SQE OWASP ACTIVITY

Harshavardhan - 2021111003 (individual submission)

This report will contain all the exploits/vulnerabilities I had found in the OWASP Juice Shop website. Each vulnerability solved will be explained in detail with the steps to reproduce the vulnerability and solutions to fix the vulnerability.

The codebase is available at GitHub

Vulnerabilities and Fixes

1. DOM XSS (Cross-Site Scripting)

This vulnerability is found in the search bar of the website. The search bar is vulnerable to DOM XSS. The search bar is vulnerable to XSS because the search bar is not sanitizing the input and directly rendering the input to the page. This can be exploited by entering a script in the search bar and the script will be executed when the search results are displayed.

I ran the recommended script `<iframe src="javascript:alert(`xss`)">`

Fix

The input from the search bar should be sanitized before rendering it to the page.

```
function escapeHtml(unsafe) {
      return unsafe
        .replace(/&/g, "&amp;")
        .replace(/</g, "&lt;")
        .replace(/>/g, "&gt;")
        .replace(/"/g, "&quot;")
        .replace(/'/g, "&#039;");
    }
```

Parsing the input through the escapeHtml function will sanitize the input and remove any malicious code, while still displaying the input as it is.

2. Login Admin (Injection)

This is a very simple and straightforward exploit, where we can simply enter the username as injected SQL query to get past the security check. We simply need to make sure that the query returns `True` in the server side to get past the login page. We can do this by entering the username as `' or 1=1 --` and any password. This works because the query will return `True` no matter what since 1=1 is always true and we use an OR operator. Also the `--` is used to comment out the rest of the query related to password.

Fix

The input from the username field should be sanitized before executing the query. Check for single quotes as usernames and passwords should not contain single quotes.

```
for (const key in req.body) {
    if (req.body.hasOwnProperty(key)) {
        if (typeof req.body[key] === "string") {
            req.body[key] = req.body[key].replace(/'/g, "");
        }
    }
}
```

This code will remove any single quotes from the input and prevent SQL injection. This is required to be done in the backend, as direct api calls can be made to the server.

3. Repititive Registration (Improper Input Validation)

I found this vulnerability while trying to create multiple user for a different task, where the confirm/reenter password field is not validated properly. When we enter any value to the confirm password field, it checks if it matches the current state of the password field. and if it matches it is fine, but if we change the password field after changing the confirm password field, the confirm password field does not get updated and the form can be submitted.

To fix this issue, the confirm password field should be updated whenever the password field is updated. The confirm password field should be validated against the password field when the form is submitted.

Fix

```
function matchValidator(passwordControl: AbstractControl) {
  return function matchOtherValidate(
    repeatPasswordControl: UntypedFormControl
  ) {
    const password = passwordControl.value;
    const passwordRepeat = repeatPasswordControl.value;
    if (password !== passwordRepeat) {
      return { notSame: true };
    }
    return null;
  };
}

function matchValidator2(repeatPasswordControl: AbstractControl) {
  return function matchOtherValidate2(passwordControl: UntypedFormControl)
{
    const password = passwordControl.value;
    const passwordRepeat = repeatPasswordControl.value;
    if (password !== passwordRepeat) {
      return { notSame: true };
    }
    return null;
    };
}
```

The above code will validate the confirm password field against the password field and return an error if they do not match. This will prevent the form from being submitted if the passwords do not match. The only change being from the initial implementation is that the confirm password field is updated whenever the password field is updated, and not just one way.
We do require more minor changes to add the `matchValidator2` to the passwordControl similar to the `matchValidator` to the repeatPasswordControl.

4. Captcha Bypass (Broken Anti-Automation)

Captchas are used to prevent bots from creating requests to the server. But in this case, the captcha can be bypassed by sending a POST request to the user user feedback endpoint with the captcha field set to the same value as the current captcha field. Looking at how the captcha system works, each time the user enters a feedback, the captcha id is updated and the solution value is set accordingly. But, this captcha id is only updated on pressing the send button in the feedback form. So, we can send a POST request with the appropriate captcha id and captcha solution without triggering the captcha update.

Fix

The frontend asks the server for a captcha and the server sends a captcha id, the expression, for which the user has to solve the captcha. The server should update the captcha id and expression each time the user sends a feedback. This will prevent the captcha from being bypassed by sending a POST request with the same captcha id and solution.
The current system saves the captcha id and solution in the session, and after matching does not remove it from the model, but we can delete the captcha id and solution from the session after the server recieves a response for that captcha id and solution. This will prevent the captcha from being bypassed by sending a POST request with the same captcha id and solution.

```
...
// delete captcha from database to prevent re-usage
CaptchaModel.destroy({
    where: { captchaId: req.body.captchaId },
});
...
```

5. Admin Registration (Insecure Design)

This exploit can be performed by creating a user with `role` set to `admin`. This can be done by sending a POST request to the user registration endpoint with the `role` set to `admin`. While creating a regular user, the `role` field is not part of the body so the frontend completes by adding the tag `role` as `customer`. But we can bypass this by sending a POST request with the `role` set to `admin`, in which case the frontend will not add the tag `role` and the user will be created as an admin.

Fix

To fix this issue, the `role` field should be validated with additional checks on the server side. The `role` field being `admin` should require additional permissions like tokens or more password checks.

```
...
app.post(
    "/api/Users",
    (req: Request, res: Response, next: NextFunction) => {
        if (
            req.body.email !== undefined &&
            req.body.password !== undefined &&
            req.body.passwordRepeat !== undefined
        ) {
            if (
                req.body.email.length !== 0 &&
                req.body.password.length !== 0
            ) {
                req.body.email = req.body.email.trim();
                req.body.password = req.body.password.trim();
                req.body.passwordRepeat =
                    req.body.passwordRepeat.trim();
                if (req.body.password !== req.body.passwordRepeat) {
                    res.status(400).send(
                        res.__("Passwords do not match")
                    );
                }
            } else {
                res.status(400).send(
                    res.__("Invalid email/password cannot be empty")
                );
            }
        }
        next();
    }
);
app.post("/api/Users", verify.registerAdminChallenge());
app.post("/api/Users", verify.passwordRepeatChallenge()); // vuln-code-
snippet hide-end
app.post("/api/Users", verify.emptyUserRegistration());
...
```

This section needed a redesign to add the `role` field to the body and validate it properly, in the next section where the user is created. This should include an additional key of sorts to validate the `role` field if it is set to `admin`. Regular users need not have any key for creating a `customer` role user.

6. Empty User Registration (Improper Input Validation)

This exploit can be performed by creating a user with empty username and password. This can be done by sending a POST request to the user registration endpoint with empty username and password. We still get a success message even though the username and password are empty.

Fix

To fix this issue, the username and password fields should be validated before creating the user. The checks should be done on the server side to check if the username and password fields are empty, as we can always bypass the client side checks using tools like postman.

```
...
app.post(
    "/api/Users",
    (req: Request, res: Response, next: NextFunction) => {
        if (
            req.body.email !== undefined &&
            req.body.password !== undefined &&
            req.body.passwordRepeat !== undefined &&
            req.body.email.length !== 0 &&
            req.body.password.length !== 0 &&
            req.body.passwordRepeat.length !== 0
        ) {
            if (
                req.body.email.length !== 0 &&
                req.body.password.length !== 0
            ) {
                req.body.email = req.body.email.trim();
                req.body.password = req.body.password.trim();
                req.body.passwordRepeat =
                    req.body.passwordRepeat.trim();
                if (req.body.password !== req.body.passwordRepeat) {
                    res.status(400).send(
                        res.__("Passwords do not match")
                    );
                }
            } else {
                res.status(400).send(
                    res.__("Invalid email/password cannot be empty")
                );
            }
        }
        next();
    }
);
...
```

The above code will check if the username and password fields are empty before creating the user. If the fields are empty, the server will return an error message and prevent the user from being created.

7. View Basket (Broken Access Control)

This exploit can be performed by changing the URL to the basket section of the website. The basket section can be accessed by changing the URL extension to `#/basket`. This can be exploited to look into the basket of any user. To do this, we simply need to change the index at the end of the basket URL to any other user's index to view their basket.

Fix

To fix this issue, the basket section should be restricted to only authenticated users. The basket section should be restricted to only the user who owns the basket, by checking the user's session (token) before rendering the basket.

```
...
// check if the user is authenticated
// if not, return an error
if (!security.authenticatedUsers.from(req)) {
    res.status(401).send("Unauthorized");
    return;
}

// check if the token and the basket id are the same
// if not, return an error
const user = security.authenticatedUsers.from(req);
    if (user.bid != id) {
        res.status(401).send("Unauthorized");
    return;
}
...
```

The above code will check if the user is authenticated and matches the id of the user whose basket is being viewed. If the user is not authenticated or does not match the id of the user whose basket is being viewed, the server will return an error message and prevent the basket from being viewed.

8. Zero Stars (Improper Input Validation)

This exploit can be performed by copy the network request made during the user feedback and sending it manually in either console or from any other tool like postman. We simply need to change the rating value to 0, or even any higher value beyond 5 possibly to give unrealistic feedback.

Fix

To fix this issue, the rating value should be validated on the server side as well. The rating value should be checked against a whitelist of allowed values or range of values.

```
...
rating: {
    type: DataTypes.INTEGER,
    allowNull: false,
    set (rating: number) {
        // if rating is 0, set it to 1
        if (rating === 0) {
            rating = 1
        }
        this.setDataValue('rating', rating)
        challengeUtils.solveIf(challenges.zeroStarsChallenge, () => {
            return rating === 0
        })
    }
}
...
```

The above code will check if the rating value is 0 and set it to 1 if it is. This will prevent the rating value from being set to 0. This is my personal preference and can be handled in multiple ways, like returning an error message if the rating is 0.

9. Exposed Metrics (Sensitive Data Exposure)

The vulnerability is found in the `/metrics` path of the website. The `/metrics` path is exposed and can be accessed by anyone. The `/metrics` path contains sensitive information about the server and the application. This can be exploited by an attacker to get information about the server and the application.

Fix

To fix this issue, the `/metrics` path should be restricted to only authenticated users. The `/metrics` path should not be exposed to the public. The `/metrics` path is the default path for the Prometheus metrics. The `/metrics` path should be disabled or restricted to only authenticated users.
I know this is a very production/deployment specific issue, but it is still a vulnerability that can be exploited, if the server is not properly configured.

10. Outdated Allowlist (Unvalidated Redirects)

To search for any crypto related redirects (as the hint suggests), I tried to search for `crypto` in the search bar, which did not return any results. But I later searched for related words like `bitcoin` which gave me a redirect url. i replaced the current url extension with the redirect url and it redirected me to some bitcoin related website.

Fix

To fix this issue, the redirect URL should be validated before redirecting the user. The redirect URL should be checked against a whitelist of allowed URLs. Also, not to be allowed to redirect to any external URLs.
Simple fix was removing the redirect URL from the source code, as it was not required for the application.

11. Missing Encoding (Improper Input Validation)

This issue can be found in the photo wall, where the first image cannot be rendered due to missing file path. on inspecting the element, the image source is found to have `#` as part of the path, which is not encoded properly. Any path must replace the `#` character with `%23` to properly encode the path.

Fix

To fix this issue, the path should be encoded properly before rendering it to the page. The path should be encoded to replace the `#` character with `%23` to properly encode the path. Any other special characters should also be encoded properly.
This was done by simply running a sanity check on the path and replacing all the special characters with their encoded values, similar to the 1st DOM XSS fix.