22. Cross-Site Scripting (XSS)

XSS is a class of attacks where an attacker injects malicious JavaScript onto a webpage. When a victim user loads the webpage, the user's browser will run the malicious JavaScript.

XSS attacks are powerful because they subvert the same-origin policy. Normally, an attacker can only run JavaScript on websites they control (such as `https://evil.com`), so their JavaScript cannot affect websites with origins different from `https://evil.com`. However, if the attacker can inject JavaScript into `https://google.com`, then when a user loads `https://google.com`, their browser will run the attacker's JavaScript with the origin of `https://google.com`.

XSS attacks allow malicious JavaScript to run in the user's browser with the same origin as a legitimate website. This allows the attacker to perform any action the user can perform at `https://google.com` or steal any user secrets associated with Google and send them back to the attacker.

There are two main categories of XSS attacks: stored XSS and reflected XSS.

22.1. Stored XSS

In a stored XSS attack, the attacker finds a way to persistently store malicious JavaScript on the web server. When the victim loads the webpage, the web server will load this malicious JavaScript and display it to the user.

A classic example of stored XSS is a Facebook post. When a user makes a Facebook post, the contents of the post are stored on Facebook's servers, so that other users can load their friends' posts. If Facebook doesn't properly check user inputs, an attacker could make a post that says

```
<script>alert("XSS attack!")</script>
```

This post is now stored in Facebook's servers. If another user loads the attacker's posts, they will receive an HTML page with this script on it, and the browser will run the script and trigger a pop-up that says `XSS attack!`

22.2. Reflected XSS

In a reflected XSS attack, the attacker finds a vulnerable webpage where the server receives user input in an HTTP request and displays the user input in the response.

A classic example of reflected XSS is a Google search. When you make an HTTP GET request for a Google search, such as `https://www.google.com/search?&q=cs161`, the returned webpage with search results will include something like

```
You searched for: cs161
```

If Google does not properly check user input, an attacker could create a malicious URL `https://www.google.com/search?&q=<script>alert("XSS attack!")</script>`. When the victim loads this URL, Google will return

```
You searched for: <script>alert("XSS attack!")</script>
```

The victim's browser will run the script and trigger a pop-up that says `XSS attack!`

22.3. Defense: Sanitize Input

A good defense against XSS is checking for malicious input that might cause JavaScript to run, such as `<script>` tags. However, it is very difficult to write a good detector that catches all XSS attacks. For example, the following input causes JavaScript to run without ever using `<script>` tags:

```
<img src=1 href=1 onerror="JavaScript:alert("XSS attack!")" />
```

Just like SQL input escaping, sanitizing potentially dangerous input can be very tricky. For example, consider an escaper that searches for all instances of `<script>` and `</script>` and removes them. An attacker could provide this malicious input:

```
<scr<script>ipt>alert("XSS attack!")</scr<script>ipt>
```

After the escaper removes the two `<script>` tags it sees, the result is `<script>alert("XSS attack!")</script>`, and the attacker can still execute JavaScript!

Another way to escape input is to replace potentially dangerous characters with their HTML encoding. For example, the less than (`<`) and greater than (`>`) signs are encoded as `&lt;` and `&gt;`, respectively. These encodings cause less than and greater than signs to display on the webpage, without being interpreted as HTML.

Fortunately, there is a standardized set of sanitizations that is known to be robust.

22.4. Defense: Content Security Policy

Another XSS defense is using a content security policy (CSP) that specifies a list of allowed domains where scripts can be loaded from. For example, `cs161.org` might allow scripts that are loaded from `*.cs161.org` or `*.google.com` and disallow all other scripts, including any inline scripts that are injected by the attacker.

CSPs are defined by a web server and enforced by a browser. In the HTTP response, the server attaches a `Content-Security-Policy` header, and the browser checks any scripts against the header.

If you enable CSP, you can no longer run *any* scripts that are embedded directly in the HTML document. You can only load external scripts specified by the `script` tag and an external URL. These scripts can only be fetched from the sites specified in the CSP. This prevents an attacker from directly injecting scripts into an HTML document or modifying the HTML document to fetch scripts from the attacker's domain.

Further reading: OWASP Cheat Sheet on XSS