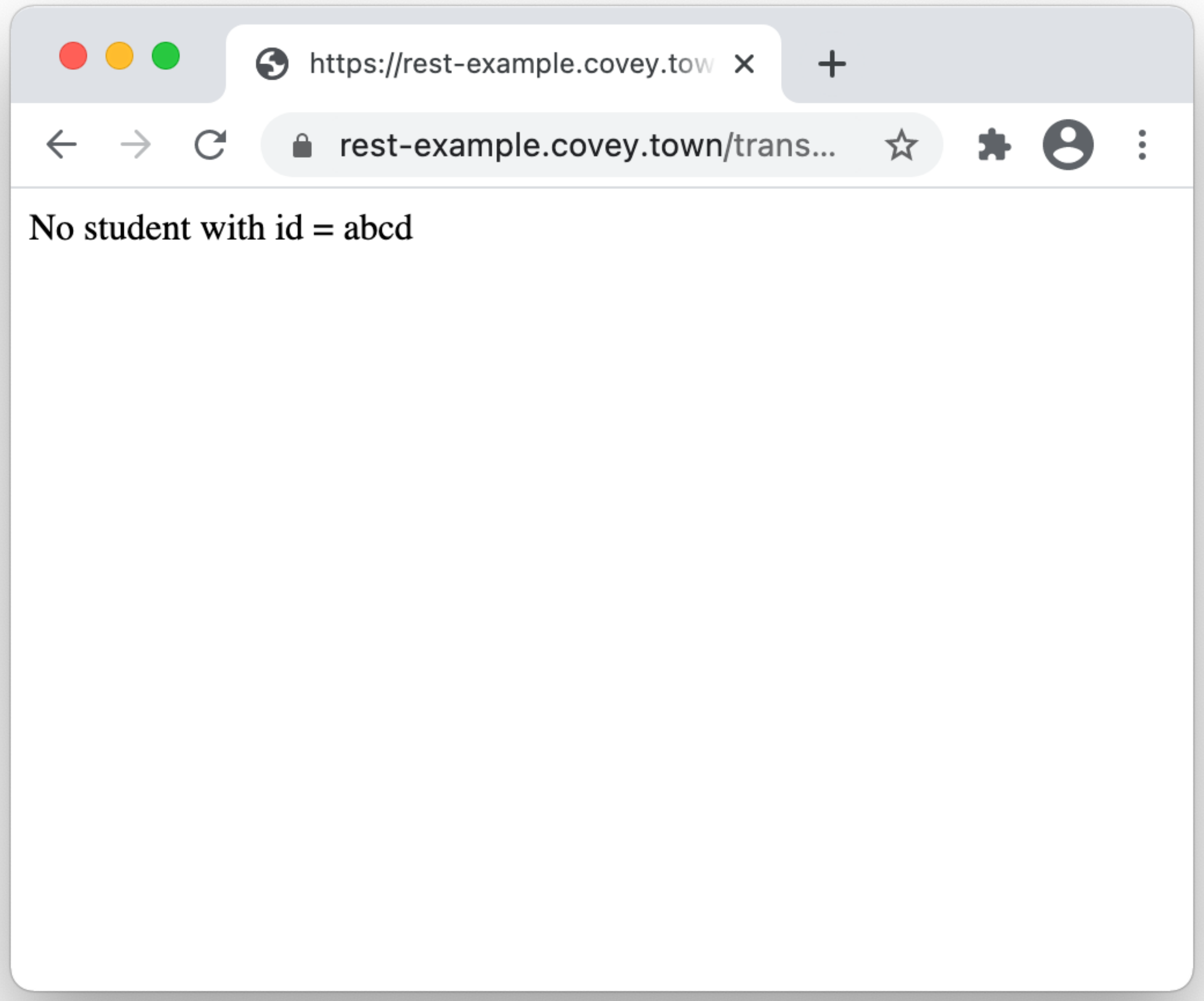


Threat: Data controlled by a user flowing into our trusted codebase

Cross-site scripting (XSS)

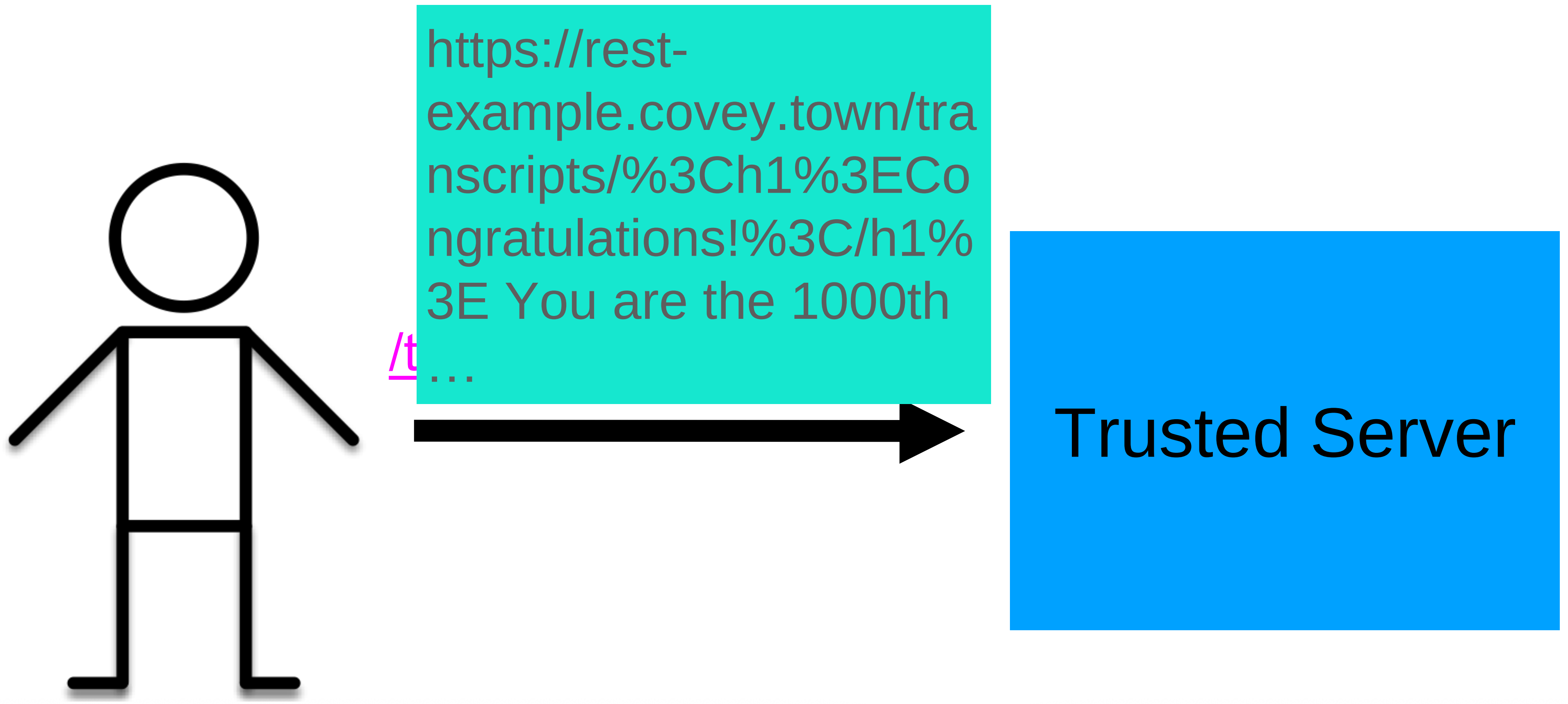



```
app.get('/transcripts/:id', (req, res) => {
  // req.params to get components of the path
  const {id} = req.params;
  const theTranscript = db.getTranscript(parseInt(id));
  if (theTranscript === undefined) {
    res.status(404).send(`No student with id = ${id}`);
  }
  {
    res.status(200).send(theTranscript);
  }
});
```



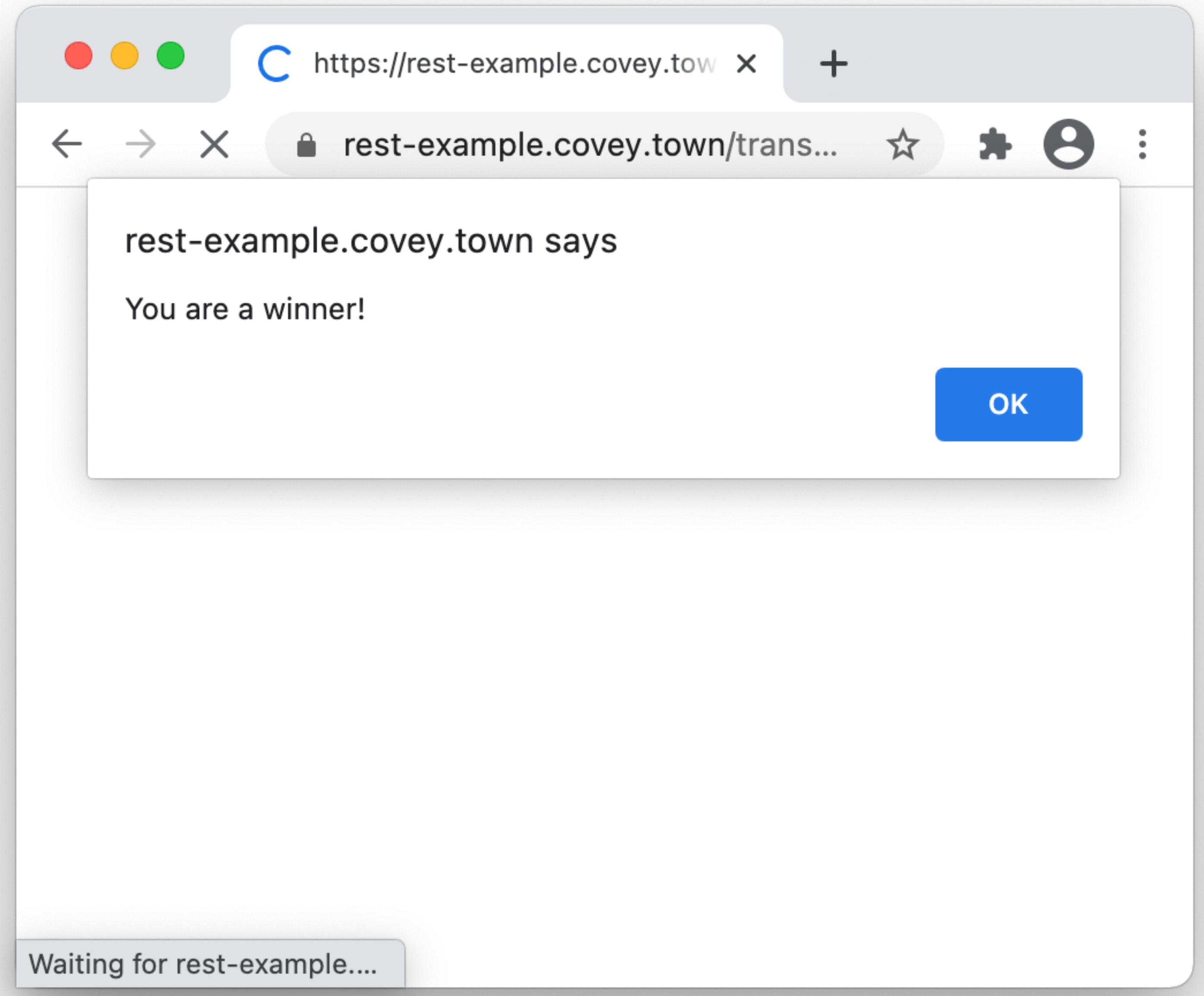
Threat: Data controlled by a user flowing into our trusted codebase

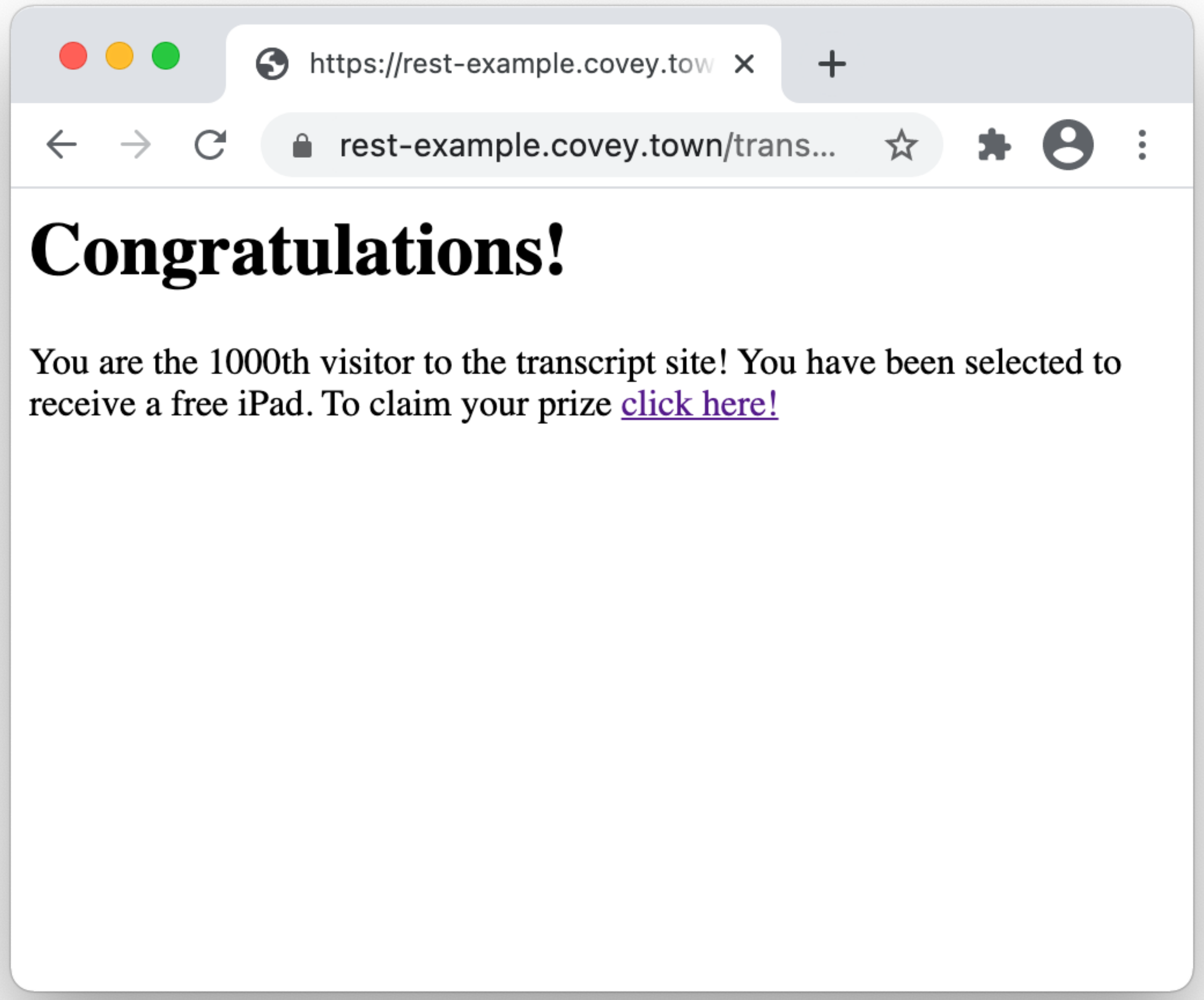
Cross-site scripting (XSS)




```
app.get('/transcripts/:id', (req, res) => {
  // req.params to get components of the path
  const {id} = req.params;
  const theTranscript = db.getTranscript(parseInt(id));
  if (theTranscript === undefined) {
    res.status(404).    d(`No student with id = ${id}`);
  }
  {
    res.status(200         theTranscript);
  }
});
```

```
<h1>Congratulations!</h1>
  You are the 1000th visitor to the transcript site! You have been selected to receive a free iPad. To claim your prize <a href='https://www.youtube.com/watch?v=DLzxrzFCyOs'>click here!</a>
  <script language="javascript">
document.getRootNode().body.innerHTML=
'<h1>Congratulations!</h1>You are the 1000th visitor to the transcript site! You have been selected to receive a free iPad. To claim your prize <a href="https://www.youtube.com/watch?v=DLzxrzFCyOs">click here!</a>';
alert('You are a winner!');
</script>
```





Threat: Data controlled by a user flowing into our trusted codebase

Java code injection in Apache Struts (@Equifax)

NEWS

Equifax Says Cybersecurity Breach Has Cost $1.4 Billion

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.

Threat: Data controlled by a user flowing into our trusted codebase

Java code injection in Log4J

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-basec

execute malicious code a

systems.

networks between May and February in a "deliberate campaign" that reflects new attack vectors and retooling by the prolific Chinese state-sponsored group.

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. Note that 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

https://duo.com/decipher/apt41-compromised-six-state-government-networks

Threat Category 3: Software Supply Chain

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

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

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

Photo Illustration by Grayson Blackmon / The Verge

PODCASTS

HARD LESSONS OF THE SOLARWINDS HACK

Cybersecurity reporter Joseph Menn on the massive breach the US didn't see coming

By Nilay Patel | @reckless | Jan 26, 2021, 9:13am EST

SHARE

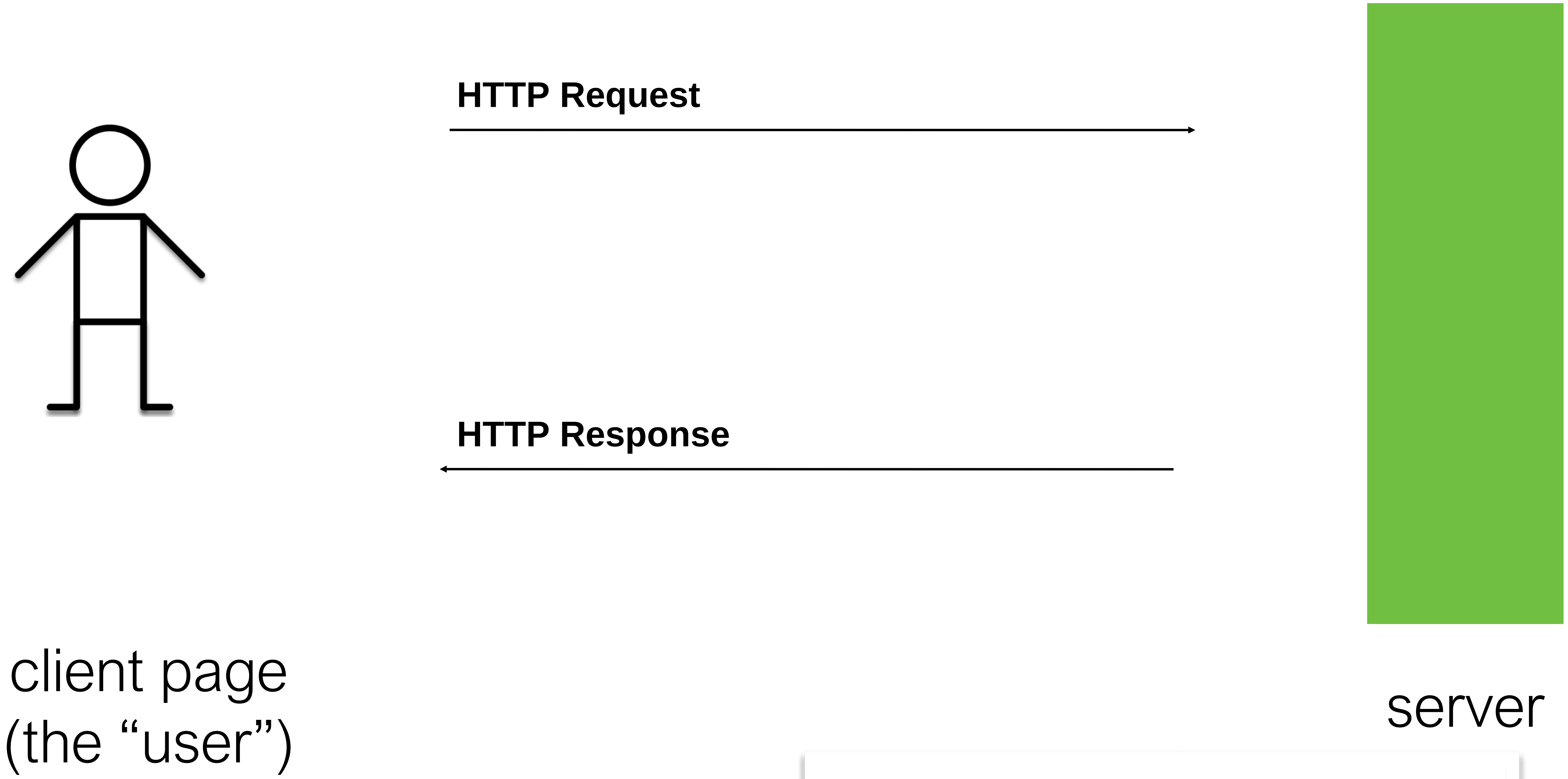
In December, details came out on one of the most massive breaches of US cybersecurity in recent history. A group of hackers, likely from the Russian government, had gotten into a network management company called SolarWinds and infiltrated its cu... to breach ever...

https://www.theverge.com/2021/1/26/22248631/solarwinds-hack-cybersecurity-us-menn-decoder-podcast

Part 3: Mitigating security threats in software engineering

- For these threats:
  - **Threat category: Code that runs in an untrusted environment**
  - Threat category: Inputs that are controlled by an untrusted user
  - Threat category: Software supply chain
- Recurring theme: No silver bullet

Threat Mitigation: Trusted Code

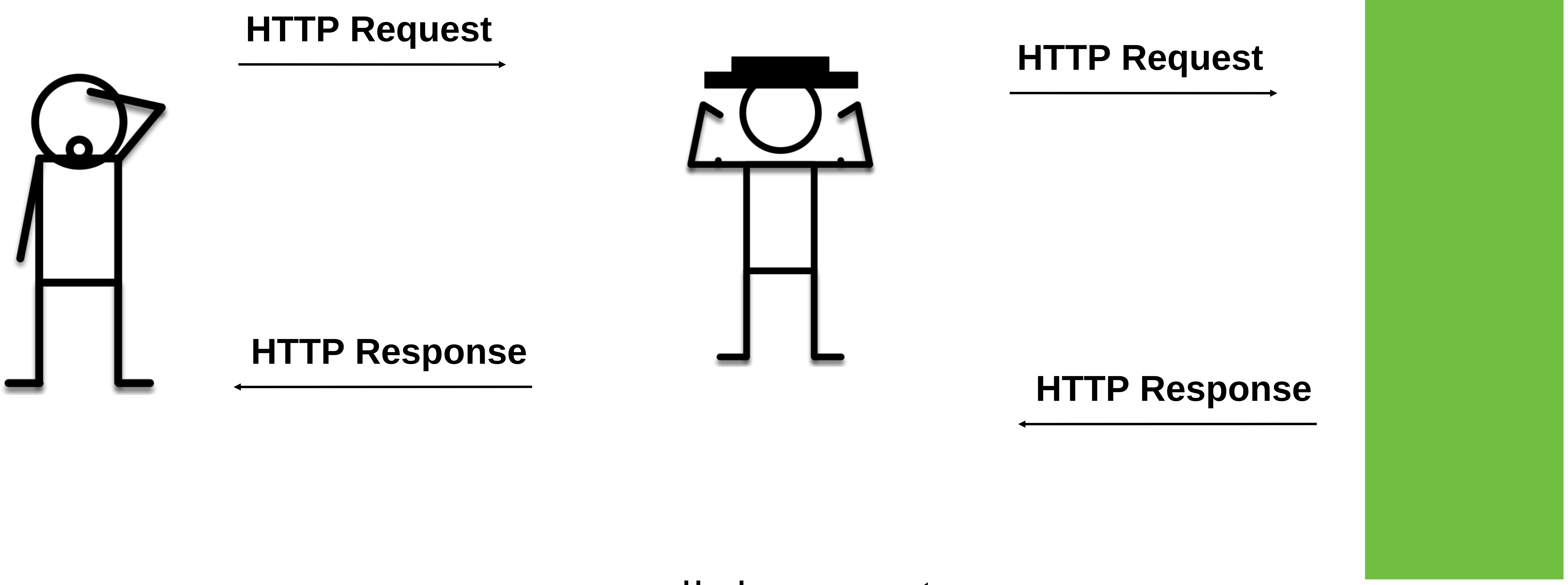


Do I trust that this response *really* came from the server?

Do I trust that this request *really* came from the user?

Threat Mitigatic e

Might be "man in the middle" that intercepts requests and impersonates user or server.



client page
(the "user")

malicious actor
"black hat"

server

Do I trust that this response *really* came from the server?

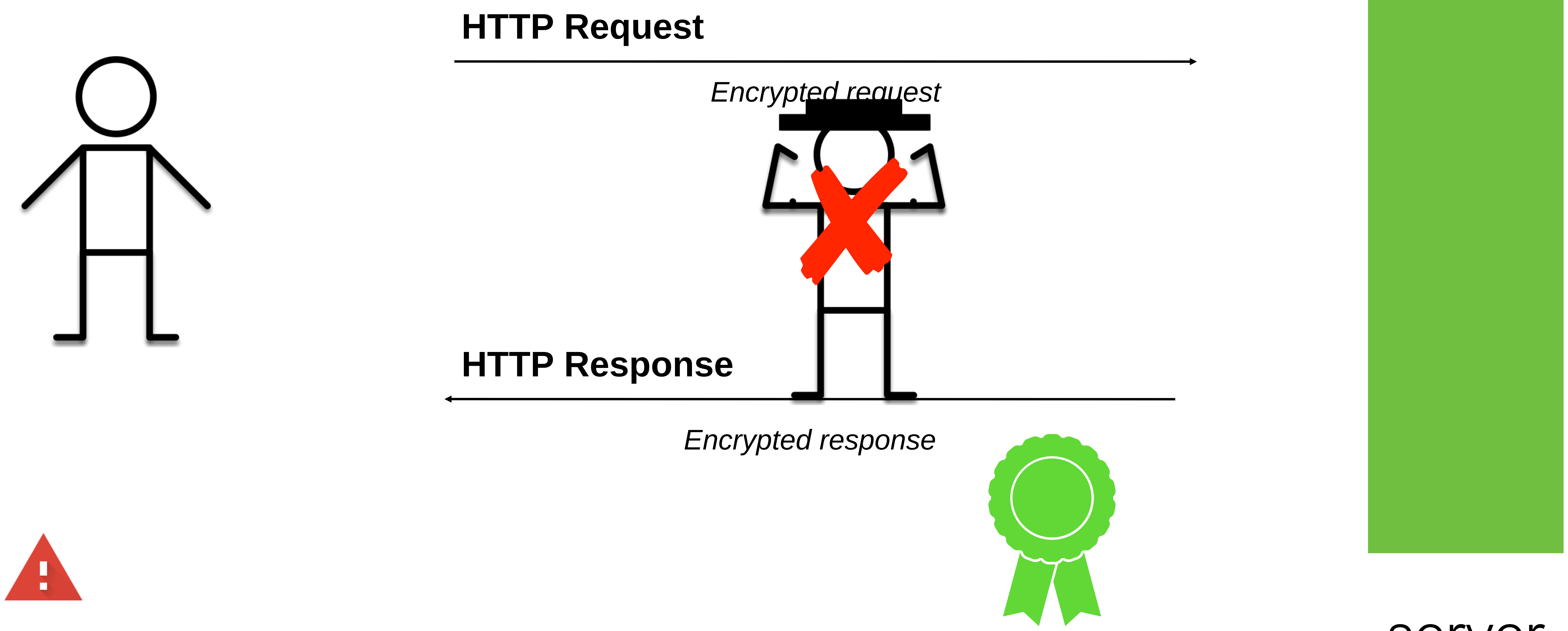
Do I trust that this request *really* came from the user?

Threat Mitigation: Trusted Code

Preventing the man-in-the-middle with SSL

Threat Mitigation: Trusted Code

Preventing the man-in-the-middle with SSL



server

Your connection is not private

Attackers might be trying to steal your information from **192.168.18.4** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

amazon.com certificate
(AZ’s public key + CA’s sig)

SSL: A perfect solution?

Certificate authorities

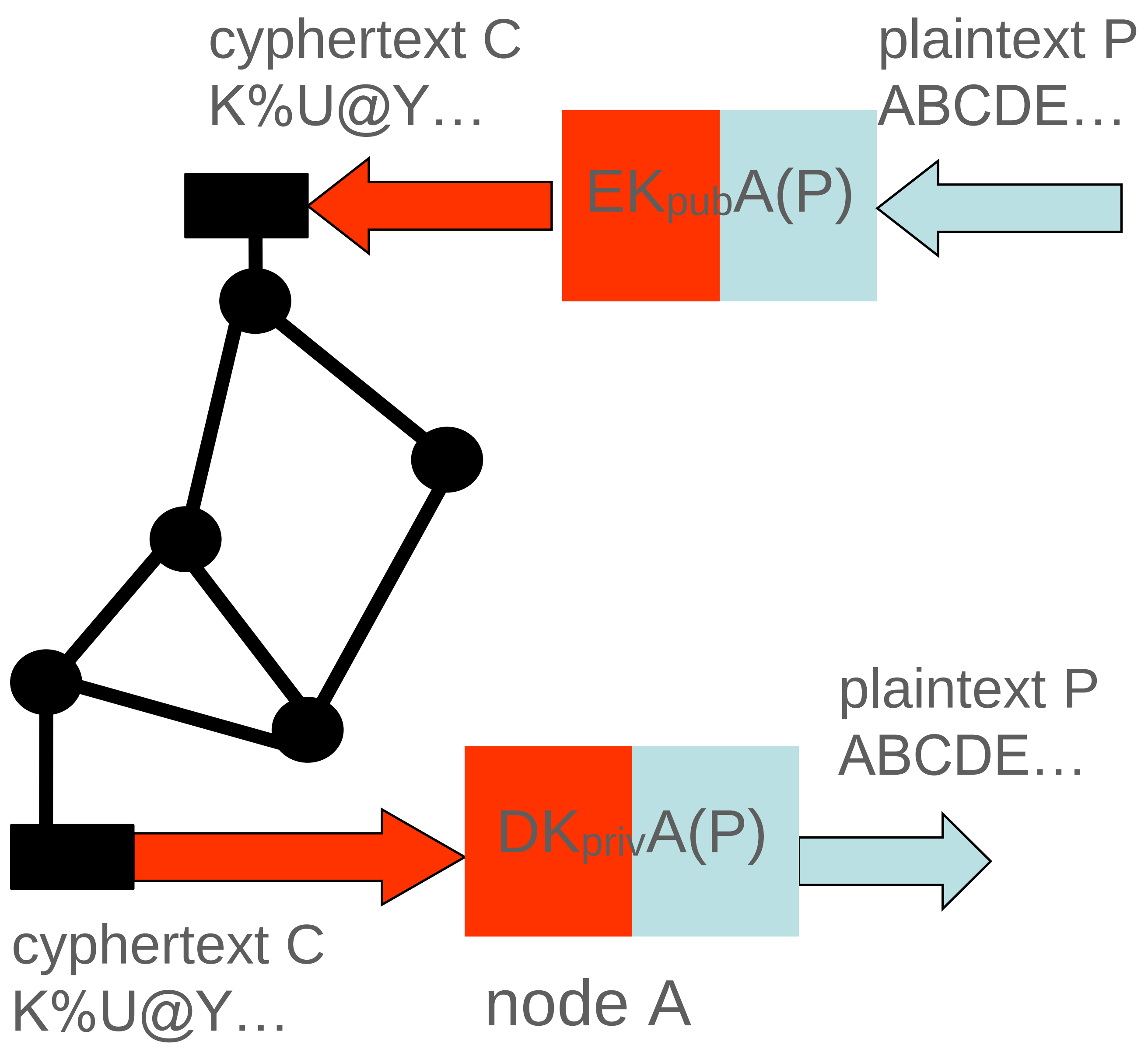
- A certificate authority (or CA) binds some public key to a real-world entity that we might be familiar with
- The CA is the clearinghouse that verifies that amazon.com is truly amazon.com
- CA creates a certificate that binds amazon.com's public key to the CA’s public key (signing it using the CA’s private key)

Asymmetric encryption: aka public key/private key

- Each actor creates two keys:
  - A public key, which it publishes. This tells the world: if you want to send a private message to me, encrypt it with this key. Then only I can decode it.
  - A private key, which it keeps secret. The private key is what the actor uses to decode the messages that are processed by the public key.

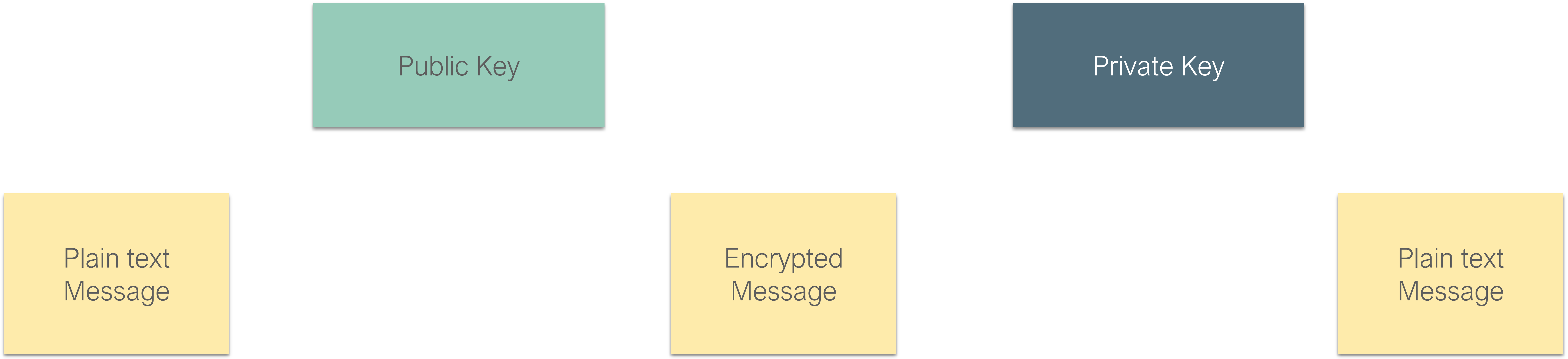
Asymmetric encryption, aka public key/private key

- When a node B wants to send a message to node A, it obtains A's public key and uses it to encrypt the message
- Only A can decrypt the message using its private key
- Computationally expensive; usually only used for authentication



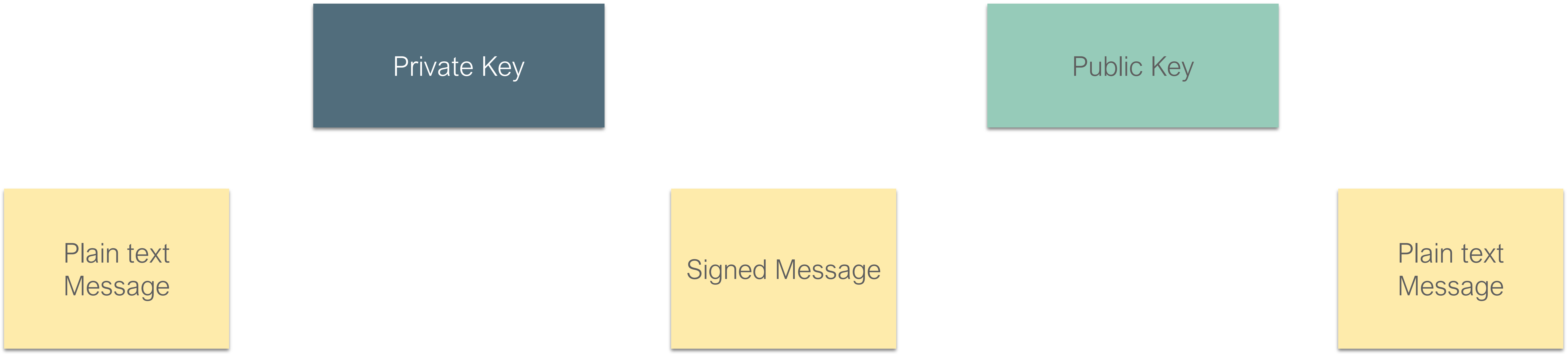
Public/Private Key Encryption

- Encrypt with public key: only private key holder can decrypt

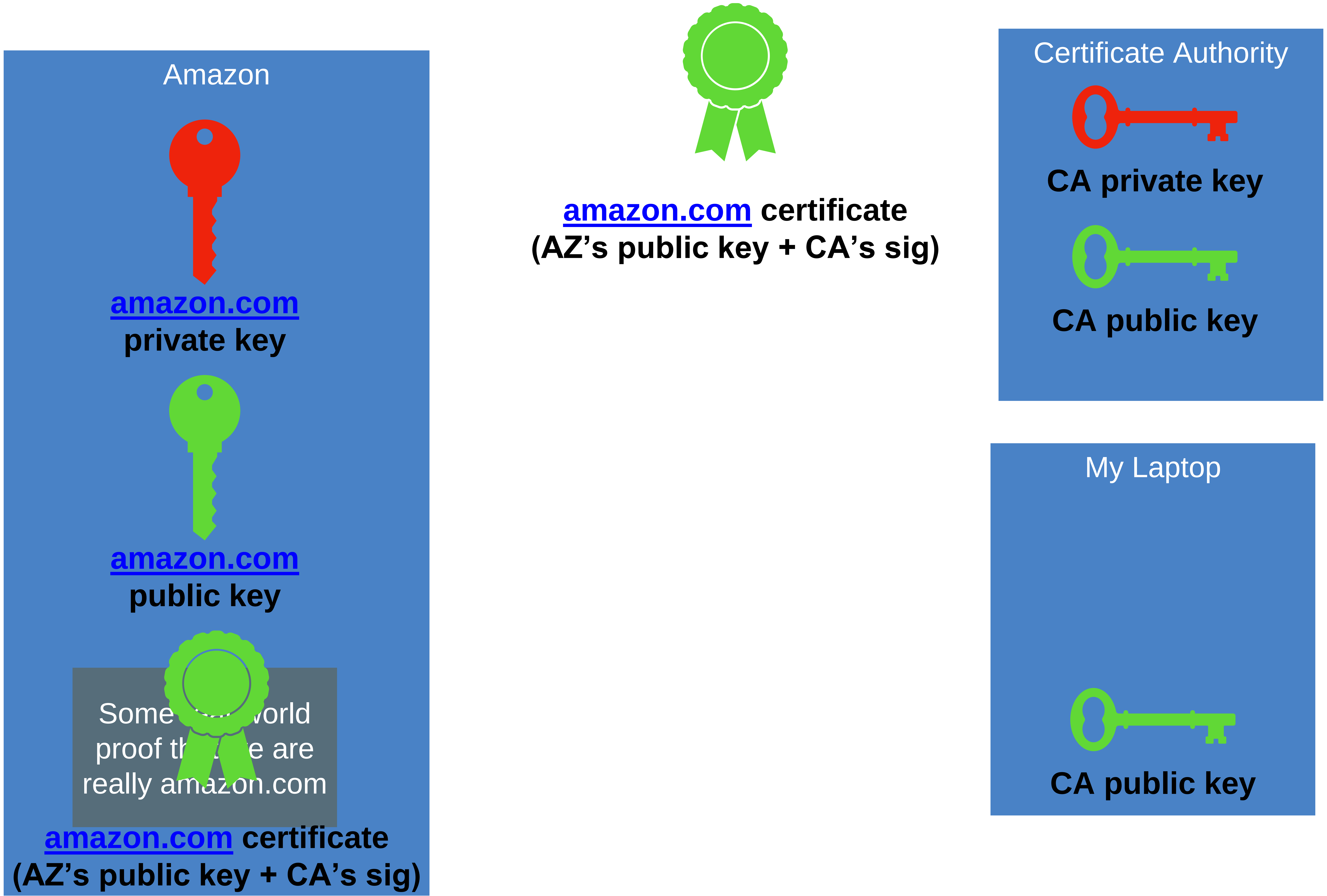


asymmetric encryption can be used for authentication, too

- Encrypt with private key: anyone with public key can decrypt and be confident about who sent it.

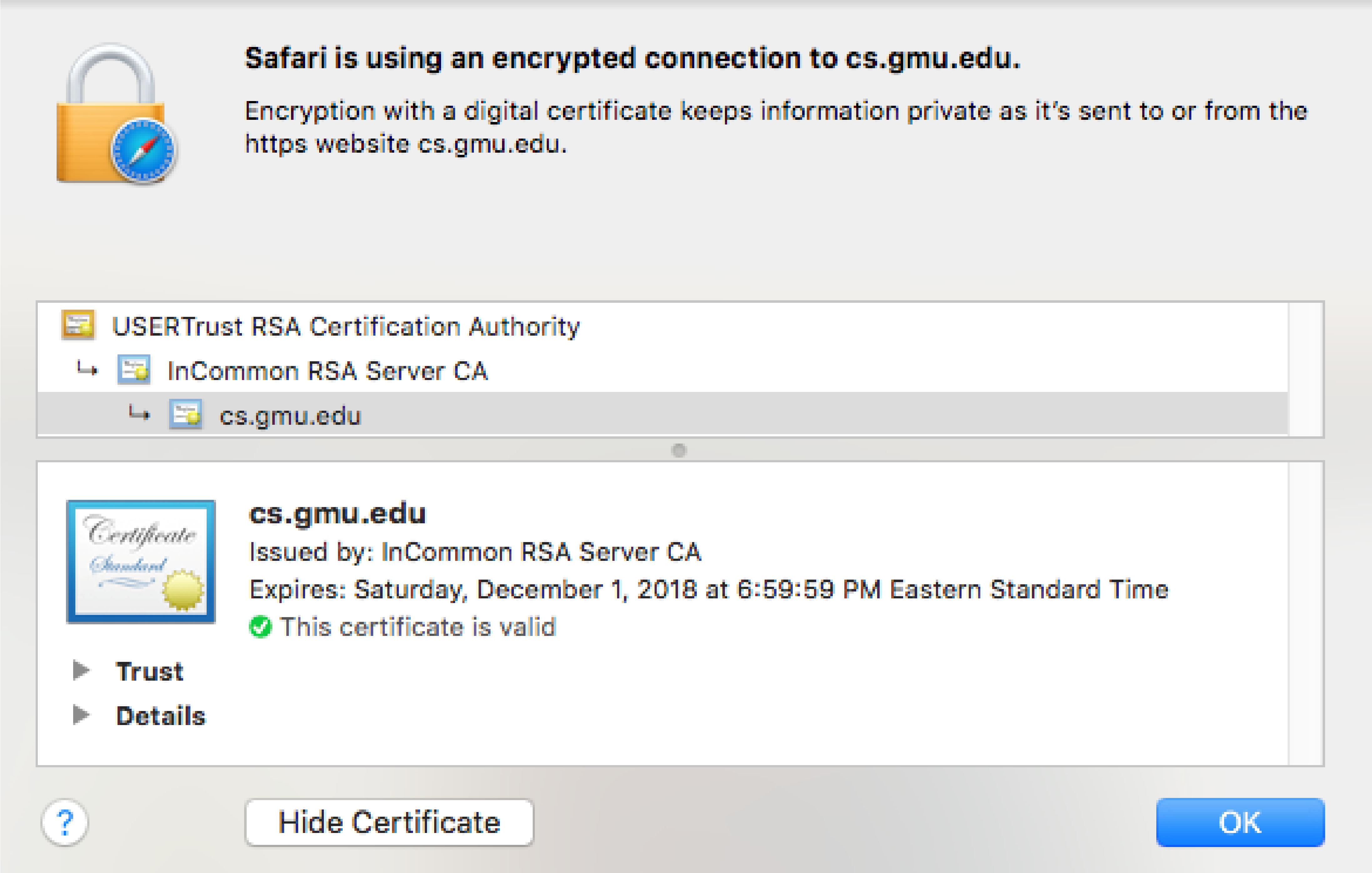


Certificate Authorities issue SSL Certificates



Certificate Authorities are Implicitly Trusted

- Note: We had to already know the CA's public key
- There are a small set of “root” CA’s (think: root DNS servers)
- Every computer/browser is shipped with these root CA public keys



Should Certificate Authorities be Implicitly Trusted?

Signatures only endorse trust if you trust the signer!

- What happens if a CA is compromised, and issues invalid certificates?
- Not good times.

Security

Comodo-gate hacker brags about forged certificate exploit

Tiger-blooded Persian cracker boasts of mighty exploits

Security

Fuming Google tears Symantec a new one over rogue SSL certs

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

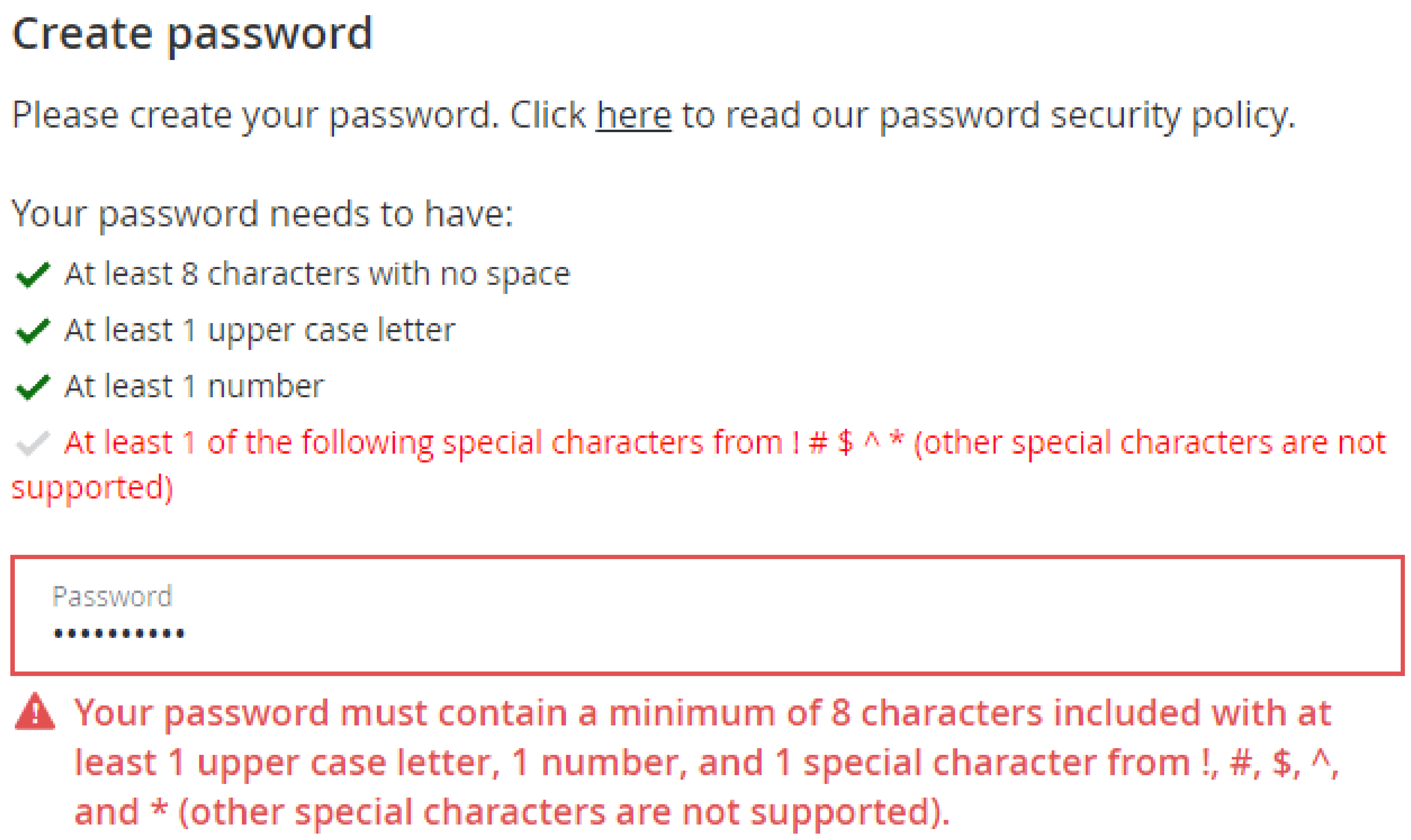
By Iain Thomson in San Francisco 29 Oct 2015 at 21:32 36 SHARE ▼

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

Threat Mitigation: Untrusted Inputs

Restrict inputs to only “valid” or “safe” characters

- Special characters like <, >, ‘, “ and ` are often involved in exploits involving untrusted inputs
- Simple fix: Prohibit such inputs using input validation

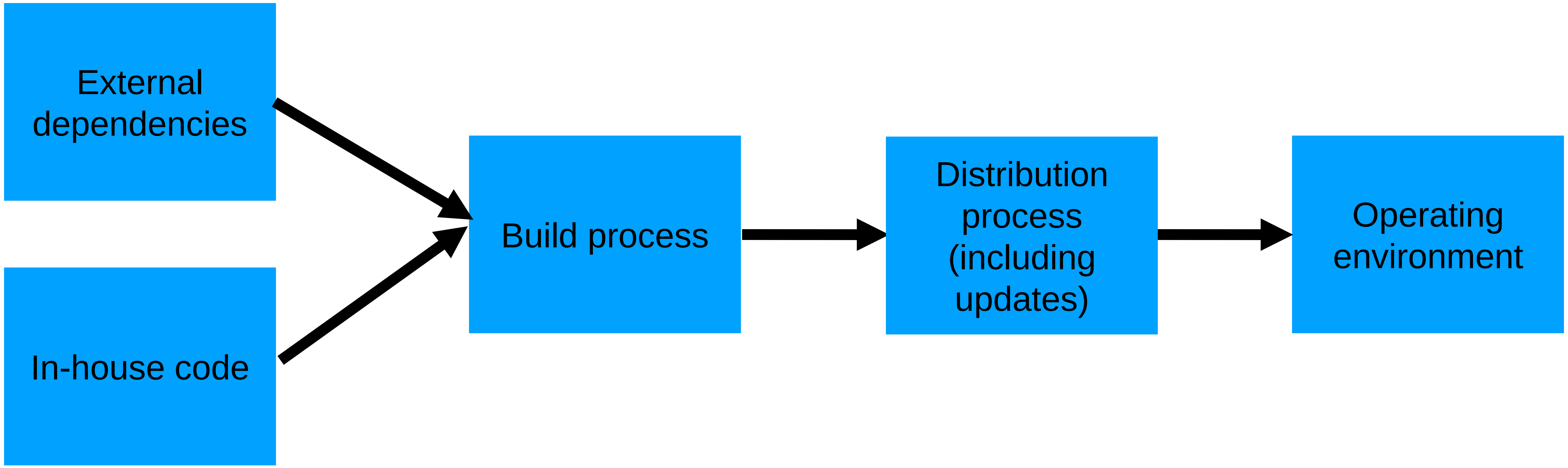


Threat Mitigation: Untrusted Inputs

- Sanitize inputs – prevent them from being executable
- Avoid use of languages or features that can allow for remote code execution, such as:
    - eval() in JS – executes a string as JS code
    - Query languages (e.g. SQL, LDAP, language-specific languages like OGNL in java)
    - Languages that allow code to construct arbitrary pointers or write beyond a valid array index

Threat Mitigation: Software Supply Chain

Consider threats at each phase



Threat Mitigation: Software Supply Chain

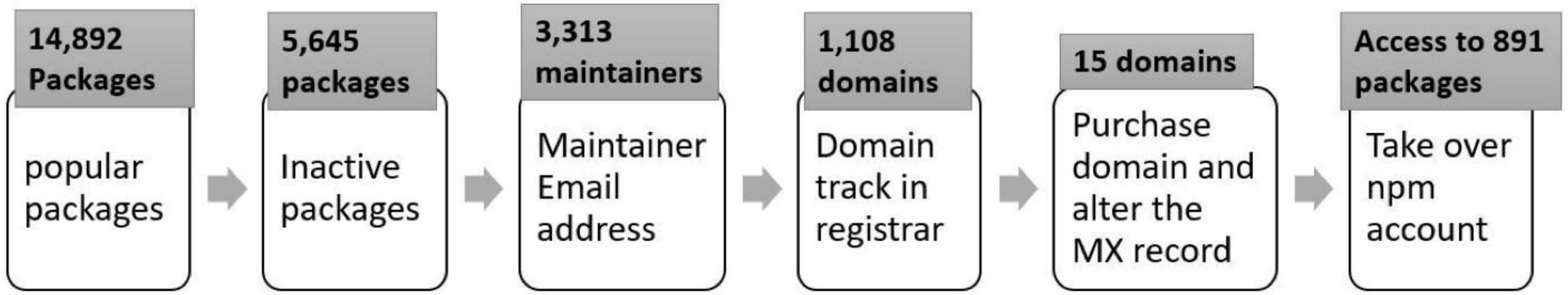
Process-based solutions for process-based problems

- 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

Weak Links in Software Supply Chain

2021 NCSU/Microsoft Study

- 8,498 NPM packages are maintained by at least one maintainer whose email address is inactive and could be purchased
- 33,249 NPM packages include installation scripts that can be exploited to run arbitrary code on developers' machines at installation-time
- 5,645 NPM packages are not actively maintained



"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

Part 4: Which threats to protect against, at what cost?

Consider various costs:

- Performance:
  - Encryption is not free
  - Preventing buffer overruns is not free
    - “Safe” languages like TS are usually (but not always) slower than optimized C.
- Expertise:
  - It is easy to try to implement these measures, it is hard to get them right
- Financial:
  - Implementing these measures takes time and resources

OWASP Top Security Risks

All 10: https://owasp.org/www-project-top-ten/

- Broken authentication + access control
- Cryptographic failures
- Code injection (various forms - SQL/command line/XSS/XML/deserialization)
- Weakly protected sensitive data
- Using components with known vulnerabilities

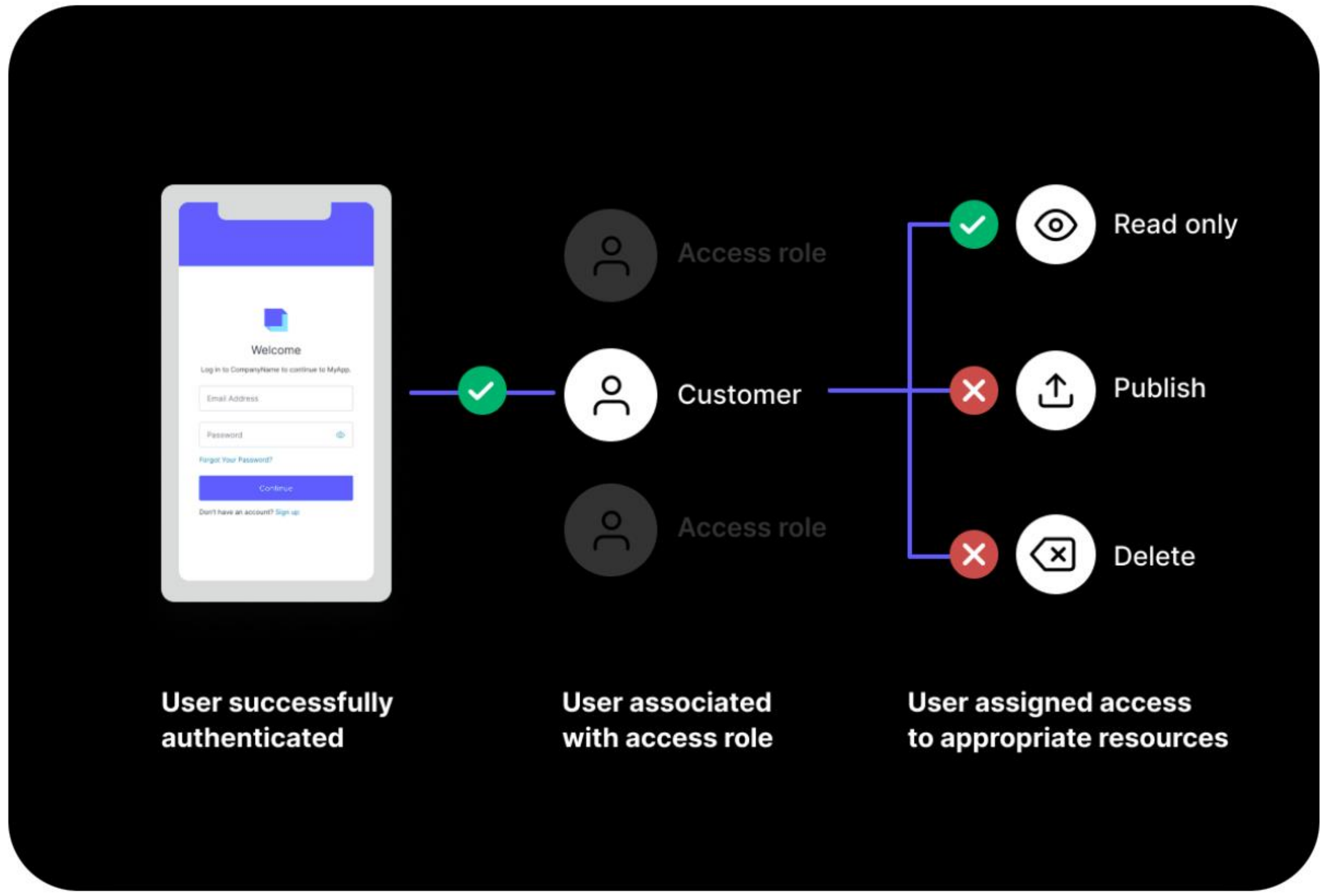
Mitigations for Broken Authentication + Access Control

OWASP #1

- Implement multi-factor authentication
- Implement weak-password checks
- Apply per-record access control
- Harden account creation, password reset pathways

Don't do this at home!
Use a trusted component instead

Auth0



https://auth0.com

Cryptographic Failures

OWASP #2

- Enforce encryption on all communication
- Validate SSL certificates; rotate certificates regularly
- Protect user-data at rest (passwords, credit card numbers, etc)
- Protect application “secrets” (e.g. signing keys)

	Amazon	Facebook	Twitter	Bitly	Flickr	Foursquare	Google	LinkedIn	Titanium
Total candidates	1,241	1,477	28,235	3,132	159	326	414	1,434	1,914
Unique candidates	308	460	6,228	616	89	177	225	181	1,783
Unique % valid	93.5%	71.7%	95.2%	88.8%	100%	97.7%	96.0%	97.2%	99.8%

Table 5: Credentials statistics from June 22, 2013 and validated on November 11, 2013. A credential may consist of an ID token and secret authentication token.

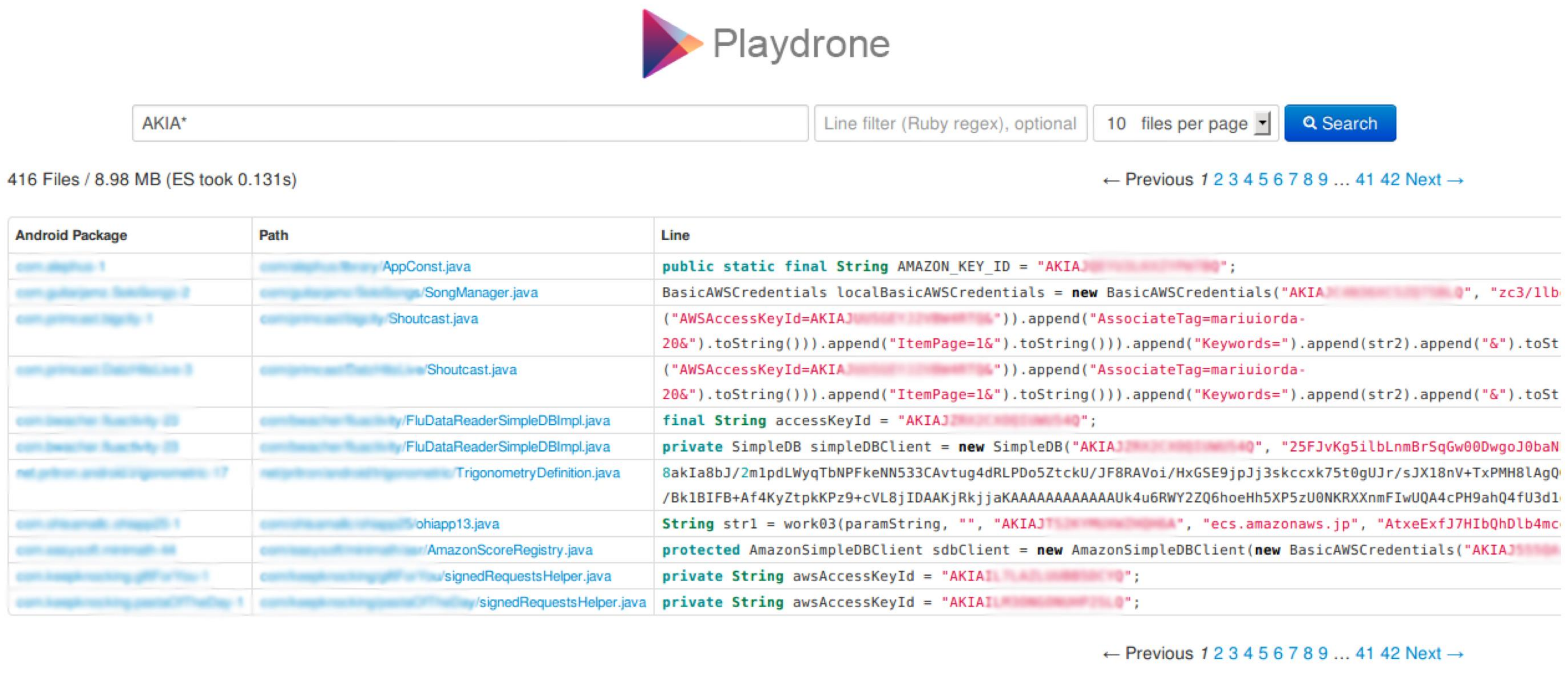
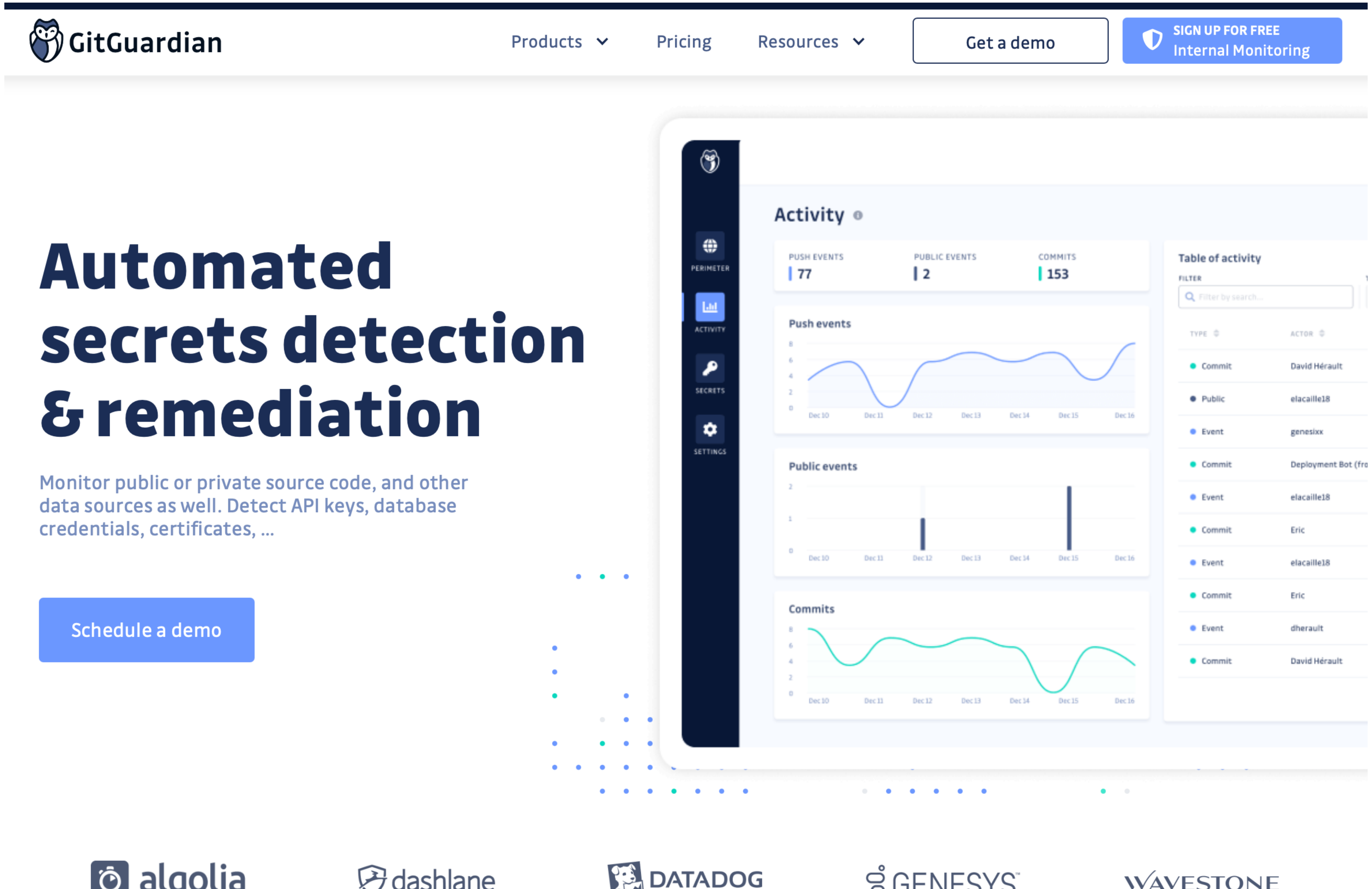


Figure 9: PLAYDRONE’s web interface to search decompiled sources showing Amazon Web Service tokens found in 130 ms.

“A Measurement Study of Google Play,” Viennot et al, SIGMETRICS ‘14

Static Analysis can help detect secrets at rest in a repo

GitGuardian (Launched in 2017)



Cryptographic Failures

Secret detection tools are not enough

These are process-related issues and so require process-related solutions.

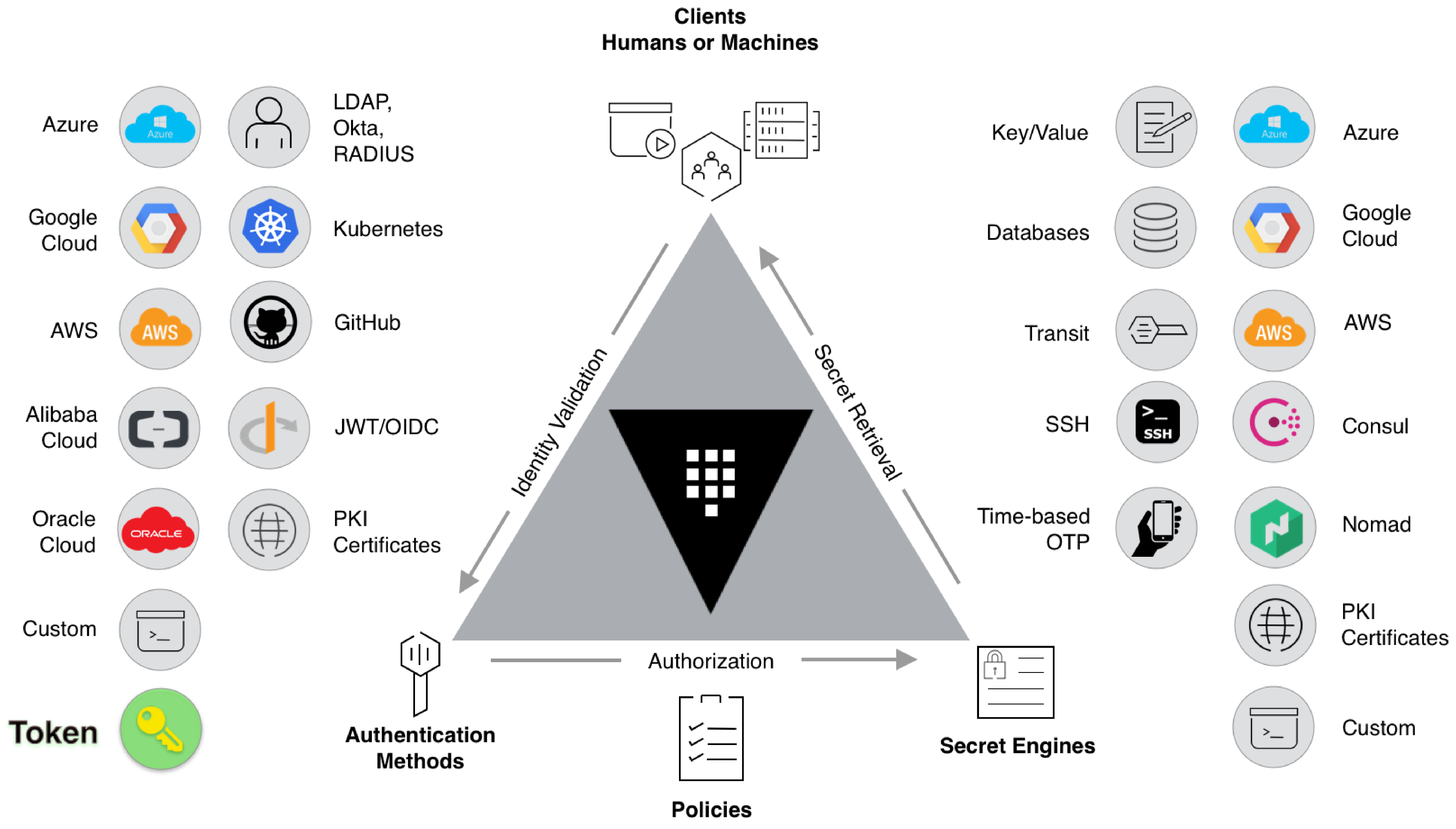
- Industrial study of secret detection tool in a large software services company with over 1,000 developers, operating for over 10 years
- What do developers do when they get warnings of secrets in repository?
  - 49% remove the secrets; 51% bypass the warning
- Why do developers bypass warnings?
  - 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"

"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

Cryptographic Failures

Secret management tools ("Vaults") centralize points of failure, and automates:

- Authorizing access to secrets
- Providing time-limited secrets
- Audit secret access

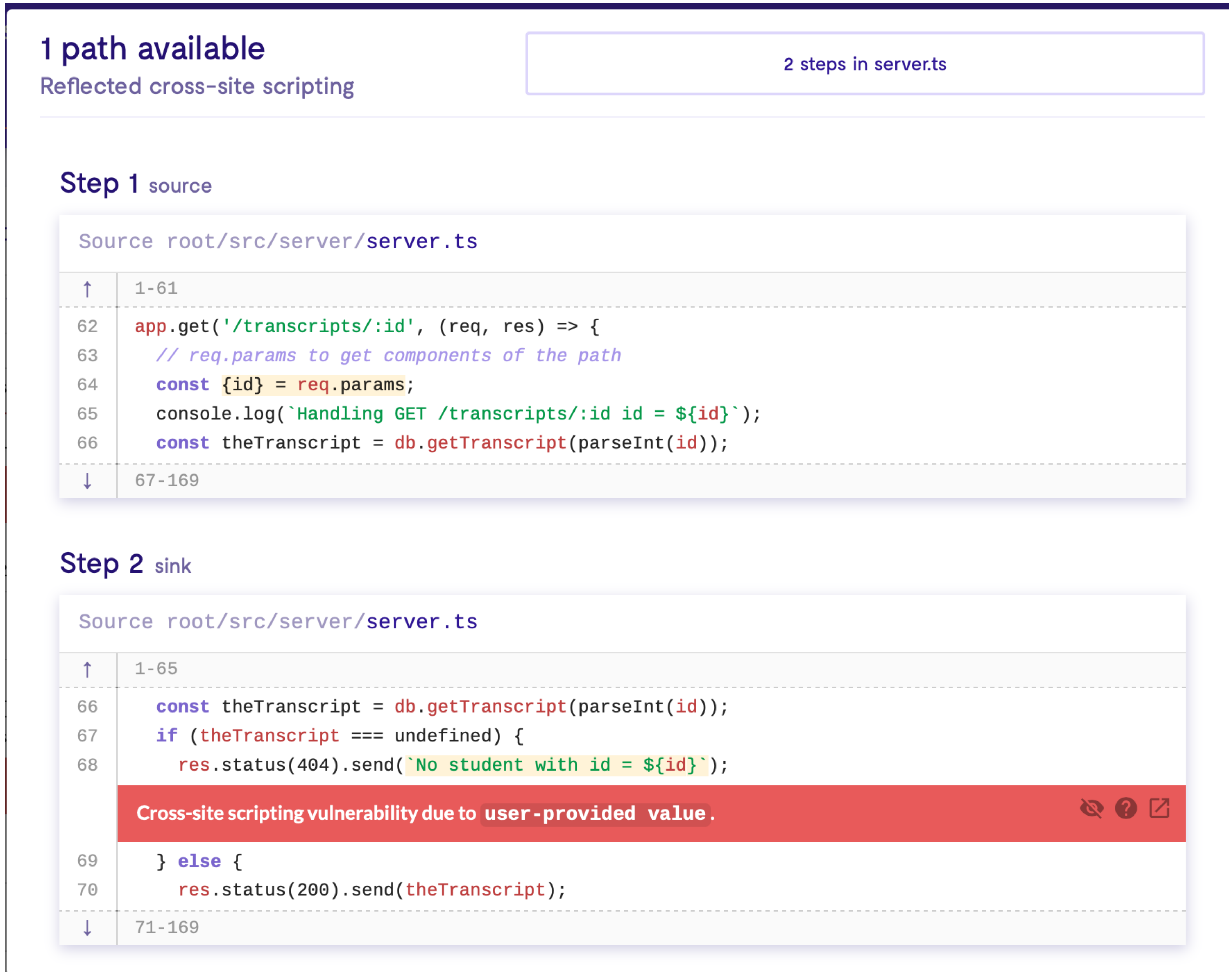


Example platform: HashiCorp Vault (open source, or cloud-hosted)
https://learn.hashicorp.com/tutorials/vault/getting-started-intro?in=vault/getting-started

Code Injection

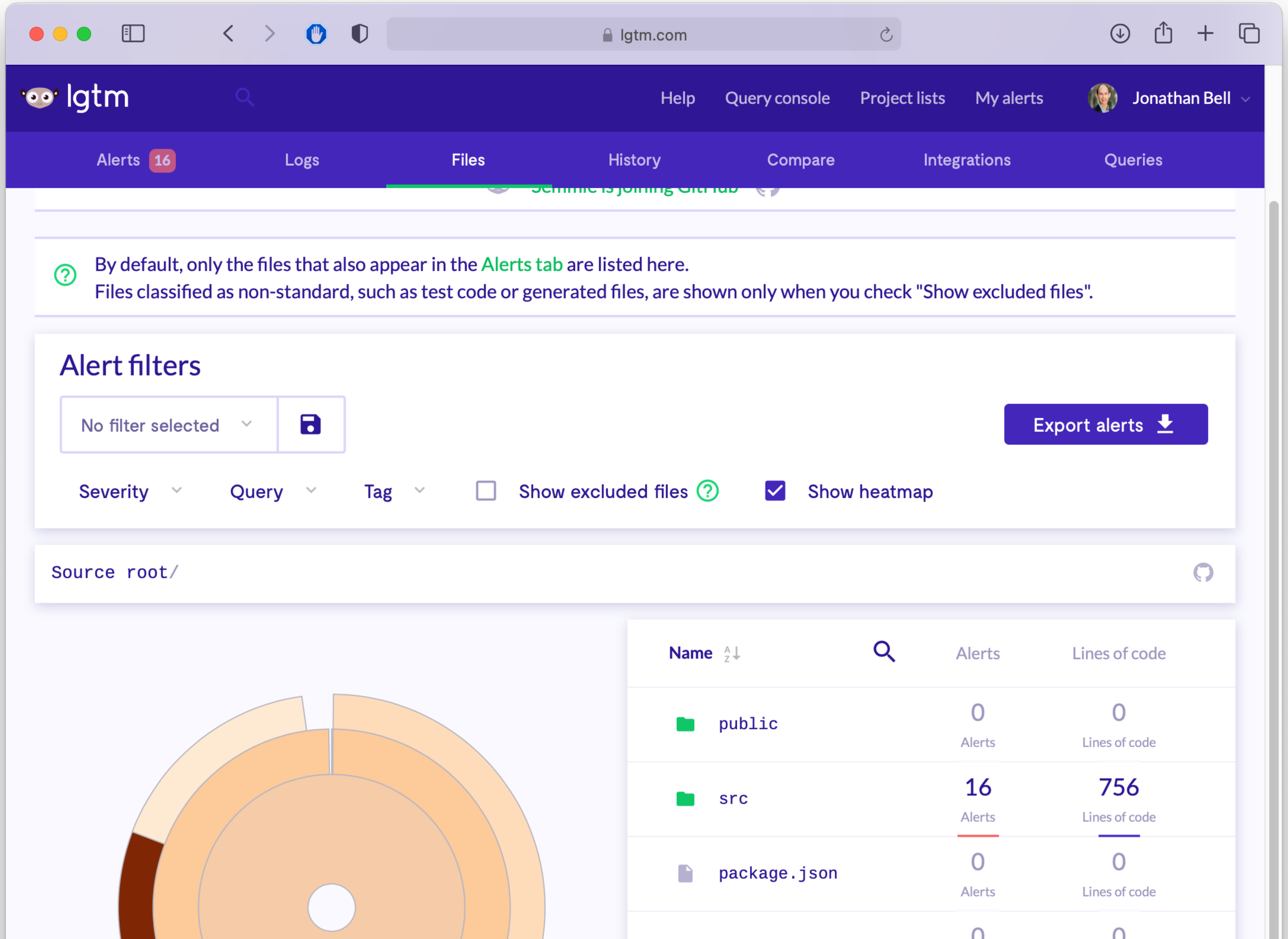
OWASP #3

- **Sanitize** user-controlled inputs (remove HTML)
- Use tools like LGTM to detect vulnerable data flows
- Use middleware that side-steps the problem (e.g. return data as JSON, client puts that data into React component)



Detecting Weaknesses in Apps with Static Analysis

LGTM + CodeQL



Clear text storage of sensitive information
Sensitive information stored without encryption or hashing can expose it to an attacker.

Clear-text logging of sensitive information
Logging sensitive information without encryption or hashing can expose it to an attacker.

Client-side cross-site scripting
Writing user input directly to the DOM allows for a cross-site scripting vulnerability.

Client-side URL redirect
Client-side URL redirection based on unvalidated user input may cause redirection to malicious web sites.

Code injection
Interpreting unsanitized user input as code allows a malicious user arbitrary code execution.

Download of sensitive file through insecure connection
Downloading executables and other sensitive files over an insecure connection opens up for potential man-in-the-middle attacks.

https://lgtm.com

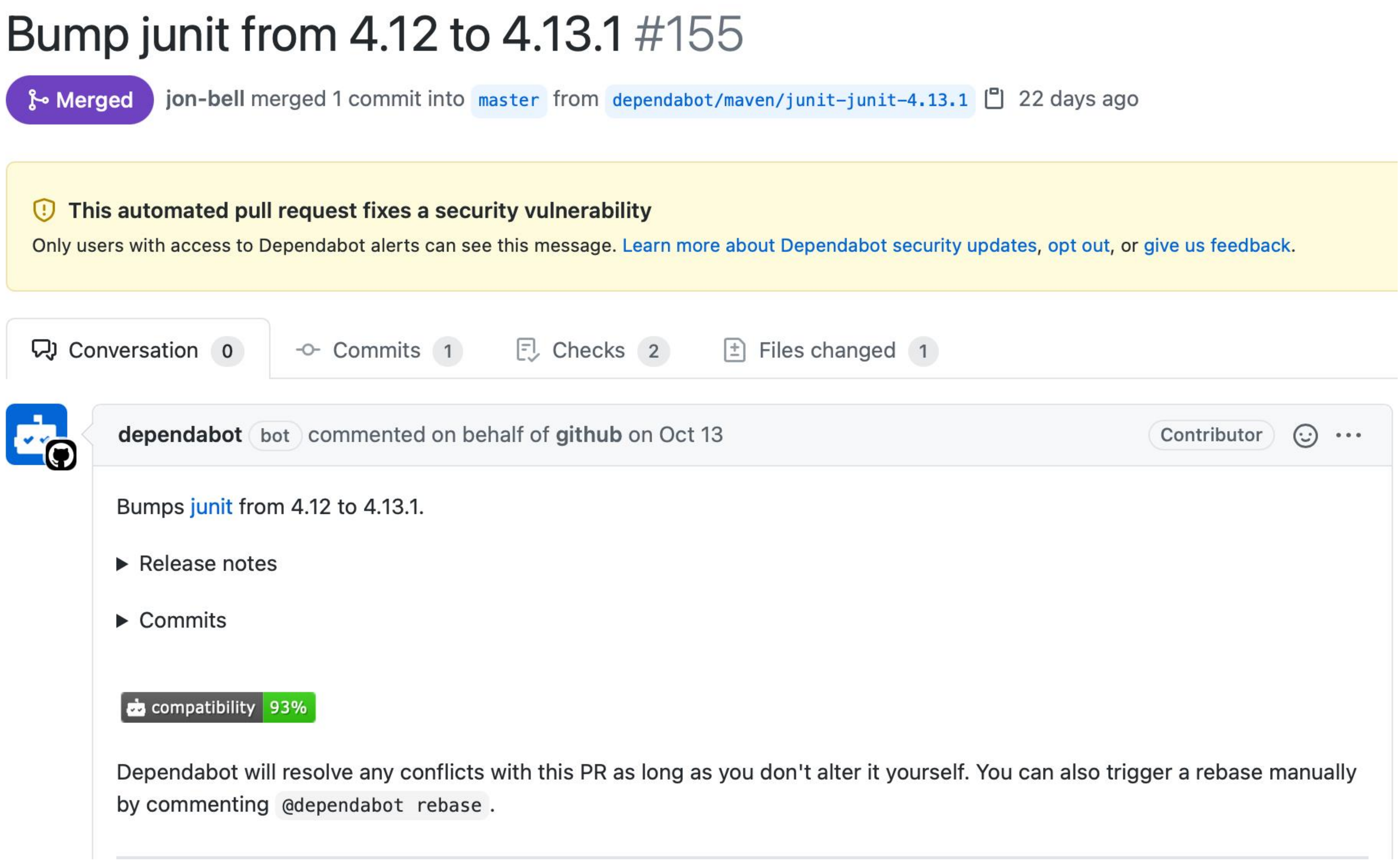
Weakly Protected Sensitive Data

OWASP #4

- Classify your data by sensitivity
- Encrypt sensitive data - in transit and at rest
- Make a plan for data controls, stick to it
- Software engineering fix: can we avoid storing sensitive data?
    - Payment processors: Stripe, Square, etc

Using Components with Known Vulnerabilties

OWASP #5



Dependabot will also send you an email

GitHub security alert digest

mwand's repository security updates from the week of **Mar 22 - Mar 29**

neu-se organization

neu-se / covey-town-roomservice-buggy

Known security vulnerabilities detected

Dependency	Version	Upgrade to
`xmlhttprequest-ssl`	`< 1.6.2`	`~> 1.6.2`

Defined in
`package-lock.json`

Vulnerabilities
CVE-2021-31597 Critical severity
CVE-2020-28502 High severity

Dependency	Version	Upgrade to
`hosted-git-info`	`< 2.8.9`	`~> 2.8.9`

Defined in
`package-lock.json`

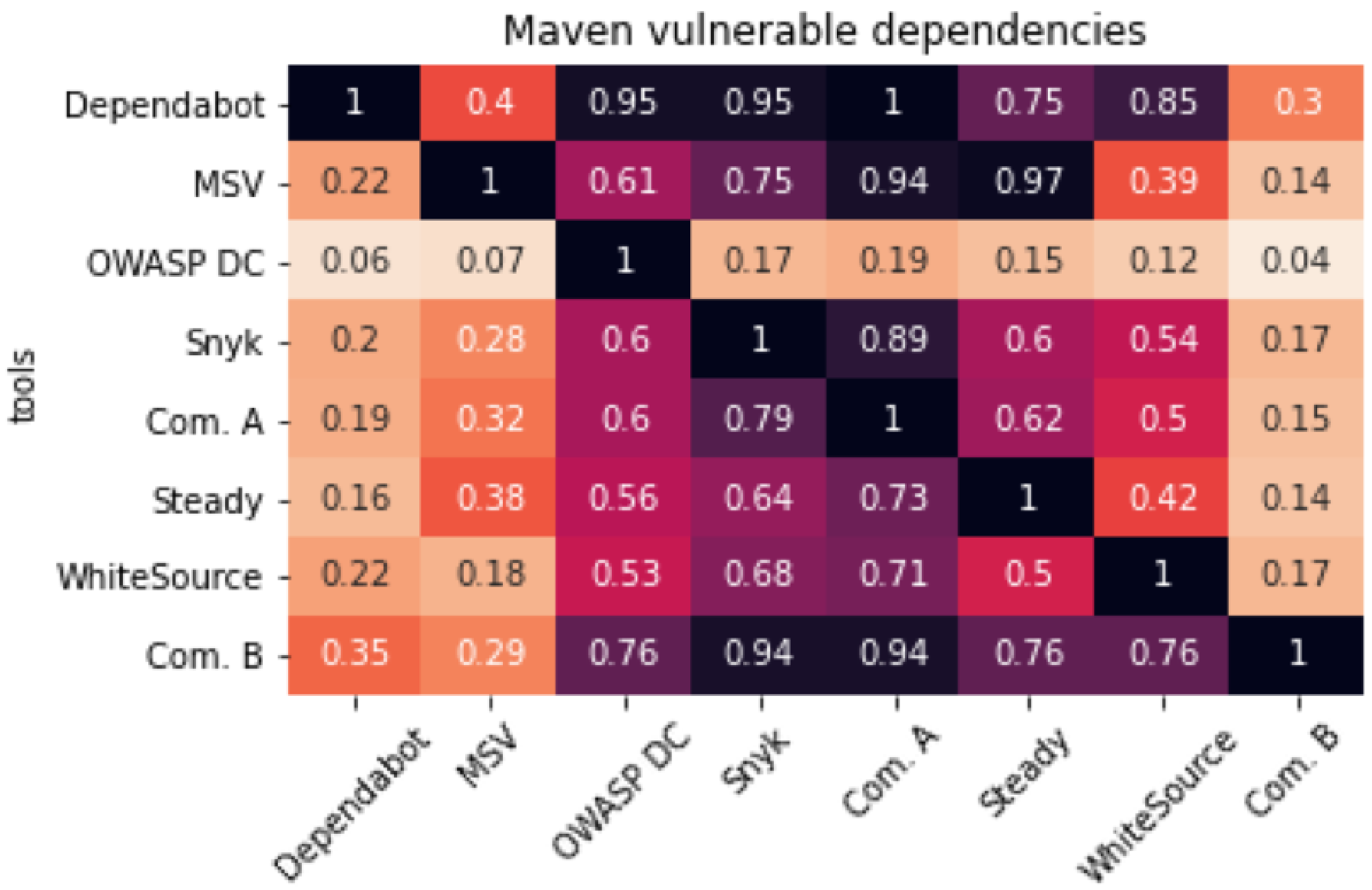
Vulnerabilities
CVE-2021-23362 Moderate severity

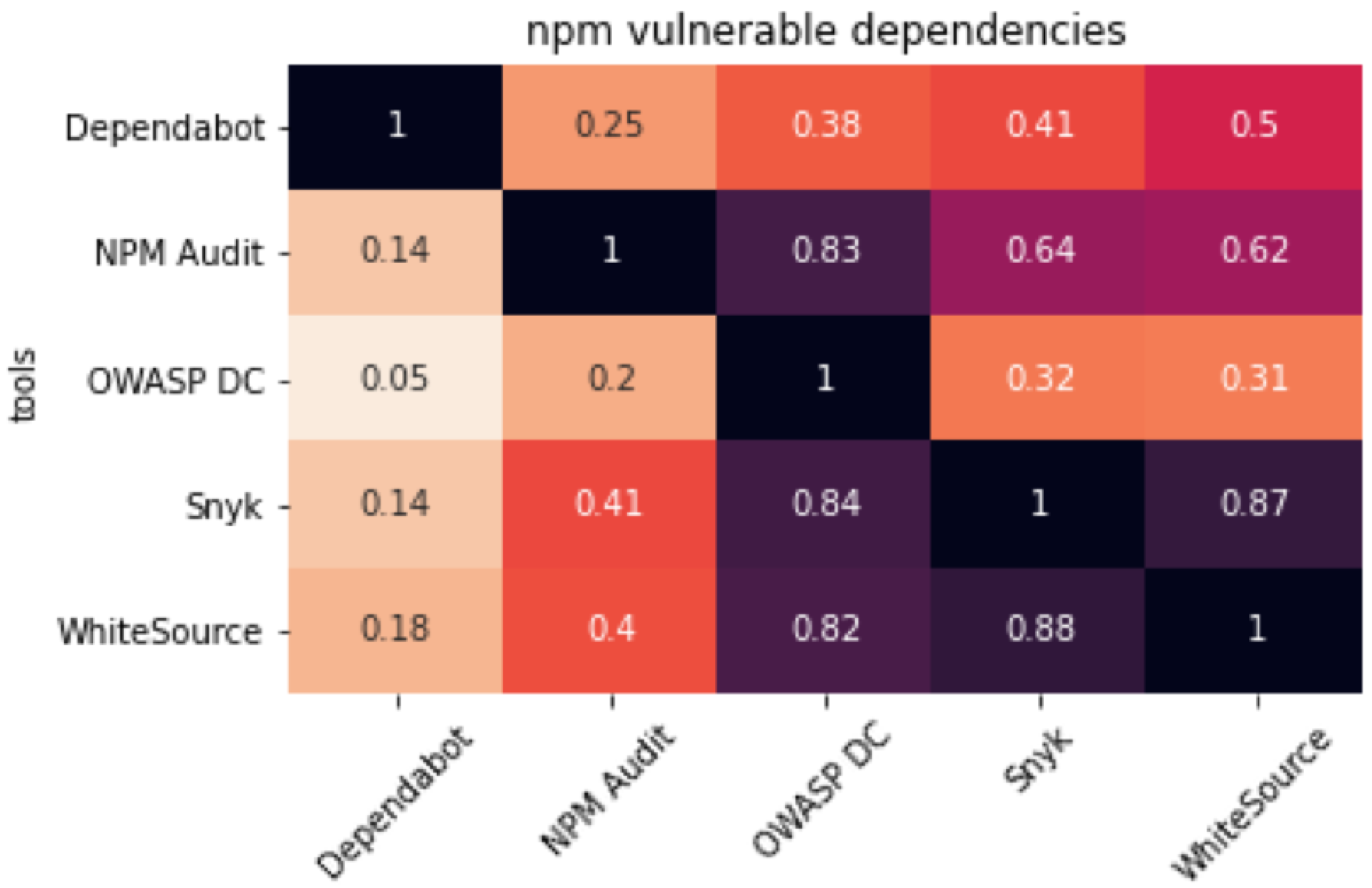
Dependency Version Upgrade to

Using Components with Known Vulnerabilties

Static analyses are imperfect

- Study: Vulnerable dependencies reported on a large, open source project, OpenMRS. Compare results across tools.





"A comparative study of vulnerability reporting by software composition analysis tools"
Nasif Imtiaz, Seaver Thorn and Laurie Williams
https://dl.acm.org/doi/10.1145/3475716.3475769

Learning Objectives for this Lesson

By the end of this lesson, you should be able to…

- Describe that security is a spectrum, and be able to define a realistic threat model for a given system
- Evaluate the tradeoffs between security and performance in software engineering
- Recognize the causes of and common mitigations for common vulnerabilities in web applications
- Utilize static analysis tools to identify common weaknesses in code