

Review Session II

ECE 422: Reliable and Secure Systems Design



Instructor: An Ran Chen
Term: 2024 Winter

Schedule for today

- **Review session II**
 - Lecture 24 Cross Site Scripting
 - Lecture 25 Cross Site Scripting Prevention
 - Lecture 26 Content Security Policy
 - Lecture 27 CSP nonce and strict-dynamic
 - Lecture 28 Phishing and Denial-of-Service
 - Lecture 29 Blocks and blockchain
 - Lecture 30 Mining Principles
 - Lecture 31 Digital Signature & Double Spending Problem
- **Google CTF: Pasteurize Web**

Security principles

- Lecture 23 Cookies and Sessions
 - Ambient authority (cookies, IP checking, certificates, basic authentication)
 - Session cookies
 - Cookie protections
 - Signing cookies [Explanation]
 - Session fixation attack
 - Cookie attributes [Explanation]

Question on cookie attributes



Suppose that we build our course website on <https://ualberta.ca/course/ECE422>. We implement our authentication system by sending a response with a Set-Cookie HTTP header to set a **sessionId** cookie in the user's browser (e.g., "Set-Cookie: sessionId=1234;").

Question: Alice proposes to specify the Path attribute. Explain how it protects the website (e.g., "Set-Cookie: sessionId=1234; Path=/ECE422")?

Question on cookie attributes

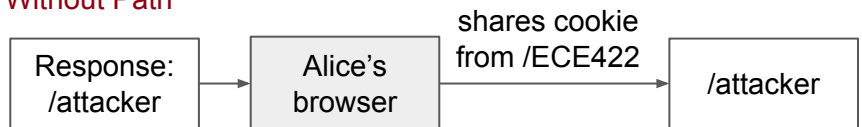


Question: Alice proposes to specify the Path attribute. Explain how it protects the website (e.g., "Set-Cookie: sessionId=1234; Path=/ECE422")?

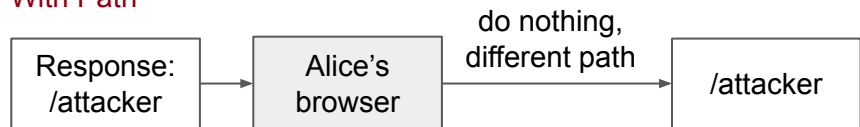
Answer: The cookie is **scoped to the path prefix /ECE422**.

- This implies that the cookie will be sent when the user visits <https://ualberta.ca/course/ECE422> or <https://ualberta.ca/course/ECE422/lectures>
- But not when they visit <https://ualberta.ca/course/attacker>
- **Preventing cookie sharing (session fixation attack)**

Without Path



With Path



Security principles

- Lecture 24 Cross Site Scripting
 - Cross-Site Request Forgery (CSRF) [Explanation]
 - Same Origin Policy (SOP) [Explanation]
 - Origin matching
 - Reflected vs Stored Cross-site scripting (XSS) [Explanation]

Question on web security



Question: "Mixing program control and user data" happens where an application accidentally treats user input as code and executes it. Which of the following attacks exploit this vulnerability? (Select all that apply)

- A. CSRF
- B. Stored XSS
- C. Reflected XSS
- D. IDN homograph attack
- E. All of the above

Question on web security



Question: "Mixing program control and user data" happens where an application accidentally treats user input as code and executes it. Which of the following attacks exploit this vulnerability? (Select all that apply)

- A. CSRF, exploit trusts from authenticated users
- B. Stored XSS, replace data with code, stored in database
- C. Reflected XSS, replace data with code, reflected as part of the response
- D. IDN homograph attack, phishing with a website that looks alike
- E. All of the above

Security principles

- Lecture 25 Cross Site Scripting Prevention
 - HttpOnly flag [Explanation]
 - HTML Escaping [Explanation]
- Lecture 26 Content Security Policy
 - CSP vs SOP: Whitelisting vs blocking data interaction
 - CSP directives [Problem]

Question on CSP



Suppose that an attacker injects a XSS payload into a HTML page sent by our web server:

```
<script>alert(document.cookie)</script>
```

Question: Given the following CSP, would the XSS attack succeed? Justify your answer.

```
Content-Security-Policy: default-src 'self';
```

Question on CSP



Suppose that an attacker injects a XSS payload into a HTML page sent by our web server:

```
<script>alert(document.cookie)</script>
```

Question: Given the following CSP, would the XSS attack succeed? Justify your answer.

```
Content-Security-Policy: default-src 'self';
```

Answer: No, because CSP checks the origin of the script. The default-src 'self' directive **only allows the script loaded from the same origin**. The above script is an **inline script** which is blocked.

Question on CSP



Given a CSP:

Content-Security-Policy: default-src 'self'; script-src 'self'; img-src 'self' https://img.example.com;

Question: Which of the following resources will be **blocked** (Select all that apply and read code carefully)?

- A. Resource A (link)
- B. Resource B (script)
- C. Resource C (img)

```
<!doctype html>
<html lang='en'>
  <head>
    <link rel='stylesheet' href='/style.css' />
  </head>
  <body>
    <script>alert>Welcome to ECE 422!</script>
    <h1>Honorable mention:</h1>
    <img src='https://img.example.org/cat.jpg'>
  </body>
</html>
```

(A)

(B)

(C)

Question on CSP



Given a CSP:

Content-Security-Policy: default-src 'self'; script-src 'self'; img-src 'self' https://img.example.com;

Question: Which of the following resources will be **blocked** (Select all that apply and read code carefully)?

- A. Resource A (link)
- B. Resource B (script)
- C. Resource C (img)

```
<!doctype html>
<html lang='en'>
  <head>
    <link rel='stylesheet' href='/style.css' />
  </head>
  <body>
    <script>alert>Welcome to ECE 422!</script>
    <h1>Honorable mention:</h1>
    <img src=https://img.example.org/cat.jpg>
  </body>
</html>
```

(A)

(B) Inline script

(C) Different origin

Common CSP directives and values



Definition of CSP: Content-Security-Policy: **directives** **'value'**;

Directives	Example	Description
default-src	default-src 'self'	Default policy to allow any resources (JavaScript, Fonts, CSS, etc) from the site's own origin
img-src	img-src *	Allow images from anywhere (* as wildcard)
script-src	script-src 'self' example.com	Allow scripts from its own origin and example.com

However, the **script-src** directive (when specified) **blocks inline scripts** and **event handling attributes**:

- Even if **'self'** is defined
- To allow this, use 'unsafe-inline' in script-src, but it is bad
 - Same as not having CSP

Security principles

- Lecture 27 CSP nonce and strict-dynamic
 - Challenge: Nested scripts
 - CSP nonces [Explanation]
 - CSP strict-dynamic [Explanation]
 - Read-only mode

Question on CSP with nonce



Suppose that an attacker injects a XSS payload into a HTML page sent by our web server:

```
<script>alert(document.cookie)</script>
```

Question: Given the following CSP, would the XSS attack succeed? Justify your answer.

```
Content-Security-Policy: script-src 'self' 'nonce-6301901420';
```


Question on CSP with nonce



Suppose that an attacker injects a XSS payload into a HTML page sent by our web server:

```
<script>alert(document.cookie)</script>
```

Question: Given the following CSP, would the XSS attack succeed? Justify your answer.

```
Content-Security-Policy: script-src 'self' 'nonce-6301901420';
```

Answer: No, because CSP checks the origin of the script. The script-src 'self' directive **only allows the script loaded from the same origin**, or **any script with the specified nonce as attribute** (e.g., `<script nonce='6301901420'>`). The above script is an inline script without nonce which is blocked.

Another example of CSP nonces



External script without CSP nonce

Content-Security-Policy:

```
script-src: 'self' https://*.google-analytics.com
```

```
<script  
src='https://www.google-analytics.com/  
analytics.js'></script>
```

External script with CSP nonce

Content-Security-Policy:

```
script-src: 'self' 'nonce-rAnd0m'
```

```
<script  
src='https://www.google-analytics.com/  
analytics.js' nonce='rAnd0m'></script>
```

- Same idea applies to external scripts
- Adding the nonce attribute to the script tag provides another solution to add <https://www.google-analytics.com/analytics.js> to CSP

Security principles

- Lecture 28 Phishing and Denial-of-Service
 - Phishing
 - Typosquatting
 - IDN Homograph attack [Explanation]
 - Pharming
 - Denial-of-Service attack
 - Goal of UI attacks: Override browser defaults, scareware, and annoy the user
 - Why some of the “features” of the Annoying Site work? [Explanation]
 - Link to CSP and SOP

Security principles

- Lecture 29 Blocks and blockchain [Explanation]
 - Blocks and blockchain
 - Block rewards and Bitcoin halving
- Lecture 30 Mining Principles
 - Proof-of-Work (PoW) [Explanation]
 - Difficulty adjustment [Problem]
 - The Longest Chain Rule
 - The 51% Attack
 - Finding the special number
- Lecture 31 Digital Signature & Double Spending Problem

Question on difficulty adjustment



Suppose that there are 5,000 Bitcoin miners on the Bitcoin network and their total computational power can achieve 6×10^{18} hash checks per hour.

Question: What should be the value of n in difficulty adjustment?

- Hint: $2^{59} = 5.8 \times 10^{17}$

Question on difficulty adjustment



Suppose that there are 5,000 Bitcoin miners on the Bitcoin network and their total computational power can achieve 6×10^{18} hash checks per hour.

Question: What should be the value of n in difficulty adjustment?

Thought process: A block is created every 10 minutes.

- Look for the number of hash checks in 10 minutes
- Find the difficulty based on the closest number of hash checks

Question on difficulty adjustment



Suppose that there are 5,000 Bitcoin miners on the Bitcoin network and their total computational power can achieve 6×10^{18} hash checks per hour.

Question: What should be the value of n in difficulty adjustment?

Step 1: Look for the number of hash checks in 10 minutes

- 6×10^{18} hash checks per hour $\rightarrow 10^{18}$ hash checks in 10 minutes

Step 2: Find the difficulty based on the closest number of hash checks

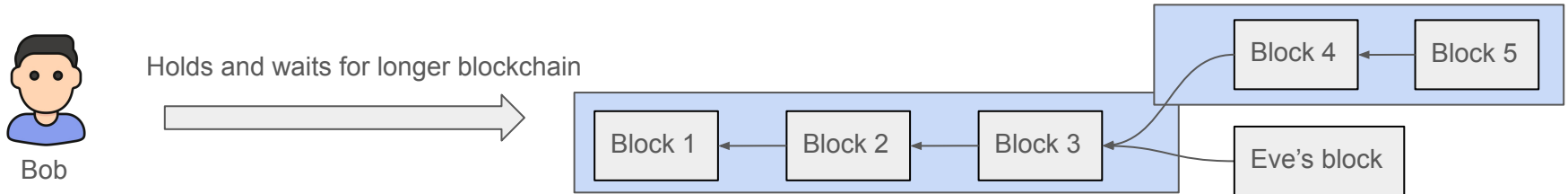
- Probability ($n=59$): 1 out of $(2^{59}) = 1$ out of 5.8×10^{17}
- Probability ($n=60$): 1 out of $(2^{60}) = 1$ out of 1.1×10^{18}
- **Difficulty should be adjusted to $n = 60$**

Why Proof-of-Work works?



Suppose Eve tries to send a block with fraudulent transactions:

- Eve first needs to find the special number based on the fraudulent transactions before everyone else, and broadcasts the blockchain
- Bob verifies the blockchain and copies it over
- **However**, Bob continues to listen to the broadcast
 - Any longer blockchain will replace the current one
 - For Bob to keep Eve's blockchain, Eve needs to keep extending the blockchain



Schedule for today

- Review session II
 - Lecture 24: Cross Site Scripting
 - Lecture 25 Cross Site Scripting Prevention
 - Lecture 26 Content Security Policy
 - Lecture 27 CSP nonce and strict-dynamic
 - Lecture 28 Phishing and Denial-of-Service
 - Lecture 29 Blocks and blockchain
 - Lecture 30 Mining Principles
 - Lecture 31 Digital Signature & Double Spending Problem
- Google CTF: Pasteurize Web

Google CTF

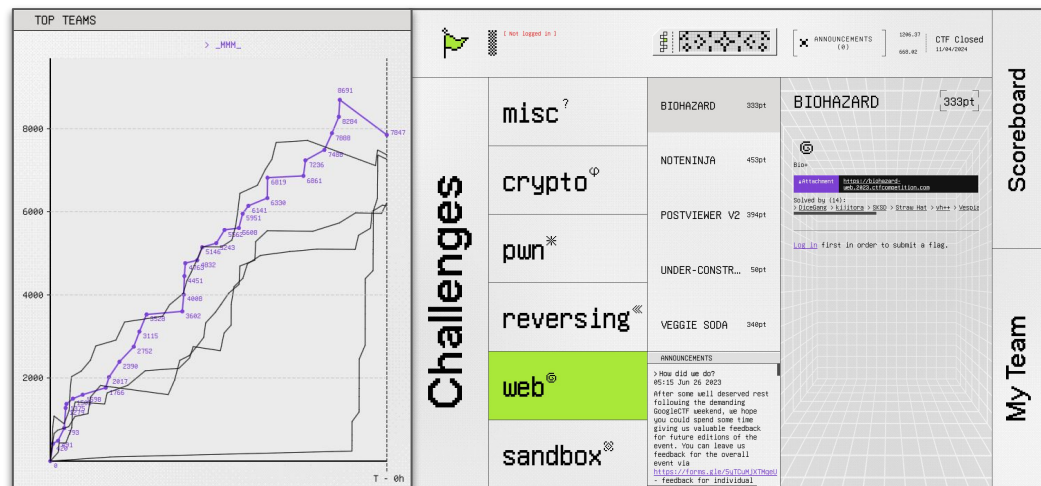
- Google CTF 2024: 21 June, 18:00 UTC — 23 June 2024, 18:00 UTC
 - Winners = greatest number of points earned
 - Points based on the number of teams that solved it
 - Vs Hackathon: time to sleep and eat

- Google CTF 2023

- Beginners quest

- Past Google CTF challenges

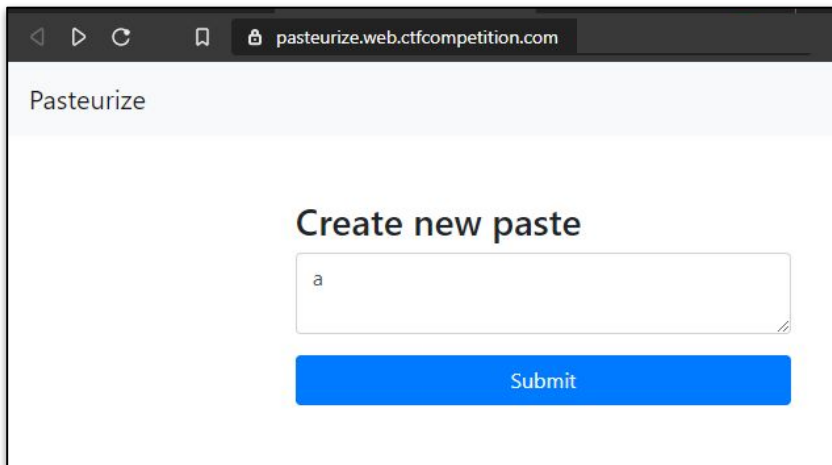
- From 2017 to 2023
 - Challenges and solutions



Google CTF: Pasteurize Web

XSS challenge in 2020: [Pasteurize web challenge + solution](#)

- Website (no longer available): create a new “Paste” and share it with a friend
- Difficulty: easy, 50 points



A screenshot of a web browser showing the 'Pasteurize' application. The address bar displays 'pasteurize.web.ctfcompetition.com'. The page has a light blue header with the word 'Pasteurize'. Below the header, the text 'Create new paste' is centered. Underneath, there is a text input field containing the letter 'a'. At the bottom of the form is a blue button labeled 'Submit'.



A screenshot of the same web browser showing the 'Pasteurize' application after a paste has been created. The address bar now shows a specific paste ID: 'pasteurize.web.ctfcompetition.com/adbc4b17-99a2...'. The page displays the paste ID 'adbc4b17-99a2-4ec4-95bf-ce8418c81ec0' in a light blue box. Below this, the text 'a' is shown. At the bottom, there are two buttons: 'share with TJMike' (with a key icon) and 'back'.

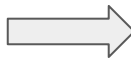
Google CTF: Pasteurize Web

Solution: Bypass the DOM purification

- Send a POST request with "?content[;alert()//]" as a query parameter

```
import request
```

```
request.post(challenge_url, data = {  
    "?content[;alert()//]"  
})
```



```
<script>  
    const note = ""; alert();//;  
    ...  
</script>
```

- **Upgrade 1:** Use automated script to send the POST request with a list of known vulnerabilities (basic loop)
 - Example: Github repository [XSS payload list](#)
 - ~ 2,600 XSS payloads

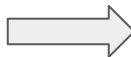
Google CTF: Pasteurize Web

Solution: Bypass the DOM purification

- Send a POST request with "?content[;alert()//]" as a query parameter

```
import request
```

```
request.post(challenge_url, data = {  
    "?content[;alert()//]"  
})
```



```
<script>  
    const note = ""; alert();//;  
    ...  
</script>
```

- **Upgrade 2:** Use Selenium to take a screenshot and compare HTML
 - Challenge: in real-world websites, XSS often leads to some hints rather than the actual vulnerability itself (e.g., malformed HTML rather than the successful injection)
 - In python, `driver.get_screenshot_as_file('site_xss_' + id + '.png')`

Be aware!



If you intend to participate in an upcoming CTF, remember to:

- **Be prepared**
 - Build a toolkit: automated scripts, static analysis scripts, list of vulnerabilities
 - Check the format: Jeopardy (challenges) vs Attack-Defense (defending vs attacking a host)
- Try to **team up with people with different backgrounds**
- **Sleep well and eat!**