Securing Cascading Style Sheets Cheat Sheet

Introduction

The goal of this `CSS` (Not XSS, but Cascading Style Sheet) Cheat Sheet is to inform Programmers, Testers, Security Analysts, Front-End Developers and anyone who is interested in Web Application Security to use these recommendations or requirements in order to achieve better security when authoring `Cascading Style Sheets`.

Let's demonstrate this risk with an example:

Santhosh is a programmer who works for a company called **X** and authors a Cascading Style Sheet to implement styling of the web application. The application for which he is writing CSS Code has various roles like **Student**, **Teacher**, **Super User** & **Administrator** and these roles have different permissions (PBAC - Permission Based Access Control) and Roles (RBAC - Role Based Access Control). Not only do these roles have different access controls, but these roles could also have different styling for webpages that might be specific to an individual or group of roles.

Santhosh thinks that it would be a great optimized idea to create a "global styling" CSS file which has all the CSS styling/selectors for all of the roles. According to their role, a specific feature or user interface element will be rendered. For instance, Administrator will have different features compared to **Student** or **Teacher** or **SuperUser**. However, some permissions or features maybe common to some roles.

Example: Profile Settings will be applicable to all the users here while *Adding Users* or *Deleting Users* is only applicable for **Administrator**.

Example:

- `.login`
- `.profileStudent`

- `.changePassword`
- `.addUsers`
- `.deleteUsers`
- `.addNewAdmin`
- `.deleteAdmin`
- `.exportUserData`
- `.exportProfileData`
- ...

Now, let's examine what are the risks associated with this style of coding.

Risk #1

Motivated Attackers always take a look at `*.CSS` files to learn the features of the application even without being logged in.

For instance: Jim is a motivated attacker and always tries to look into CSS files from the View-Source even before other attacks. When Jim looks into the CSS file, they see that there are different features and different roles based on the CSS selectors like `.profileSettings,` `.editUser,` `.addUser,` `.deleteUser` and so on. Jim can use the CSS for intel gathering to help gain access to sensitive roles. This is a form of attacker due diligence even before trying to perform dangerous attacks to gain access to the web application.

In a nutshell, having global styling could reveal sensitive information that could be beneficial to the attacker.

Risk #2

Let's say, Santhosh has this habit of writing the descriptive selector names like `.profileSettings,` `exportUserData,` `.changePassword,` `.oldPassword,` `.newPassword,` `.confirmNewPassword` etc. Good programmers like to keep code readable and usable by other Code Reviewers of the team. The risk is that attackers could map these selectors to actual features of a web application.

Defensive Mechanisms to Mitigate Attacker's Motivation

Defense Mechanism #1

As a CSS Coder / Programmer, always keep the CSS isolated by access control level. By this, it means **Student** will have a different CSS file called as `StudentStyling.CSS` while **Administrator** has `AdministratorStyling.CSS` and so on. Make sure these `*.CSS` files are accessed only for a user with the proper access control level. Only users with the proper access control level should be able to access their `*.CSS` file.

If an authenticated user with the **Student** Role tries to access `AdministratorStyling.CSS` through forced browsing, an alert that an intrusion is occurring should be recorded.

Defense Mechanism #2

Another option is to modify your CSS files to remove any identifying information. As a general rule, it's recommended that your website have a consistent style between pages, and it's best to write your general CSS rules in such a way that they apply across multiple pages. This reduces the need for specific selectors in the first place. Furthermore, it's often possible to create CSS selectors that target specific HTML elements without using IDs or class names. For example, `#UserPage .Toolbar .addUserButton` could be rewritten to something more obscure such as `#page_u header button:first-of-type`.

Build-time and runtime tools also exist, which can be integrated to obfuscate your class names. This can reduce the chance of an attacker guessing the features of your application. Some examples:

- JSS (CSS in JS) has a `minify` option which would generate class names such as `.c001`, `.c002`.
- CSS Modules has a `modules` and `localIdentName` option, which functions similarly to JSS, but allows importing any CSS file without major structural changes to your application.

- .Net Blazor CSS Isolation can be used to scope your CSS to the component it's used in, and results in selectors like `button.add[b-3xxtam6d07]`.
- CSS libraries such as Bootstrap and Tailwind can reduce the need for specific CSS selectors as they provide a strong base theme to work from.

Defense Mechanism #3

Web applications that allow users to author content via HTML input could be vulnerable to malicious use of CSS. Uploaded HTML could use styles that are allowed by the web application but could be used for purposes other than intended which could lead to security risks.

Example: You can read about how LinkedIn had a vulnerability which allowed malicious use of CSS to execute a Clickjacking attack. This caused the document to enter a state where clicking anywhere on the page would result in loading a potentially malicious website. You can read more about mitigating clickjacking attacks here.