SCENARIO

The application contains a DOM based clobbering vulnerability where the comment functionality allows "safe" HTML. We’ll try to exploit this vulnerability by forcing the user to inject a cookie that will cause XSS on the page and call the alert() function.

**PROCEDURE**

1. Access the application and go to any blog post.
2. Then add the Payload 1 into the comment section of that blog.
3. Return to the blog post and create a second comment containing any random text. The next time the page loads, the alert() function is called.
4. The page for a specific blog post imports the JavaScript file loadCommentsWithDomPurify.js, which contains the following code:

let defaultAvatar = window.defaultAvatar || {avatar: '/resources/images/avatarDefault.svg'}

The defaultAvatar object is implemented using this dangerous pattern containing the logical OR operator in conjunction with a global variable. This makes it vulnerable to DOM clobbering. You can clobber this object using anchor tags. Creating two anchors with the same ID causes them to be grouped in a DOM collection. The name attribute in the second anchor contains the value "avatar", which will clobber the avatar property with the contents of the href attribute. Notice that the site uses the DOMPurify filter in an attempt to reduce DOM-based vulnerabilities. However, DOMPurify allows you to use the cid: protocol, which does not URL-encode double-quotes. This means you can inject an encoded double-quote that will be decoded at runtime. As a result, the injection described above will cause the defaultAvatar variable to be assigned the clobbered property {avatar: ‘cid:"onerror=alert(1)//’} the next time the page is loaded. When you make a second post, the browser uses the newly-clobbered global variable, which smuggles the payload in the onerror event handler and triggers the alert().

**PAYLOAD**

<a id=defaultAvatar><a id=defaultAvatar name=avatar href="cid:&quot;onerror=alert(1)//">

**REMEDIATION**

1. **Avoid Direct Global References:** Do not use global DOM references (like window.someVariable). It's better to use more explicit references such as document.getElementById('someID') to avoid overwriting by DOM elements with the same name or ID.
2. **Sanitize Input with Caution:** DOMPurify allows the creation of certain elements like <a>, but it's still possible to misuse these in DOM clobbering attacks. Review the configuration of your sanitization library to ensure that only absolutely necessary tags and attributes are allowed.
3. **Separate Data from Code:** Instead of injecting data directly into your code, separate the data and process it separately. This will ensure that malicious inputs don't get executed inadvertently.
4. **Use Non-Clobberable Properties:** Instead of using properties that can be clobbered (like window.defaultAvatar), use properties/methods that aren't prone to DOM clobbering, like document.querySelector().
5. **Content Security Policy (CSP):** Implement a strict CSP. It will make it harder for attackers to run malicious code even if they manage to inject something into the DOM.