Assignment 1

Student ID: AC3837

Student name: Thanaphon Sombunkaeo

Group ID: TTV19S1

First part A3:2017 - Sensitive Data Exposure:

1. [Reflection] "What I wish to learn from this course" (1 pts)
    - SQL injecttion because it is a technique I have heard before from youtube but never try by myself on the real web app. I also heard from web app development course that we can use escaping query values in NodeJS to prevent SQL Injection.
    - Cross-Site Scripting (XSS) because I never heard of it. So I try to search for information about it, and it is interesting for me because it is similar to SQL injection in some perspectives e.g. attacker fill in some bad javascript in search input then sent link a to the victim. After the victim opens the link, the bad script is activated.
2. [Reading Report] "RWBH Chapter 1: Bug Bounty Basics" (pp. 1-9) (4 pts)

    Select and describe at least 2 things/concepts from web fundamentals you didn't know before (from Chapter 1)

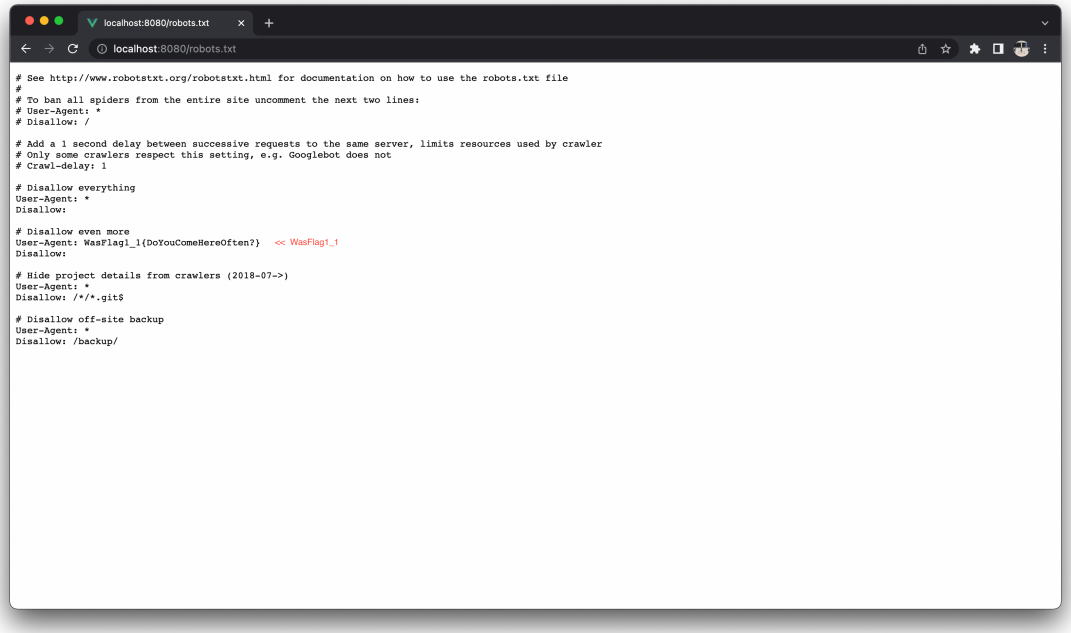
    - TRACE and OPTIONS request method: TRACE is a request method that server will send the request message back to the client. The qequester will see what information the medium has added or changed to the request message before it reachs the destination server. OPTIONS is a method that allow requester to see what avaialble method that server provide e.g. GET, POST, PUT.
    - HTTP is stateless: It means each HTTP request that sent to the server is a completely new request. Server have no idea about the previous request that comes from the same client. That's why website might use cookie to avoid resending username and password for each HTTP request for authentication.
3. [Issue Report] "Hello Wasdat!" (4 pts)

    Title: Sensitive data is exposed in `/backup` which comes from `/robots.txt`

    Description: Annonymous user can access `robots.txt` file to see the route that developer disallow the search engine crawlers to access. Then they see `/backup` that might contain sensitive data.

    Step to produce:

    - WasFlag1_1: You can find WasFlag1_1 in `localhost:8080/robots.txt`



- WasFlag1_2
  - Access `localhost:8080/backup` then download file `2021-01-22-backup.tar.gz`
  - Unzip the file and open `backup.sh` WasFlag1_2 is in the 14th line

Mitigation:

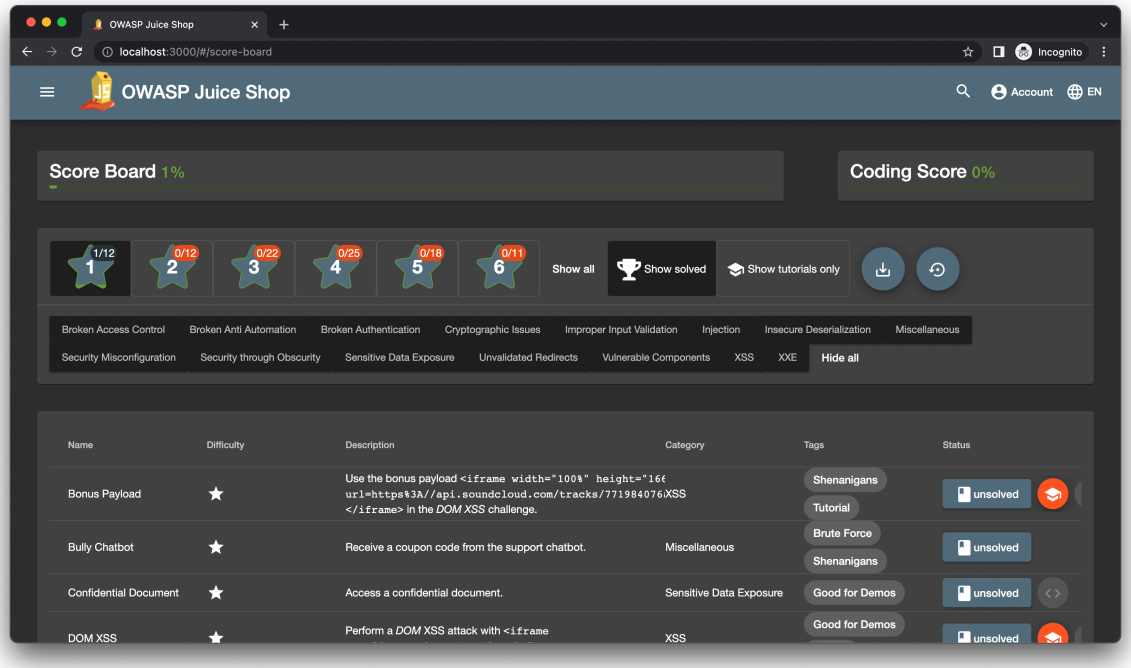
- First option: remove Disallow: /backup/ from `robots.txt` So, it can prevent anonymous user to see the route that we try to protect from crawlers.
- Second option: config the nginx to prevent every user from accessing `/backup`

4. [Reflection] "Finding the score board from Juice Shop" (1 pts)

> localhost:3000/#/score-board

Since this project use angular as a frontend framework, we can look in the route folder to see all available routes.

- `git clone https://github.com/juice-shop/juice-shop`
- `cd juice-shop/frontend`
- `find . -name "score*"` to see all directory starts with score and I get `./src/app/score-board` that means score board page is located at `/#/score-board`



Second part A1:2017 - SQL Injection:

1. [Reading Report] "RWBH Chapter 9: SQL Injection" (pp. 81-93) (4 pts)
    - Select and describe at least 2 factors that would make finding and exploiting an SQL injection vulnerability easy and fun.
        - SQLi attack are highly reward: If attacker can perform SQLi, they can do lots of things. For example, login to any user that they have username or email, can create new admin account, can drop the whole database table.
        - SQLi is easy to understand and perform: If the backend did not use prepare statement to escape the parameter value, it's very easy for attacker to perform SQLi. SQLi is easy to understand because the basic concept of SQLi is just to add string that will break the original SQL command and we can add any command we want e.g. `'--`
    - What is "Blind SQLi"?
        - Blind SQLi is a type of SQLi that we can inject the SQL statement by passing our own value to parameter but we did not get the result that we expected directly on the website. On the other hand, we can use that output to infer some information.

          Example from YAHOO SPORTS BLIND SQLI that they try to pass parameter year to be `(2010)and(if(mid(version(),1,1))='5',true,false))--` to the SQL command that might look like `SELECT * FROM PLAYERS WHERE year = ? AND type = ? AND round = ?;` So the command will be `SELECT * FROM PLAYERS WHERE year = (2010)and(if(mid(version(),1,1))='5',true,false))-- AND type = ? AND round = ?;`. That means, if they can see the player in year 2010, the MySQL version of website is 5 otherwise not. Which means we can infer some information from output and that's what blind SQLi is.
2. [Issue Report] TARGET: Juice Shop => "Login Jim" (2 pts)

   Title: Anonymous user can login to any user that they have an email.

Description: The login function of backend execute the SQL statement from plain SQL string which can lead to SQL injection. So if anonymous user have an email of other users, they can login without knowing the victim's password.

Step to produce: to login as Jim

- Find email of user Jim: jim@juice-sh.op
- Login with email: jim@juice-sh.op'-- ('-- will end the SQL statement that check the password)
- Can use any password to make the password validation successful.

Mitigation:

- Use prepare statement instead of plain string (see: https://github.com/sidorares/node-mysql2#using-prepared-statements)

3. [Issue Report] TARGET: Wasdat? => "Wasdat login SQLi" (4 pts)

   Title: Anonymous user can perform SQL injection to login to any user that they have an email.

   Description: SQL statement of backend API POST `/api/users/login` does not use prepare statement or other methods to escape the password value which can prevent SQL injection.

   Step to produce: to login as wasdat-victim@example.com without providing password.

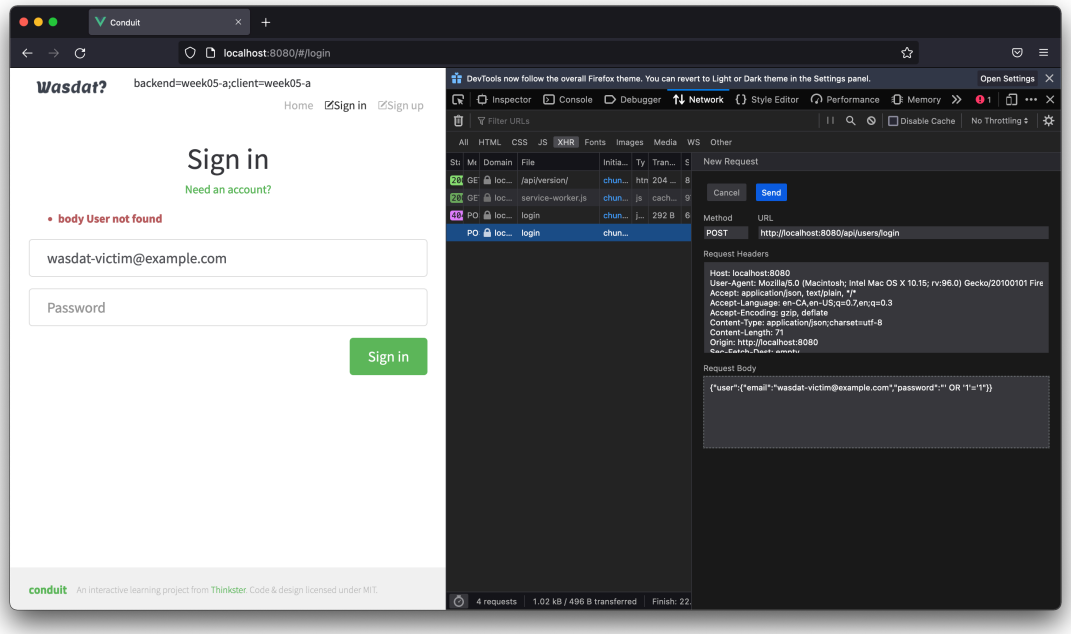
   Since frontend performs password encryption before sending the login request, we need to send the request by ourselves to perform SQL injection. Otherwise it's hard to do SQL injection.

   Find SQL injection vulnerability in Wasdat's login (2 pts)

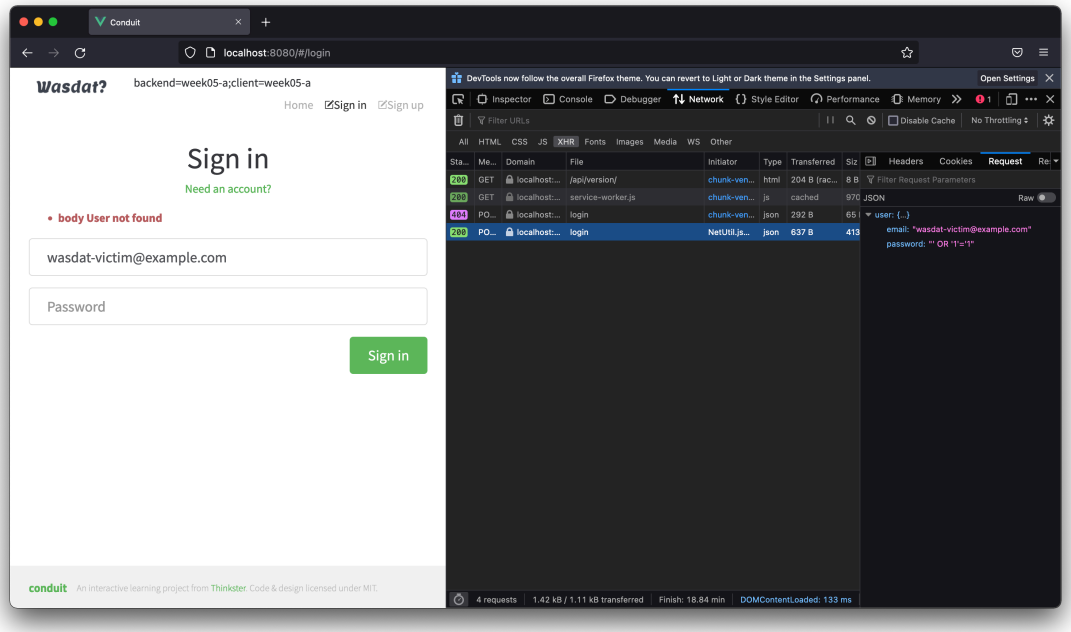
   - Use firefox as a web browser to login. Email is wasdat-victim@example.com (can let password field empty)
   - Open DevTools. Edit and resend `/api/users/login` by editing password to be `' OR '1'='1` to perform SQL injection. The SQL statement will look like

     ```
     SELECT * from user WHERE email =${email} AND password='' OR '1'='1''
     ```

     So this statement will always be true from `OR '1'='1'`

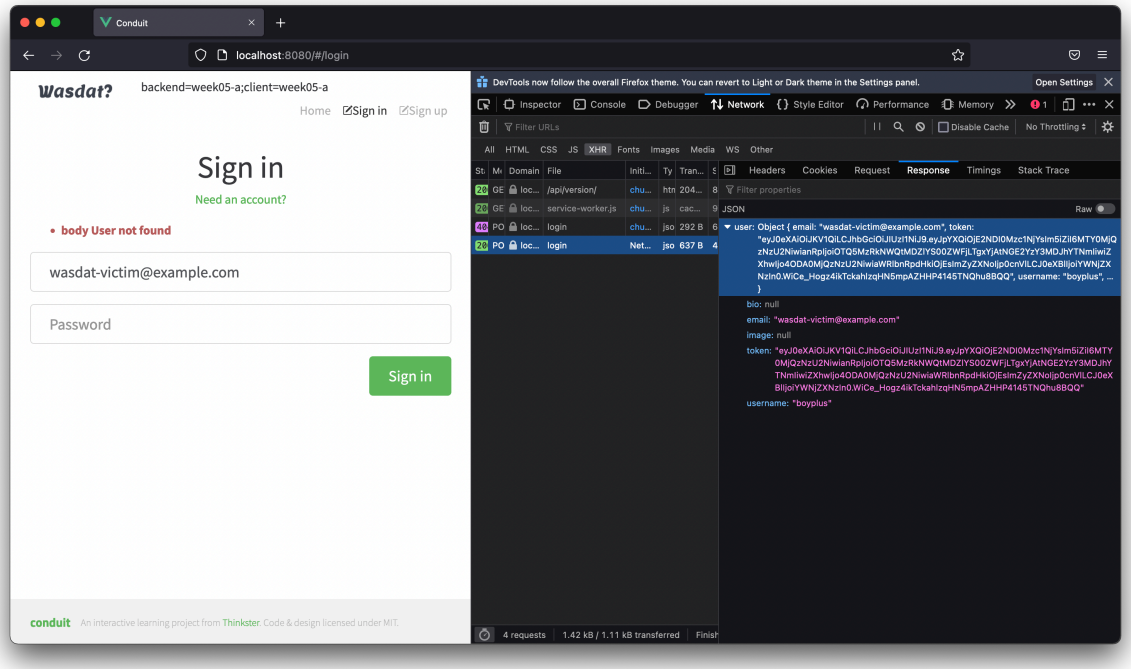


Now the login is successful.



Get JWT authentication token for wasdat-victim@example.com by exploiting SQL injection vulnerability (2 pts)

Since we can login to wasdat-victim@example.com, we can see the JWT from response of POST `/api/users/login` which show by below image



Mitigation:

- Use prepare statement instead of plain SQL statement string to prevent SQL injection
  - See: https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursorprepared.html for MySQL of python