# XSS Prevention

By now, we should have a good understanding of what an XSS vulnerability is and its different types, how to detect an XSS vulnerability, and how to exploit XSS vulnerabilities. We will conclude the module by learning how to defend against XSS vulnerabilities.

As discussed previously, XSS vulnerabilities are mainly linked to two parts of the web application: A `Source` like a user input field and a `Sink` that displays the input data. These are the main two points that we should focus on securing, both in the front-end and in the back-end.

The most important aspect of preventing XSS vulnerabilities is proper input sanitization and validation on both the front and back end. In addition to that, other security measures can be taken to help prevent XSS attacks.

## Front-end

As the front-end of the web application is where most (but not all) of the user input is taken from, it is essential to sanitize and validate the user input on the front-end using JavaScript.

### Input Validation

For example, in the exercise of the `XSS Discovery` section, we saw that the web application will not allow us to submit the form if the email format is invalid. This was done with the following JavaScript code:

Code: javascript

```
function validateEmail(email) {
    const re = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
    return re.test($("#login input[name=email]").val());
}
```

As we can see, this code is testing the `email` input field and returning `true` or `false` whether it matches the Regex validation of an email format.

### Input Sanitization

In addition to input validation, we should always ensure that we do not allow any input with JavaScript code in it, by escaping any special characters. For this, we can utilize the DOMPurify JavaScript library, as follows:

Code: javascript

```
<script type="text/javascript" src="dist/purify.min.js"></script>
let clean = DOMPurify.sanitize( dirty );
```

This will escape any special characters with a backslash `\`, which should help ensure that a user does not send any input with special characters (like JavaScript code), which should prevent vulnerabilities like DOM XSS.

### Direct Input

Finally, we should always ensure that we never use user input directly within certain HTML tags, like:

1. JavaScript code `<script></script>`
2. CSS Style Code `<style></style>`
3. Tag/Attribute Fields `<div name='INPUT'></div>`
4. HTML Comments `<!-- -->`

If user input goes into any of the above examples, it can inject malicious JavaScript code, which may lead to an XSS vulnerability. In addition to this, we should avoid using JavaScript functions that allow changing raw text of HTML fields, like:

- `DOM.innerHTML`
- `DOM.outerHTML`
- `document.write()`
- `document.writeln()`
- `document.domain`

And the following jQuery functions:

- `html()`
- `parseHTML()`
- `add()`
- `append()`
- `prepend()`
- `after()`
- `insertAfter()`
- `before()`
- `insertBefore()`
- `replaceAll()`
- `replaceWith()`

As these functions write raw text to the HTML code, if any user input goes into them, it may include malicious JavaScript code, which leads to an XSS vulnerability.

## Back-end

On the other end, we should also ensure that we prevent XSS vulnerabilities with measures on the back-end to prevent Stored and Reflected XSS vulnerabilities. As we saw in the `XSS Discovery` section exercise, even though it had front-end input validation, this was not enough to prevent us from injecting a malicious payload into the form. So, we should have XSS prevention measures on the back-end as well. This can be achieved with Input and Output Sanitization and Validation, Server Configuration, and Back-end Tools that help prevent XSS vulnerabilities.

### Input Validation

Input validation in the back-end is quite similar to the front-end, and it uses Regex or library functions to ensure that the input field is what is expected. If it does not match, then the back-end server will reject it and not display it.

An example of E-Mail validation on a PHP back-end is the following:

Code: **php**

```php
if (filter_var($_GET['email'], FILTER_VALIDATE_EMAIL)) {
    // do task
} else {
    // reject input - do not display it
}
```

For a NodeJS back-end, we can use the same JavaScript code mentioned earlier for the front-end.

### Input Sanitization

When it comes to input sanitization, then the back-end plays a vital role, as front-end input sanitization can be easily bypassed by sending custom `GET` or `POST` requests. Luckily, there are very strong libraries for various back-end languages that can properly sanitize any user input, such that we ensure that no injection can occur.

For example, for a PHP back-end, we can use the `addslashes` function to sanitize user input by escaping special characters with a backslash:

Code: **php**

```php
addslashes($_GET['email'])
```

In any case, direct user input (e.g. `$_GET['email']`) should never be directly displayed on the page, as this can lead to XSS vulnerabilities.

For a NodeJS back-end, we can also use the DOMPurify library as we did with the front-end, as follows:

Code: **javascript**

```javascript
import DOMPurify from 'dompurify';
var clean = DOMPurify.sanitize(dirty);
```

### Output HTML Encoding

Another important aspect to pay attention to in the back-end is `Output Encoding`. This means that we have to encode any special characters into their HTML codes, which is helpful if we need to display the entire user input without introducing an XSS vulnerability. For a PHP back-end, we can use the `htmlspecialchars` or the `htmlentities` functions, which would encode certain special characters into their HTML codes (e.g. `<` into `<`), so the browser will display them correctly, but they will not cause any injection of any sort:

Code: **php**

```php
htmlentities($_GET['email']);
```

For a NodeJS back-end, we can use any library that does HTML encoding, like `html-entities`, as follows:

Code: **javascript**

```javascript
import encode from 'html-entities';
encode('<'); // -> '&lt;'
```

Once we ensure that all user input is validated, sanitized, and encoded on output, we should significantly reduce the risk of having XSS vulnerabilities.

### Server Configuration

In addition to the above, there are certain back-end web server configurations that may help in preventing XSS attacks, such as:

- Using HTTPS across the entire domain.
- Using XSS prevention headers.
- Using the appropriate Content-Type for the page, like `X-Content-Type-Options=nosniff`.
- Using `Content-Security-Policy` options, like `script-src 'self'`, which only allows locally hosted scripts.
- Using the `HttpOnly` and `Secure` cookie flags to prevent JavaScript from reading cookies and only transport them over HTTPS.

In addition to the above, having a good `Web Application Firewall (WAF)` can significantly reduce the chances of XSS exploitation, as it will automatically detect any type of injection going through HTTP requests and will automatically reject such requests. Furthermore, some frameworks provide built-in XSS protection, like ASP.NET.

In the end, we must do our best to secure our web applications against XSS vulnerabilities using such XSS prevention techniques. Even after all of this is done, we should practice all of the skills we learned in this module and attempt to identify and exploit XSS vulnerabilities in any potential input fields, as secure coding and secure configurations may still leave gaps and vulnerabilities that can be exploited. If we practice defending the website using both `offensive` and `defensive` techniques, we should reach a reliable level of security against XSS vulnerabilities.