Implementing Protections Against Cross-Site Scripting and Cross-Site Request Forgery Attacks

Brandon Rothweiler, Scott Enriquez, Javier Serrano

Goal of This Project

Our team's goal was to write our own implementations of the protections that are commonly used to defend against XSS and CSRF attacks. Since the protections that exist are already fairly effective, we did not intend on creating a new method of defense. Rather, we simply recreated the defenses from scratch, in a form that can be generalized and applied to a wide variety of web applications. The protections we implemented include input sanitization to prevent XSS, and token generation and verification with HTTP requests to prevent CSRF. We also created a vulnerable web application to demonstrate how these attacks can be performed and exploited in order to manipulate or hijack a user's session.

Evaluation of Success

The project will be successful if our implementations of the protections are effective in mitigating or removing the vulnerabilities that we are able to demonstrate can exist in web applications.

Previous Approaches

There already exist several effective methods of protecting against XSS and CSRF attacks in web applications. One way to protect against XSS is to sanitize any user input that the application inserts into web pages. In addition, if the web developer is careful to only place user input in specific locations within the HTML, it is possible to avoid the use of sanitization. For CSRF, the most common defense is by using tokens which prevent attackers from making valid requests on the user's behalf. It is also common to protect cookies, for example by using an HttpOnly flag.

Background

Node.js is widely used for developing web applications, and many of the most popular websites on the Internet use it. Express is the most popular framework for building web applications in Node.js. It is for these reasons that we decided to implement our protections in the form of

Express middleware. By building it this way, we are able to apply our protections to any Express application by importing our independent modules.

In Express, a middleware function is a function which assists in the processing of incoming requests. Once a middleware function is attached to a web application, it will run for each request as the request is being processed. Middleware functions can manipulate the parameters or data stored in a request, and they can also manipulate the response before it is sent to the client. Since JavaScript is a functional language, a middleware function is simply an object, and therefore can be imported from an external module. These aspects made Express middleware functions an ideal medium for our implementation of the protections.

It is also necessary to provide background about Express template engines. Express is compatible with template engines which allow web developers to reuse parts of HTML objects while inserting text fragments dynamically into specific parts of the page. Typically, when a web request is being handled, the web application would make a database query to retrieve the information to be displayed on the page. Once the information is retrieved, the application calls an Express `render` function which generates the HTML to be delivered to the user's web browser. The text fragments to be inserted into the page are passed as arguments to the `render` function so that it can place them into the correct locations within the HTML.

Implementation

For our XSS protection module, we created a middleware function which replaces Express's built-in `render` function with a custom one created by us. Our custom `render` function takes the text fragments as arguments, sanitizes them, and then passes the sanitized text to the original built-in `render` function. The sanitization is performed by replacing all characters in the text with their html-encoded equivalents. This causes the browser to render them as text on the page rather than parsing them as HTML. It is not necessary to replace all characters in the text, but determining which characters must be replaced requires knowledge of where on the page the text is going to be inserted, which is not something that can be determined from the context of an Express middleware function.

For our CSRF protection module, rather than overwriting Express's `render` function, we overwrite the `send` function, which handles sending a response back to the user that issued a request. The function we write intercepts the `send` function before any object is sent back to the user. If the object to be sent is an HTML document, it is parsed and any forms in the page are located. A session-specific token is generated for the user and inserted as a hidden field into each form on the page, and the altered HTML document proceeds to be sent to the user. The token is a bcrypt hash of the session ID of the user and a server-side secret, so that even if the session ID can be stolen, the token should be secure. The module also intercepts each incoming request. If it is a POST request, our middleware verifies that there is a CSRF token

present with the request, and that it matches the session-specific token generated for the requesting user. If so, the request is made, otherwise an HTML error code is returned.

Testing

To test these modules, we built a simple blog website that allows users to sign in and make posts that will be visible to everyone on the website. To test our XSS protection we used an account to make a post containing a HTML `<script>` tag that would run code for anyone who visited the page. With our XSS protection enabled, the text was sanitized and the tag's contents were displayed instead of being run. To test CSRF protection we ran another server that serves a page that makes an AJAX request to the blog to post on the user's page. When the middleware is enabled, this request is checked for a CSRF token that matches the user's. Since there is not, they will receive a HTML 400 error code and the blog post will not be posted.

Conclusion

We attempted to find existing Express applications online that we could use to demonstrate our modules working with applications that we did not create ourselves, but unsurprisingly it is difficult to find any such applications that use Express, use a templating engine, and are left vulnerable to these types of attacks. In fact, if an Express application were left vulnerable to these attacks, it would probably have to be intentionally so. The application that we created had to explicitly use insecure features and settings that were different from the default in order to showcase the vulnerabilities and include our own protections. All of the most popular packages used in Express applications have these security features built in already. We do not feel this takes away from our project, however. The risk of these attacks still exist, it's just that everyone else already recognizes that and have taken their own steps to protect their projects.