A3-Cross-Site Scripting (XSS)

Exploitability: AVERAGE | Prevalence: VERY WIDESPREAD | Detectability: EASY | Technical Impact: MODERATE

Description

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victims' browser, which can access any cookies, session tokens, or other sensitive information retained by the browser, or redirect user to malicious sites.

Attack Mechanics

There are two types of XSS flaws:

1. Reflected XSS: The malicious data is echoed back by the server in an immediate response to an HTTP request from the victim.
2. Stored XSS: The malicious data is stored on the server or on browser (using HTML5 local storage, for example), and later gets embedded in HTML page provided to the victim.

Each of reflected and stored XSS can occur on the server or on the client (which is also known as DOM based XSS), depending on when the malicious data gets injected in HTML markup.

How Do I Prevent It?

1. **Input validation and sanitization:** Input validation and data sanitization are the first line of defense against untrusted data. Apply white list validation wherever possible.
2. **Output encoding for correct context:** When a browser is rendering HTML and any other associated content like CSS, javascript etc., it follows different rendering rules for each context. Hence *Context-sensitive output encoding* is absolutely critical for mitigating risk of XSS.

   Here are the details about applying correct encoding in each context:

Context	Code Sample	Encoding Type

HTML Entity	UNTRUSTED DATA	Convert & to & Convert < to < Convert > to > Convert " to " Convert ' to ' Convert / to /
HTML Attribute Encoding	<input type="text" name="fname" value=" UNTRUSTED DATA">	Except for alphanumeric characters, escape all characters with the HTML Entity &#xHH; format, including spaces. (HH = Hex Value)
URI Encoding	clickme	Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the HTML Entity &#xHH; format, including spaces. (HH = Hex Value)
JavaScript Encoding	<script>var currentValue=' UNTRUSTED DATA'; </script> <script>someFunction(' UNTRUSTED DATA'); </script>	Ensure JavaScript variables are quoted. Except for alphanumeric characters, escape all characters with ASCII values less than 256 with \uXXXX unicode escaping format (X = Integer), or in xHH (HH = HEX Value) encoding format.
CSS Encoding	Selection	Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the \HH (HH= Hex Value) escaping format.

3. **HTTPOnly cookie flag:** Preventing all XSS flaws in an application is hard. To help mitigate the impact of an XSS flaw on your site, set the HTTPOnly flag on session cookie and any custom cookies that are not required to be accessed by JavaScript.
4. **Implement Content Security Policy (CSP):** CSP is a browser side mechanism which allows creating whitelists for client side resources used by the web application, e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources. For example, the CSP header below allows content only from example site's own domain (mydomain.com) and all its sub domains.

   ```
   Content-Security-Policy: default-src 'self' *.mydomain.com
   ```

5. **Apply encoding on both client and server side:** It is essential to apply encoding on both client and server side to mitigate DOM based XSS attack, in which untrusted data never leaves the browser.

Source: XSS Prevention Cheat Sheet[1]

Source Code Example

The demo web application is vulnerable to stored XSS attack on profiles form. On form submit, the first and last name field values are submitted to the server, and without any validation get saved in database. The values are then sent back to the browser without proper escaping to be shown at the top right menu.

Two measures can be taken to mitigate XSS risk:

1. In `server.js` , enable the HTML Encoding using template engine's auto escape flag.

```
swig.init({
    root: __dirname + "/app/views",
    autoescape: true //default value
});
```

2. Set HTTPOnly flag for session cookie while configuring the express session

```
// Enable session management using express middleware
app.use(express.session({
    secret: "s3Cur3",
    cookie: {
        httpOnly: true,
        secure: true
    }
}));
```

There were no additional contexts that needed encoding on the demo page; otherwise, it is necessary to encode for correct context depending on where data get placed at.

Output Encoding Context

An important observation when handling output encoding to prevent XSS is the notion of context.

When output encoding is performed, it must match the context in which it is being injected to. For example, if a user input is being injected to an HTML element then it will require different encoding semantics to escape malicious input than if it were injected to say an HTML attribute or a JavaScript context altogether (such as in a script tag).

An example for how to take advantage and exploit this mis-understanding exists on the profile page. See code references in `profile.js` and `profile.html`

Further Reading

1. XSS Prevention Cheat Sheet (https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)
2. Types of Cross-Site Scripting (https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting#Server_XS)
3. XSS Filter Evasion Cheat Sheet (https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet#STYLE_sheet)
4. Unraveling some of the Mysteries around DOM-based XSS (https://www.owasp.org/images/c/c5/Unraveling_some_Mysteries_around_DOM-based_XSS.pdf)