Web Applications [2023/2024]

Lab 7: Fetching data from server and experimenting with XSS

In this lab, you will interact with the server using the browsers' fetch API. Additionally, a simple Cross-Site Scripting (XSS) case will be analyzed and prevented.

1. Fetch the Film Library data from the server.

Starting from the outcome of previous labs, take the server API and add a folder for static content. Copy the client part developed in the previous lab and add it to such folder. Add an instruction in the index.js to serve such static content under a certain path, e.g., `/static`, using the `express.static()` middleware.

Run the server and check that everything is working by loading the URL corresponding to the HTML page in the browser, e.g., `http://localhost:3001/static/index.html`

Now modify the JS code to load the list of films from the server, using the `fetch()` function available in the JS browser environment. Use the API endpoint that you already developed from the server side (for instance, `/api/films`). While developing this modification, you can separate the code into different modules. Write the function containing the `fetch()` into a different file (for instance, `API.js`), and import it from the main JS file. Also, if you defined functions such as **Film** and **FilmLibrary** to mimic the object-oriented programming style, put such functions in a different file and import them into the main and the fetch JS file, since you probably need to use them in both places.

Check that everything is working by reloading the URL and observing, via the browser developer tools and the logs on the server, that the GET call to `/api/films` actually takes place and returns the JSON you expect.

2. Test and prevent simple XSS.

Modify the JS code of your client so that HTML tags can be used and displayed in the title of your films. For instance, `<b>` for bold text and `<i>` for italics. To make them work, use the `innerHTML` property to set the content of the tag that displays the title. To test that the HTML code has the desired effect, modify some records in the database by changing the title into text surrounded by an HTML tag, such as `<i>Matrix</i>`. Reload the page and check that the text formatting has been applied.

Now, assume an attacker has compromised your database by inserting the following string as title of a film:

```
<img src="/" onerror="alert('hacked xss');">
```

Modify the title of a film in your database by inserting such string. Check that the code (`alert` …) is indeed executed and it displays an alert window.

For your convenience, a database file which already includes such string is provided within the folder corresponding to this lab with the name `films_xss.db`

Now, prevent this from happening by modifying the content returned by the server, making it safe for displaying. Use the `DOMPurify` library to sanitize the text before returning it from the server. Check that the XSS behavior is now prevented.

Finally, note that sanitizing before returning data from the server is tedious and the risk of forgetting a place where sanitization is real. This is because, typically, there are more places where data is read than where it is written. So, if possible, sanitize data before storing them in the database.