ASVS Security Testing

Selected ASVS points

Type	ID	Description	Reasoning
Session Management	3.3.4	Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.	I did not check this before, and I wish to check it now.
Server-Side Attacks	5.3.4	Verify that data selection or database queries (e.g., SQL, HQL, ORM, NoSQL) use parameterized queries, ORMs, entity frameworks, or are otherwise protected from database injection attacks.	I was unsure which option to choose.
XSS	12.5.2	Verify that direct requests to uploaded files will never be executed as HTML/JavaScript content.	While this is not possible with the current application (users can upload/download any type of file, but it is not executed by the browser), I would like to think through how to make it safer when using it in a different application or webpage.

Type	ID	Description	Reasoning
REST/API	13.2.3	Verify that RESTful web services that utilize cookies are protected from Cross-Site Request Forgery via the use of at least one or more of the following: double submit cookie pattern, CSRF nonces, or Origin request header checks.	I did not check this before, and I wish to check it now.
File Management	12.1.1	Verify that the application will not accept large files that could fill up storage or cause a denial of service.	Primitive, but I was hesitant which test to choose as many were not applicable.

Tests

All test were performed with Google Chrome browser Version 128.0.6613.119 (Official Build) (64-bit) and with Mozilla Firefox 133.0 browser.

3.3.4 Session Management

Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.

Adaptation/Scope

Application does not have a feature to view currently active sessions, therefore this part of the verification will be omitted.

"Verify that users are able to log out of all currently active sessions and devices."

Verification Methodology

Verification by active testing.

Prerequisites

1. User has a valid account.

Procedure

1. Log in to the webpage in browser 1.

2. Open the webpage in another browser and log in with the same credentials in browser 2.
3. Log out from browser 1 session by sending a GET request to the `http://{server_host}:8070/logout` page.
4. In browser 2, navigate to the `http://{server_host}:8070/files` page.

Expected behaviour

The user is logged out in browser 2 and redirected to the login page.

Result(s)

The user is incorrectly logged in and authenticated, allowing them to perform all user actions.

Explanation

Terminating a session does not invalidate the current and other active session tokens, which remain valid until their MaxValue expiration time.

Threat analysis

If a malicious actor gains access to the session token, they can continue using the session even after the user has logged out, until the session token expires (MaxValue).

The malicious actor can perform the following actions:

1. View the list of all files uploaded by the user.
2. Download any file from the user's account.
3. Upload new files to the user's account.
4. Delete any file from the user's account.

Recommendations

Invalidate all session tokens associated with the user when the user logs out.

Implementation example

Store the last logout time in the database and include the login time in the session cookie. If the login time is earlier than the logout time, discard any requests associated with this session token.

5.3.4 Server-Side Attacks

Verify that data selection or database queries (e.g., SQL, HQL, ORM, NoSQL) use parameterized queries, ORMs, entity frameworks, or are otherwise protected from database injection attacks.

Adaptation/Scope

No adaptation needed.

Verification Methodology

Verification by code inspection. (Could also be verified by active testing.)

Prerequisites

1. Access to the source code.

Procedure

1. Identify all places where data selection or database queries are used.
2. Verify that the queries are using parameterized queries, ORMs, entity frameworks, or are otherwise protected from database injection attacks.

Expected behaviour

All database queries are protected from SQL injection attacks.

Result(s)

1. Database query that writes username, password hash and salt to the DB is in the signupPageHandler. This query is using Exec method which is parameterized. **This query is protected from SQL injection attacks.**
2. Database query that retrives user login data from the DB is in the logInPageHandler. This query is using QueryRow which is parameterized. **This query is protected from SQL injection attacks.**
3. Database query that retrives uploaded data in the user selected order is in the filesPageHandler. This query is partially parameterized. User ID is parameterized, but the order and sort directions are not. Order and sort directions are checked against a list of allowed values. If the value is not in the list, the default value is used. **This query is protected from SQL injection attacks.**
4. Database query that deletes a file from the DB is in the deleteFileHandler. This query is using Exec method which is parameterized. **This query is protected from SQL injection attacks.**
5. Database query that uploads a file to the DB is in the uploadHandler. This query is using QueryRow method which is parameterized. **This query is protected from SQL injection attacks.**
6. Database query that downloads a file from the DB is in the downloadFileHandler. This query is using QueryRow method which is parameterized. **This query is protected from SQL injection attacks.**

Explanation

Parameterized queries protect against SQL injection by ensuring that user input is treated strictly as data, not as part of the SQL query itself. This prevents malicious users from manipulating the SQL query structure to execute unintended commands.

Threat analysis

This application is not vulnerable to SQL injection attacks.

Recommendations

N/A

12.5.2 XSS

Verify that direct requests to uploaded files will never be executed as HTML/JavaScript content.

Adaptation/Scope

No adaptation needed.

Verification Methodology

Verification by code inspection and and by testing.

Prerequisites

1. Access to the source code.
2. Access to the webpage.

Procedure

1. Log in to the webpage.
2. Create three files with content

```
<script>
  alert('This is executed!');
</script>
```

name one file `test.html`, the other `test.txt` and the third `test`.

1. Upload all files to the webpage.
2. Download all files from the webpage.
3. Inspect if alert message was executed.

Expected behaviour

Only direct request made to the uploaded files is downloading the files. None of the files should execute the JavaScript code when downloaded.

Result(s)

None of the files executed the JavaScript code.

Explanation

Content-Disposition header is set to attachment, which forces the browser to download the file instead of displayed inline and executing it.

Threat analysis

While MIME sniffing is not a security issue with current web application, it may become a security issue when the application adds additional features like browser renders the file content by allowing to preview the file that will be uploaded.

MIME sniffing

Some browsers may perform MIME sniffing and ignore the Content-Type header if the `X-Content-Type-Options: nosniff` header is not set. If nosniff is not set, the browser may sniff the content of the files and treat files `test` and `test.txt` as HTML/JavaScript, potentially executing malicious script.

Recommendations

1. Whitelist allowed MIME types.
2. Discard files with unexpected file content.
3. Set `X-Content-Type-Options: nosniff` header.
4. Scan file content with antivirus scanner before uploading it to the server.

13.2.3 REST/API

Verify that RESTful web services that utilize cookies are protected from Cross-Site Request Forgery via the use of at least one or more of the following: double submit cookie pattern, CSRF nonces, or Origin request header checks.

Adaptation/Scope

The scope of the test is to veryfy that the DELETE request is protected from CSRF attacks.

Verification Methodology

Verification by testing.

Prerequisites

1. Access to the webpage.

Procedure

1. Log in to the webpage.
2. Upload a file to the webpage.
3. Check uploaded file ID using browser developer tools or by hovering with mouse over download button.
4. Serve file CSRF_delete.html to browser
    a. Set the correct id for DELETE request in ./docs/CSRF_delete.html line 10.
    b. Serve file CSRF_delete.html to browser: `cd /docs && python3 -m http.server 8000`
    c. Navigate to http://{server_host}:8000/CSRF_delete.html
    d. Click on the button to delete the file.

Expected behaviour

DELETE request is protected from CSRF attacks.

Result(s)

DELETE request is protected from CSRF attacks.

Explanation

When doing CORS requests, server makes preflight request to check if the request is allowed. If the request is not allowed, the server will respond with 405 Method Not Allowed.

Threat analysis

N/A

Recommendations

1. Set `Access-Control-Allow-*` headers explicitly in the code for clarity.

12.1.1 Files

Verify that the application will not accept large files that could fill up storage or cause a denial of service.

Adaptation/Scope

No adaptation needed.

Verification Methodology

Verification by testing and by code inspection.

Prerequisites

1. Access to the webpage.
2. Valid user account.

Procedure

1. Log in to the webpage.
2. Upload a file larger than 1 GB.

Expected behaviour

File larger than 1 GB is not uploaded.

Result(s)

File larger than 1 GB is not uploaded. UI diplays a message: "File size exceeds the 10 MiB limit", however upload request returned 200 OK status code.

Explanation

Source code checks if the file size is larger than 10 MiB and if it is, it does not save the file to PostgreSQL.

Threat analysis

N/A

Recommendations

1. Return `413 Payload Too Large` status code when file size exceeds the limit.