Lec 18: CSRF and XSS

CSED415: Computer Security

Spring 2026

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Team project
  - Please submit CVE candidates by Friday, May 1
  - PLMS assignment:

> **[Project] CVE candidates**
>
> Please submit three CVE candidates. For each candidate, provide:
>
> 1. CVE identifier
>
> 2. Justification (reason for selection).
>
> Note: Only the team leader should submit on behalf of the team.

Recap

POSTECH

- Web authentication
  - Cookies
  - Sessions
  - JWT and SSO

Cross-Site Request Forgery (CSRF)

Review: Cookies and session tokens

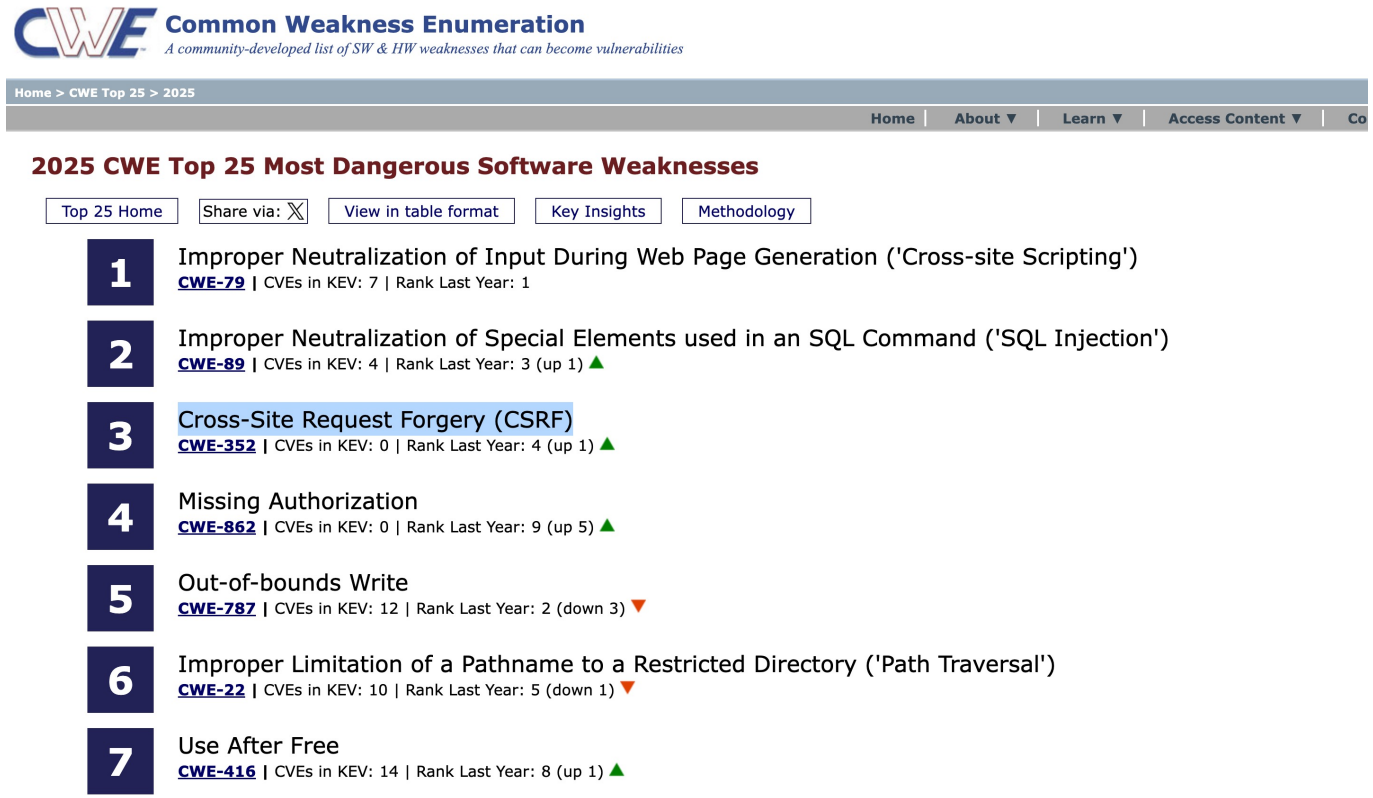
- Session token cookies are used to associate a request with a user
- The browser automatically attaches relevant cookies in every request
  - [From Lecture 17] The browser sends the cookie if both are true:
    - The `Domain` attribute is a domain suffix of the server's domain
    - The `Path` attribute is a prefix of the server's path

Cross-Site Request Forgery (CSRF)

- What if an attacker tricks the victim into making an unintended request?
  - The victim's browser will automatically attach relevant cookies
  - The server will think the request came from the victim!
- CSRF:
  - An attack that exploits cookie-based authentication to perform an action as the victim

Cross-Site Request Forgery (CSRF)

- CSRF is ranked #3 in the “2025 CWE Top 25 Most Dangerous Software Weaknesses”



Common Weakness Enumeration

A community-developed list of SW & HW weaknesses that can become vulnerabilities

Home > CWE Top 25 > 2025

Home | About ▼ | Learn ▼ | Access Content ▼ | Co

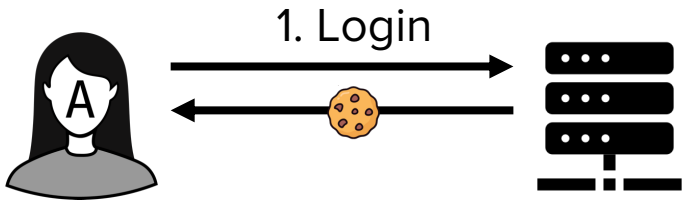
2025 CWE Top 25 Most Dangerous Software Weaknesses

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

1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 7 | Rank Last Year: 1
2. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3 (up 1) ▲
3. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 4 (up 1) ▲
4. Missing Authorization
   CWE-862 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲
5. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 12 | Rank Last Year: 2 (down 3) ▼
6. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 10 | Rank Last Year: 5 (down 1) ▼
7. Use After Free
   CWE-416 | CVEs in KEV: 14 | Rank Last Year: 8 (up 1) ▲

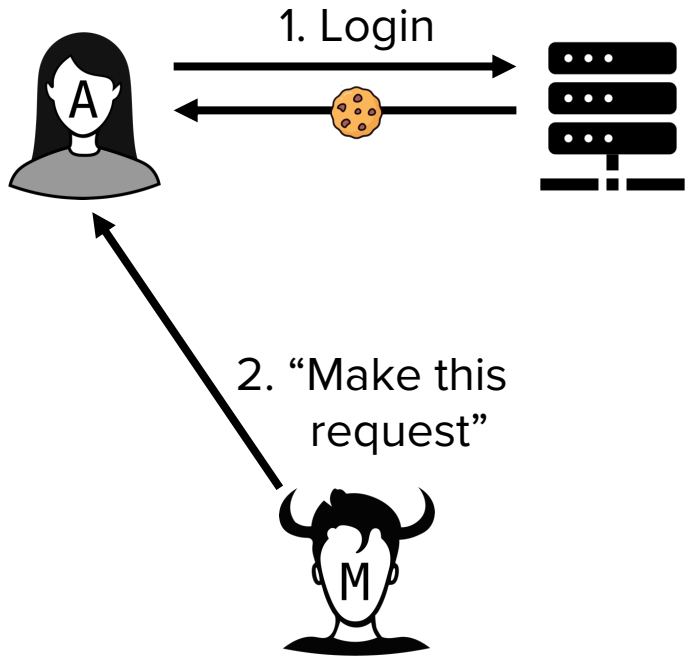
CSRF attack workflow

1. User authenticates to the server
   - Receives a cookie with a valid session token



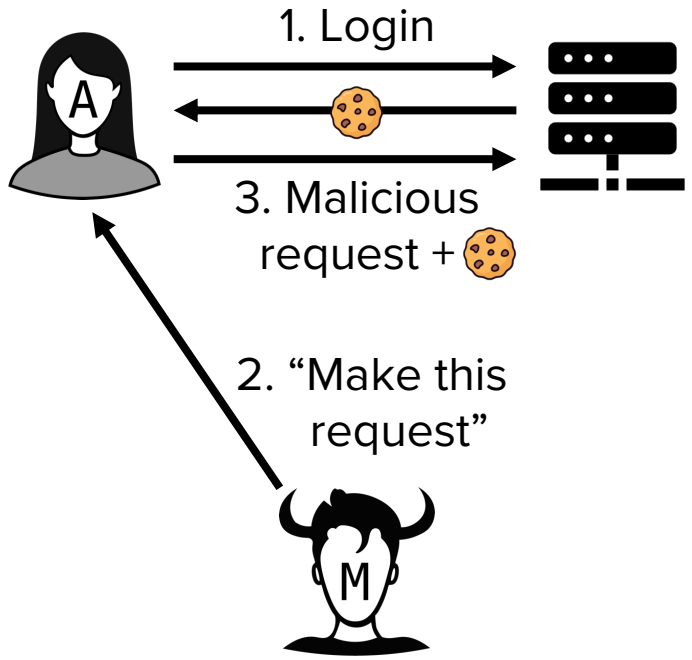
CSRF attack workflow

1. User authenticates to the server
   - Receives a cookie with a valid session token
2. Attacker tricks the user into making a malicious request to the server



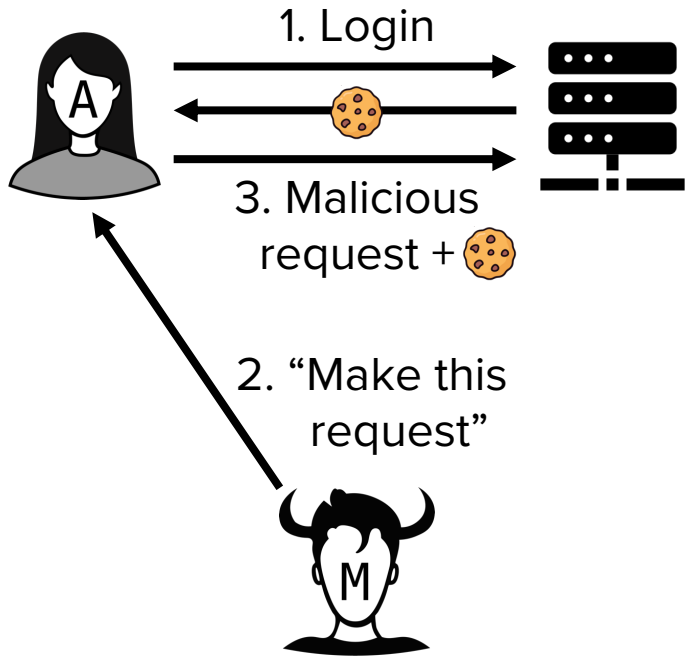
CSRF attack workflow

1. User authenticates to the server
   - Receives a cookie with a valid session token
2. Attacker tricks the user into making a malicious request to the server
3. The server accepts the malicious request from the user
   - Because the legitimate cookie is automatically attached to the request



CSRF attack workflow

1. User authenticates to the server
   - Receives a cookie with a valid session token
2. Attacker tricks the user into making a malicious request to the server
3. The server accepts the malicious request from the user
   - Because the legitimate cookie is automatically attached to the request



Challenge: How to trick a user?

Background: GET and POST requests

- GET request
  - An HTTP method used to retrieve data from a server
  - Parameters are sent via URL query string

```
https://example.com/search?user=alice&title=security
```

domain path query #1 query #2

- POST request
  - An HTTP method used to send data to a server
  - Parameters are sent in the request body (not in the URL)
    - HTTP forms are typically used

```
<form action="/login" method="POST">
  <input name="username">
  <input name="password" type="password">
  <button type="submit">Login</button>
</form>
```

Tricking a user into making a GET request

POSTECH

- GET Strategy #1: Trick the user into clicking a link
  - Send an email with a fake link that directly triggers a GET request:

```
<a href="https://bank.com/transfer?amount=100&to=Mallory">Click for free gift!</a>
```

  - Or send a link that opens an attacker-controlled website, which silently issues a GET request to target site using JavaScript:

```
<html><body>
<script>
  // Automatically send a GET request when the page loads
  fetch("https://bank.com/transfer?amount=100&to=Mallory", {
    method: "GET",
    credentials: "include" // include victim's cookies
  });
  </script>
</body></html>
```

Tricking a user into making a GET request

POSTECH

- GET Strategy #2: Inject HTML into a website the user visits
  - e.g.,
    - The user visits a bulletin board
    - The attacker posts content that contains:

```
<img src="https://bank.com/transfer?amount=100&to=Mallory">
```

    - Even if the image fails to load, the browser still sends the GET request, along with the user's cookies

Tricking a user into making a POST request

- POST Strategy #1: Trick the user into clicking a link
  - Send a link that opens an attacker-controlled website, which silently issues a POST request to target site using JavaScript:

```
<html>
<body>
  <form id="csrfForm" action="https://bank.com/transfer" method="POST">
    <input type="hidden" name="amount" value="100">
    <input type="hidden" name="to" value="Mallory">
  </form>
  <script>
    // Automatically submit the form when the page loads
    document.getElementById("csrfForm").submit();
  </script>
</body>
</html>
```

Tricking a user into making a POST request

POSTECH

- POST Strategy #2: Inject JavaScript into a website victim visits
  - JavaScript can issue POST requests:

```
<script>
const form = document.createElement("form");
form.method = "POST";
form.action = "https://bank.com/transfer";

const amount = document.createElement("input");
amount.type = "hidden";
amount.name = "amount";
amount.value = "100";
const to = document.createElement("input");
to.type = "hidden";
to.name = "to";
to.value = "Mallory";

form.appendChild(amount);
form.appendChild(to);
document.body.appendChild(form);
form.submit();
</script>
```

CSRF vulnerabilities in the wild

- YouTube (2008)
  - Attack mechanism
    - Victim is logged in to YouTube
    - The victim visits an attacker-controlled page
    - The page silently issues authenticated requests to YouTube with cookies
  - Attacker capabilities
    - Add videos to the victim's Favorites
    - Subscribe the victim to channels
    - Post comments on the victim's behalf

```
<img src="https://www.youtube.com/add_favorite?video_id=ATTACKER_VIDEO">
```

[Malicous `img` tag on the attacker-controlled page]

CSRF vulnerabilities in the wild

POSTECH

- Facebook (2019)
  - Attack mechanism
    - Victim is logged in to Facebook
    - Victim is tricked into visiting a malicious URL that appears as `facebook.com`
    - The URL abuses an internal endpoint to issue authenticated requests
  - Attacker capabilities
    - Post content on the victim's timeline
    - Remove the victim's profile picture
    - Delete the victim's account

```
https://www.facebook.com/comet/dialog_DONOTUSE/?
url=/profile/picture/remove_picture/%3fdelete_from_album=1%26profile_id={ID}
```

[Malicious URL]

CSRF Defenses

CSRF defenses

- CSRF defenses must be implemented by the server
  - Unlike the cookie policy and same origin policy, which are enforced by the browser

CSRF defense #1: CSRF tokens

- Idea: Add a secret value in the request that the attacker doesn't know
  - The server only accepts requests if it has a valid secret
  - The attacker can no longer create a malicious request without knowing the secret
- CSRF token: A secret value provided by the server to the user
  - CSRF tokens cannot be sent to the server in a cookie
  - They should be explicitly sent in a GET parameter or POST content

CSRF defense #1: CSRF tokens

- CSRF token example
  - Server embeds a hidden CSRF token in a form of its website
    - The token is unique per session/user and unpredictable

```
<!-- https://bank.com/transfer -->
<form action="/transfer" method="POST">
  <input type="hidden" name="csrf_token" value="A9f3XkT..."> <!-- server-generated -->
  <input name="amount" value="100">
  <input name="to" value="Bob">
  <button type="submit">Send</button>
</form>
```

  - User fills in the form and submits it, issuing a POST request
    - The request includes the hidden CSRF token
  - The server verifies the token before processing the request

CSRF defense #1: CSRF tokens

- CSRF token example
  - Server embeds a hidden CSRF token in a form of its website
    - The token is unique per session/user and unpredictable

```
<!-- https://bank.com/transfer -->
<form action="/transfer" method="POST">
  <input type="hidden" name="csrf_token" value="A9f3XkT..."> <!-- server-generated -->
  <input name="amount" value="100">
  <input name="to" value="Bob">
  <button type="submit">Send</button>
</form>
```

  - Effectiveness
    - Attacker cannot read or guess the victim's CSRF token
    - Therefore, a forged request will have no token or an invalid token!

CSRF defense #2: Referer Header

POSTECH

- Observation: In a CSRF attack, the victim usually makes the malicious request from a different website
- Referer header: A header in an HTTP request that indicates the origin of the request
  - Note: “Referer” is a typo in the HTTP standard ☺
  - Examples:
    - If you type your username and password into Instagram, the Referer header for that request is `https://instagram.com`
    - If JavaScript on an attacker-controlled website forces your browser to make a request, the Referer header for that request is the attacker website’s URL

CSRF defense #2: Referer Header

- Referer header policy:
  - Allow same-site requests

```
POST /transfer HTTP/1.1
Host: bank.com
Referer: https://bank.com/transfer
```

(POST request from `bank.com` to `bank.com`)

  - Reject cross-site requests

```
POST /transfer HTTP/1.1
Host: bank.com
Referer: https://evil.com
```

(POST request from `evil.com` to `bank.com`)

- Effectiveness
  - Blocks requests coming from attacker-controlled sites

CSRF defense #3: SameSite cookies

- Idea: Embed a policy flag in the cookie that disallows cross-site requests
- SameSite
  - A recently added cookie attribute that controls when cookies are sent in cross-site requests
  - Helps prevent CSRF attacks by limiting cookie inclusion

Name	session
Value	abc123
Domain	compsec.postech.ac.kr
Path	/research
SameSite	Strict
Secure	True
HttpOnly	False
Expires	27 Apr 2026, 15:30:00

CSRF defense #3: SameSite cookies

POSTECH

- SameSite=Strict
  - Cookies are sent only for same-site requests
    - Cross-site requests never include cookies
  - Example
    - Victim is logged in to **`bank.com`**
    - The victim is tricked into visiting **`evil.com`**, which tries:

```
<img src="https://bank.com/transfer?amount=100&to=Mallory">
```

    - Or the victim clicks a malicious link:

```
<a href="https://bank.com/transfer?amount=100&to=Mallory">Click for free gift!</a>
```

→ In both cases, browser does not send cookies if `SameSite=Strict`

Strongest protection against CSRF, with reduced usability (e.g., breaks SSO)

CSRF defense #3: SameSite cookies

POSTECH

- SameSite=Lax (default in modern browsers)
  - Cookies are sent for:
    - Same-site requests (GET, POST, etc.)
    - Top-level cross-site navigations (GET only) that replaces the entire page
  - Example
    - Victim is logged in to **bank.com** and visits attacker-controlled link or site:

```
<img src="https://bank.com/transfer?amount=100&to=Mallory">
```
→ Cookie not sent (no nav.)

```
<form action="/transfer" method="POST">
```
→ Cookie not sent (cross-site POST)

```
<a href="https://bank.com/transfer?amount=100&to=Mallory">Click for free gift!</a>
```
→ Cookie is sent (top-level GET nav.)

Blocks most CSRF attacks, but state-changing GET requests remain dangerous

Cross-Site Scripting (XSS)

Review: Same-origin policy

- Same-origin policy
  - Two webpages with different origins should not be able to access each other's resources
  - e.g.,
    - JavaScript on `http://evil.com` cannot access the information on `https://bank.com`

Background: JavaScript

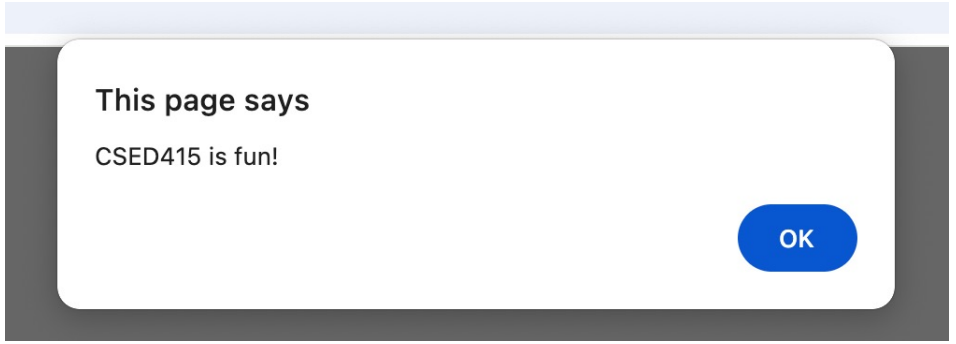
- JavaScript (JS)
  - A programming language for running code in the web browser
  - Manipulates HTML and CSS
    - Makes modern websites interactive
    - JS can be directly embedded in HTML using `<script>` tags
  - [Important] JS is client-side
    - JS code is sent by the server as part of the response
    - The code runs in the browser, not in the server!

Background: JavaScript

- JavaScript (JS)
  - Example: Pop-up message

```
<html>
<body>
<script>alert("CSED415 is fun!");</script>
</body>
</html>
```

HTML + JS



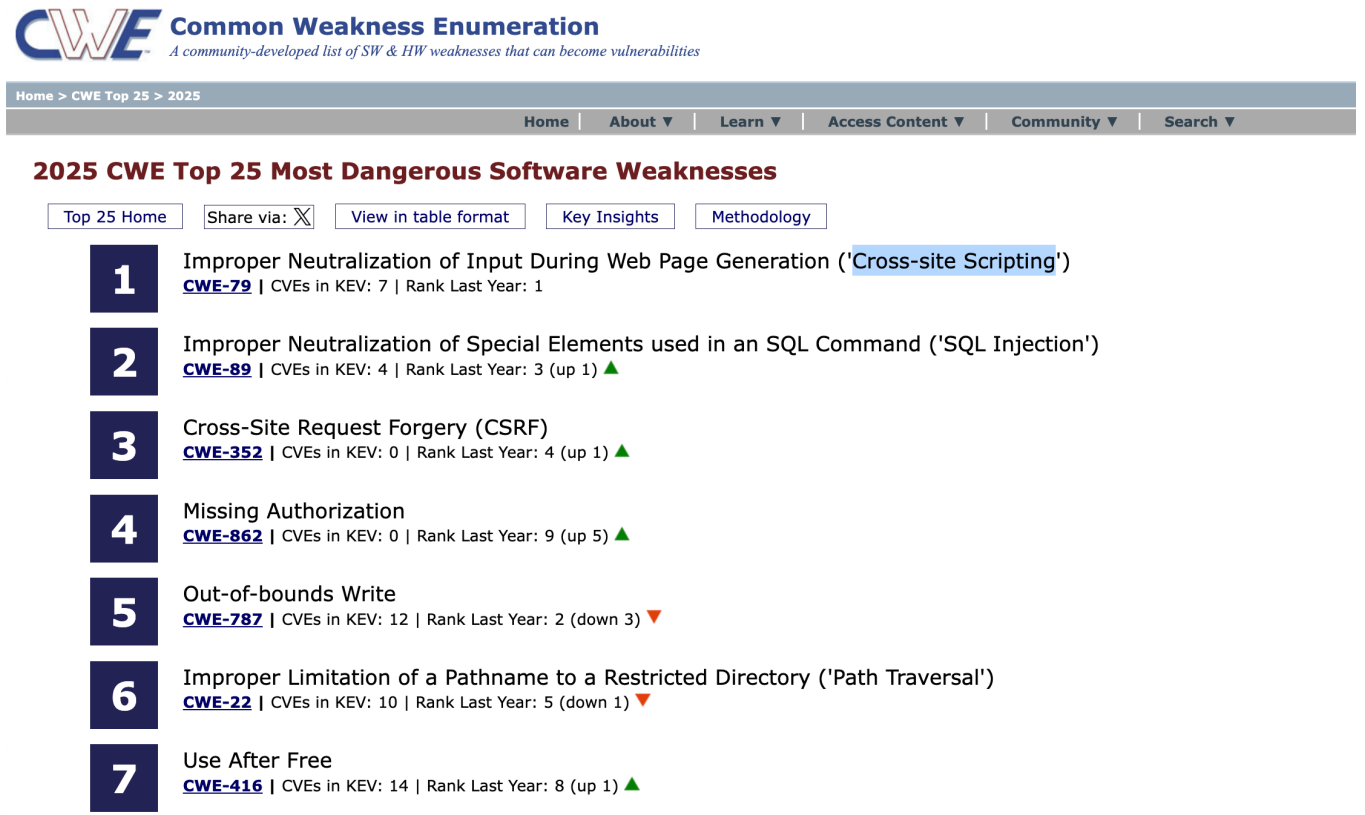
When the browser loads the HTML, it runs the embedded JS and renders a pop-up message

Cross-Site Scripting (XSS)

- XSS: Injecting JS into a legitimate website
  - The website sends the injected script to users
  - The victim's browser executes the attacker's script
  - The script runs with the origin of the legitimate website
    - The script can read/modify page content, access cookies, and perform actions on behalf of the user
- XSS effectively bypasses the same-origin policy by executing the attacker code within the trusted origin

Cross-Site Scripting (XSS)

- XSS is ranked #1 in the “2025 CWE Top 25 Most Dangerous Software Weaknesses”



Common Weakness Enumeration

A community-developed list of SW & HW weaknesses that can become vulnerabilities

Home > CWE Top 25 > 2025

Home | About ▼ | Learn ▼ | Access Content ▼ | Community ▼ | Search ▼

2025 CWE Top 25 Most Dangerous Software Weaknesses

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

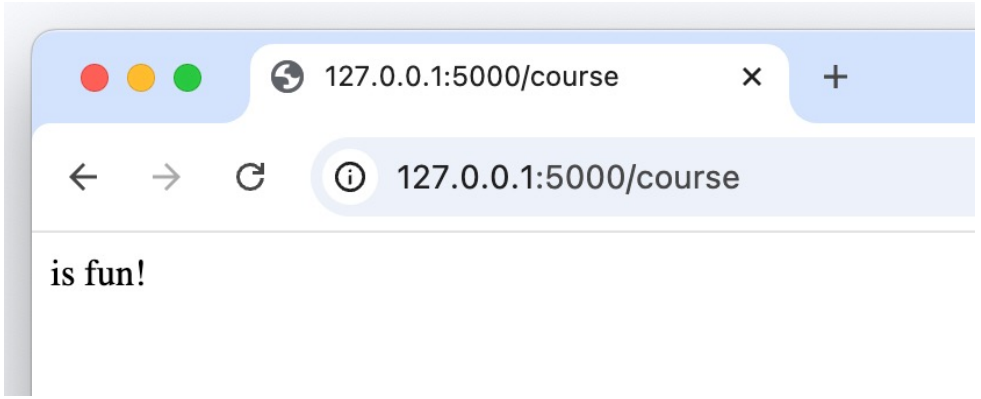
1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 7 | Rank Last Year: 1
2. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3 (up 1) ▲
3. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 4 (up 1) ▲
4. Missing Authorization
   CWE-862 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲
5. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 12 | Rank Last Year: 2 (down 3) ▼
6. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 10 | Rank Last Year: 5 (down 1) ▼
7. Use After Free
   CWE-416 | CVEs in KEV: 14 | Rank Last Year: 8 (up 1) ▲

Example: A vulnerable HTTP server

- A Flask server:
  - Retrieves `course_id` from GET request's parameter
- URL:
  - `http://.../course`
- Response:

```python
from flask import Flask, request
app = Flask(__name__)

@app.route("/course")
def handle_course():
  cid = request.args.get("course_id", "")
  return f"<html><body>{cid} is fun!</body></html>"

if __name__ == "__main__":
  app.run(debug=True)
```

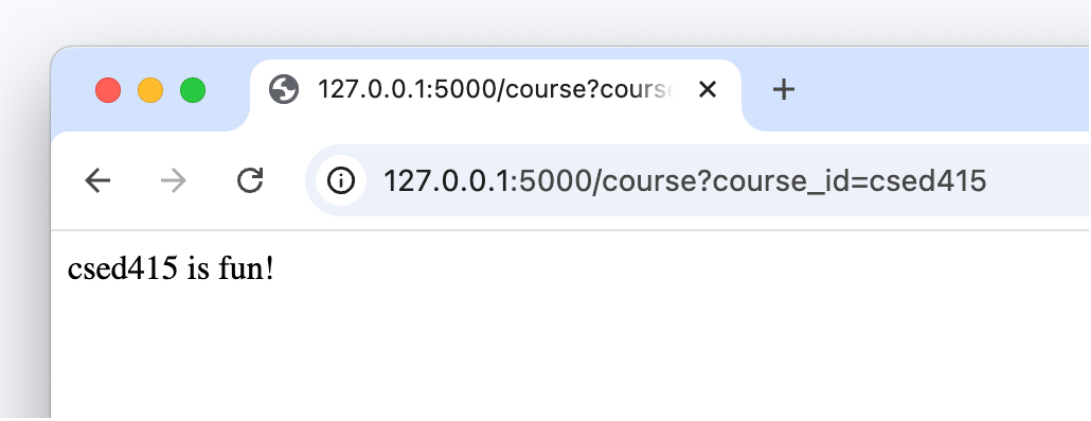
(empty parameter)



Example: A vulnerable HTTP server

- A Flask server:
  - Retrieves `course_id` from GET request's parameter
- URL:
  - `http://.../course?course_id=csed415`
- Response:

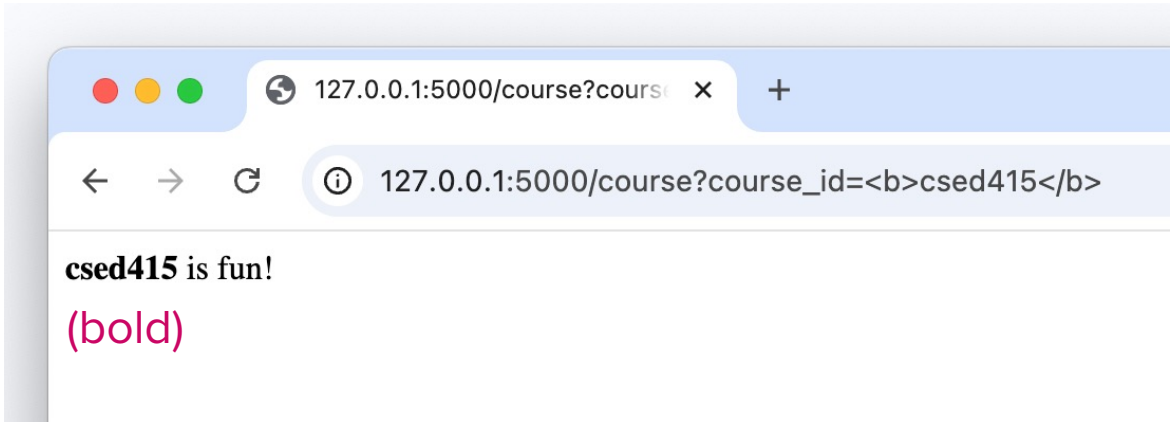
```python
from flask import Flask, request
app = Flask(__name__)

@app.route("/course")
def handle_course():
  cid = request.args.get("course_id", "")
  return f"<html><body>{cid} is fun!</body></html>"

if __name__ == "__main__":
  app.run(debug=True)
```



Example: A vulnerable HTTP server

- A Flask server:
  - Retrieves `course_id` from GET request's parameter
- URL:
  - `http://.../course?course_id=<b>csed415</b>`
- Response:

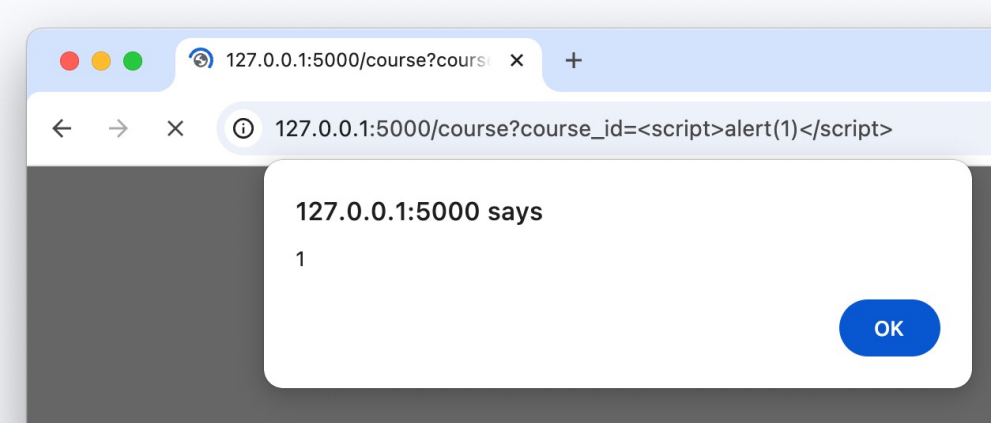
```python
from flask import Flask, request
app = Flask(__name__)

@app.route("/course")
def handle_course():
  cid = request.args.get("course_id", "")
  return f"<html><body>{cid} is fun!</body></html>"

if __name__ == "__main__":
  app.run(debug=True)
```



Example: A vulnerable HTTP server

- A Flask server:
  - Retrieves `course_id` from GET request's parameter
- URL:
  - `http://.../course?course_id=<script>alert(1)</script>`
- Response:

```python
from flask import Flask, request
app = Flask(__name__)

@app.route("/course")
def handle_course():
  cid = request.args.get("course_id", "")
  return f"<html><body>{cid} is fun!</body></html>"

if __name__ == "__main__":
  app.run(debug=True)
```

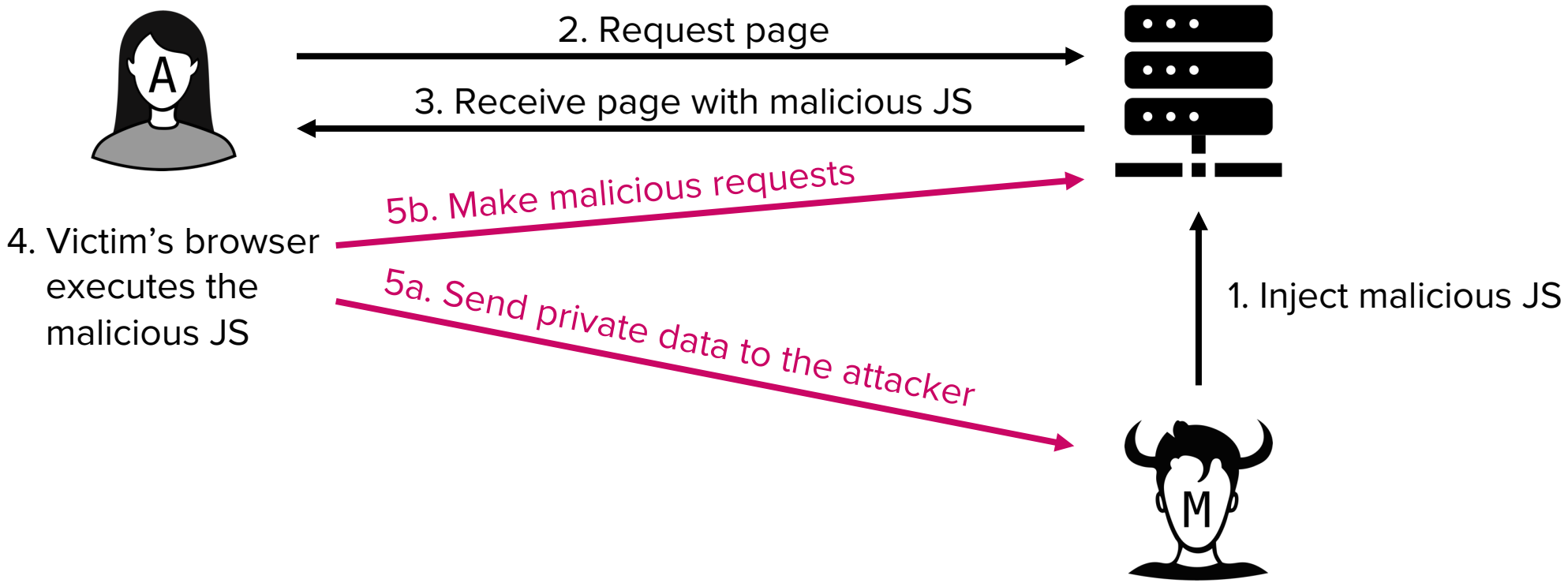


XSS type 1: Stored XSS

- Stored XSS:
  - The attacker stores malicious JS on the legitimate server so they are sent to browsers
  - Example: Facebook pages
    - Attacker embeds malicious JS on their Facebook pages
    - Anyone loading the attacker's page will load the malicious JS
      - With the origin of Facebook

XSS type 1: Stored XSS

- Stored XSS workflow

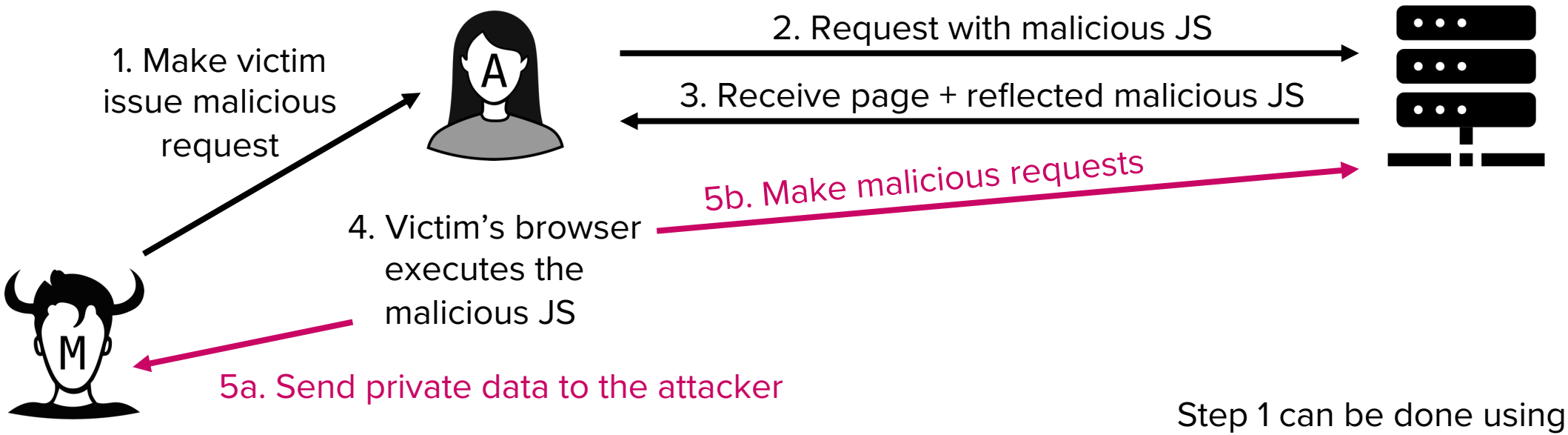


XSS type 2: Reflected XSS

- Stored XSS:
  - The attacker causes the victim to input JS into a request, and the content is reflected (copied) into the response from the server
  - Example: Search results
    - Request: `https://google.com/search?q=csed415`,
      Google's response: `"10,000 results for csed415 ..."`
    - Request: `https://google.com/search?q=<script>alert(1)</script>`,
      Google's response: `"10,000 results for <script>alert(1)</script>"`

XSS type 2: Reflected XSS

- Reflected XSS workflow:



Step 1 can be done using
- Phishing email with URL
- Forum post with URL
- Message with URL
- ...

XSS vs CSRF

- Reflected XSS and GET-based (link-triggered) CSRF both require the victim to click a malicious link
- Differences
  - XSS: Injects and executes malicious JS code in the victim's browser
  - CSRF: Relies on the browser's behavior that automatically send authentication cookies

XSS Defenses

XSS defense #1: HTML sanitization

- HTML sanitization
  - Server should sanitize HTML documents to escape special characters, such as < and >, or encode them to prevent injected JS from being interpreted as an executable script
  - e.g., replace `<` with `<`, replace `"` with `&quot;`, …

XSS defense #1: HTML sanitization

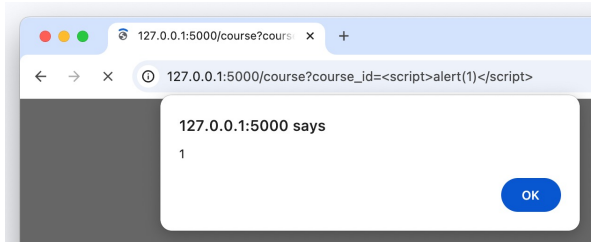
- HTML sanitization
  - Vulnerable server (previous example)

```python
from flask import Flask, request
app = Flask(__name__)

@app.route("/course")
def handle_course():
  cid = request.args.get("course_id", "")
  return f"<html><body>{cid} is fun!</body></html>"

if __name__ == "__main__":
  app.run(debug=True)
```

URL: .../course?course_id=<script>alert(1)</script>

```
<html>
<body>
<script>alert(1)</script> is fun!
</body>
</html>
```

Unsanitized HTML with injected JS



Injected JS is executed

XSS defense #1: HTML sanitization

POSTECH

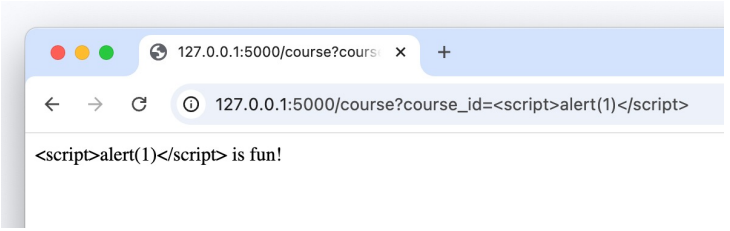
- HTML sanitization
  - Safe server, relying on Flask's **`escape( )`** sanitizer method

```
from flask import Flask, request, escape
app = Flask(__name__)

@app.route("/course")
def handle_course():
  cid = escape(request.args.get("course_id", ""))
  return f"<html><body>{cid} is fun!</body></html>"

if __name__ == "__main__":
  app.run(debug=True)
```

URL: .../course?course_id=<script>alert(1)</script>

```
<html>
<body>
&lt;script&gt;alert(1)&lt;/script&gt; is fun!
</body>
</html>
```

Sanitized HTML



Injected JS is not executed,
and is just rendered as plaintext

XSS defense #2: CSP

- Content Security Policy (CSP)
  - Idea: Instruct the browser to restrict which resources can be loaded and executed
    - Server delivers CSP in an HTTP header
  - Typical policy
    - Disallow all inline scripts, e.g., `<script></script>` within HTML
    - Allow scripts only from specified domains
  - Example

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' https://cdn.jsdelivr.net https://apis.google.com;
```

XSS defense #2: CSP

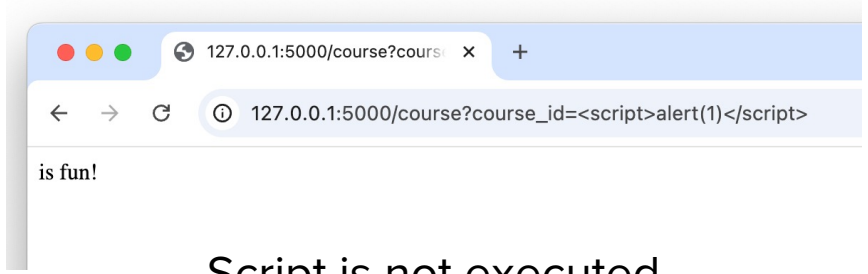
- Content Security Policy (CSP)
  - CSP header Example:
    - `default-src` disables inline script execution
    - `script-src` specifies allowed script sources

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' https://cdn.jsdelivr.net https://apis.google.com
```

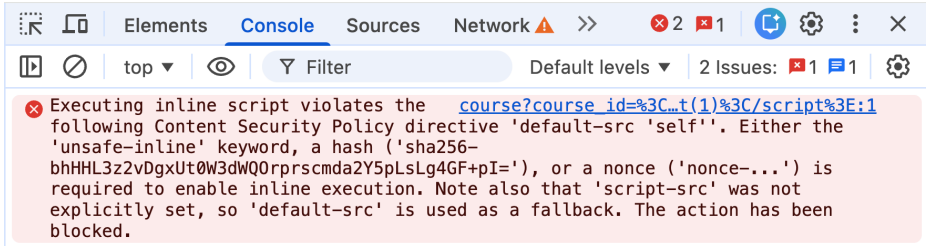
XSS defense #2: CSP

- Content Security Policy (CSP)
  - Applied to the vulnerable Flask server:

```python
from flask import Flask, request
app = Flask(__name__)

@app.after_request
def add_csp(response):
  response.headers["Content-Security-Policy"] = (
    "default-src 'self'; "
  )
  return response

@app.route("/course")
def handle_course():
  cid = request.args.get("course_id", "")
  return f"<html><body>{cid} is fun!</body></html>"

if __name__ == "__main__":
  app.run(debug=True)
```



Script is not executed



Console log

Summary

- CSRF and XSS are high-severity web vulnerabilities
  - CSRF exploits cookie-based authentication by tricking the victim's browser into sending a malicious request
  - XSS injects malicious JS into a legitimate website to make the victim's browser execute the script with the website's origin
- XSS is generally more powerful because it can often perform CSRF-like actions

Lessons:
Users should never trust the browser.
Servers should treat every client request as untrusted.

Q & A