Gatsby's architecture provides several security benefits relative to traditional website development:

- Because Gatsby compiles your site to flat files, rather than having running app servers and databases, it reduces the attack surface of the site to outsiders.
- Gatsby adds a layer of indirection which obscures your CMS — so even if your CMS is vulnerable, bad actors have no idea where to find it. This is in contrast to systems where bad actors can easily locate the admin dashboard at, e.g., `/wp-admin` and attempt to hack in.
- With Gatsby, you serve your site from a global CDN e.g. Akamai, Cloudflare, Fastly, etc., which effectively eliminates the risk of DDoS attacks.

However, there are still a couple of coding patterns you need to watch out for when building your Gatsby site:

Cross-Site Scripting (XSS)

Cross-Site Scripting is a type of attack that injects a script or an unexpected link to another site into the client side of the application.

JSX elements automatically escape HTML tags by design. See the following example:

```
// highlight-next-line
const username = `<script src='https://path/to/badness.js'></script>`

const User = () => <p> hello {username}</p>
```

When you try to inject the JSX element above, it will render `hello <script src='https://path/to/badness.js'></script>` as a string inside the `p` tag.

On the other hand, fields in your application may need to render inner HTML tags, such as a content field in a blog or a comment in a comments section, that are built into rich-text editors.

In order to render those HTML tags you need to use an HTML parser (e.g. html-react-parser) or the `dangerouslySetInnerHTML` prop, like so:

```
const CommentRenderer = comment => (
  // highlight-next-line
  <p dangerouslySetInnerHTML={{ __html: comment }} />
) // dangerous indeed.
```

That is when you expose your application to XSS attacks.

How can you prevent cross-site scripting?

The most straightforward way to prevent a XSS attack is to sanitize the innerHTML string before dangerously setting it. Fortunately, there are npm packages that can accomplish this; packages like sanitize-html and DOMPurify.

Cross-Site Request Forgery (CSRF)

Cross-Site request forgery is a type of exploit that deceives the browser into executing unauthorized actions. By default, in any request made, the browser automatically appends any stored cookies of the destination domain. Combining this with a crafted request, a malicious website can read and write data without the user's action or knowledge.

For example, assume that the comments in your blog are sent in a form similar to this one:

```
<form action="http://mywebsite.com/blog/addcomment" method="POST">
  <input type="text" name="comment" />
  <input type="submit" />
</form>
```

A malicious website could inspect your site and copy this snippet to theirs. If the user is logged in, the associated cookies are sent with the form and the server cannot distinguish the origin of it. Even worse, the form could be sent when the page loads with information you don't control:

```
// highlight-next-line
<body onload="document.csrf.submit()">
  <!-- ... -->
  <form action="http://mywebsite.com/blog/addcomment" method="POST" name="csrf">
    // highlight-next-line
    <input
      type="hidden"
      name="comment"
      value="Hey visit http://maliciouswebsite.com, it's pretty nice"
    />
    <input type="submit" />
  </form>
</body>
```

How can you prevent cross-site request forgery?

Don't use GET requests to modify data

Actions that do not read data should be handled in a POST request. In the example above, if the `/blog/addcomment` endpoint accepts a GET request, the CSRF attack can be done using an `<img />` tag:

```
<img src="http://mywebsite.com/blog/addcomment?comment=unwanted%20comment" />
```

CSRF Tokens

If you want to protect a page your server will provide an encrypted, hard to guess **token**. That token is tied to a user's session and must be included in every POST request. See the following example:

```
<form action="http://mywebsite.com/blog/addcoment" method="POST">
  <input type="text" name="comment" />
  // highlight-next-line
  <input type="hidden" name="token" value="SU9J3tMoT1w5q6uJ1VMXaaf9UXzLvyNd" />
  <input type="submit" />
</form>
```

When the form is sent, the server will compare the token received with the stored token and block the action if they are not the same. Make sure that malicious websites don't have access to the CSRF token by using HTTP Access Control.

Same-Site Cookies Attribute

This cookie attribute is targeted to prevent CSRF attacks. If you need to create a cookie in your application, make sure to protect them by using this attribute, that could be of `Strict` or `Lax` type:

```
Set-Cookie: example=1; SameSite=Strict
```

Using the `SameSite` attribute allows the server to make sure that the cookies are not being sent by a **cross-site** domain request. Check out [MDN Docs]() for more information on configuring a cookie. You will also want to note current browser support which is available on the [Can I Use page]().

This cookie attribute is not a replacement for a CSRF Token (and vice-versa). They can work together as security layers in your website. Otherwise, a Cross-Site Scripting attack can be used to defeat these CSRF mitigation techniques. Check out [OWASP CSRF prevention cheat sheet]() for more information.

Third-party Scripts

Some third-party scripts like Google Tag Manager give you the ability to [add arbitrary JavaScript]() to your site. This helps integrate third-party tools but can be misused to inject malicious code. To avoid this, be sure to [control access]() to these services.

Check Your Dependencies

In your Gatsby project, you are going to have some dependencies that get stored in `node_modules/`. Therefore, it is important to check if any of them, or their dependencies, have security vulnerabilities.

Using npm

In npm, you can use the `npm audit` command to check your dependencies. This command is available in all npm versions greater than `6.0.0`. Check [npm docs]() for more options.

Using yarn

Similar to npm, you can use the `yarn audit` command. It is available starting with version `1.12.0` though it is not yet available in version 2. Check the [yarn docs]() for more options.

Key Security

Gatsby allows you to [fetch data from various APIs]() and those APIs often require a key to access them. These keys should be stored in your build environment using [Environment Variables](). See the following example for fetching data from GitHub with an Authorization Header:

```
{
  resolve: "gatsby-source-graphql",
  options: {
    typeName: "GitHub",
    fieldName: "github",
    url: "https://api.github.com/graphql",
    headers: {
      // highlight-next-line
      Authorization: `Bearer ${process.env.GITHUB_TOKEN}`,
      },
    }
  }
```

Storing keys in client-side

Sometimes in your Gatsby website, you will need display sensitive data or handle authenticated routes (e.g. a page that shows a user's orders in your ecommerce). Gatsby has an Authentication Tutorial if you need assistance with setting up authentication flow. Use cookies to store the credentials client-side, preferably with the `SameSite` attribute listed above. Check out MDN Docs to further understand these attributes and how to configure them.

Content Security Policy (CSP)

Content Security Policy is a security layer added in web applications to detect and prevent attacks, e.g. the XSS attack mentioned above.

To add it to your Gatsby website, add gatsby-plugin-csp to your `gatsby-config.js` with the desired configuration. Note that currently there is a compatibility issue between gatsby-plugin-csp and other plugins that generate hashes in inline styles, including gatsby-plugin-image.

> *Note that not all browsers support CSP, check can-i-use for more information.*

Other Resources

- Security for Modern Web Frameworks
- Docs React: DOM Elements
- OWASP XSS filter evasion cheatsheet
- OWASP CSRF prevention cheat sheet
- Warn for JavaScript: URLs in DOM sinks #15047
- How to prevent XSS attacks when using `dangerouslySetInnerHTML` in React
- Exploiting XSS via Markdown
- Auditing package dependencies for security vulnerabilities
- CSRF tokens
- SameSite cookies explained