

# 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 injection 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 requester will see what information the medium has added or changed to the request message before it reaches the destination server. OPTIONS is a method that allows requester to see what available methods that server provides e.g. GET, POST, PUT.
- HTTP is stateless: It means each HTTP request sent to the server is a completely new request. Server has no idea about the previous request that comes from the same client. That's why websites might use cookies 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:** Anonymous user can access `robots.txt` file to see the routes that developers 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`

```

# See http://www.robotstxt.org/robotstxt.html for documentation on how to use the robots.txt file
#
# To ban all spiders from the entire site uncomment the next two lines:
# User-Agent: *
# Disallow: /
#
# Add a 1 second delay between successive requests to the same server, limits resources used by crawler
# Only some crawlers respect this setting, e.g. Googlebot does not
# Crawl-delay: 1
#
# Disallow everything
User-Agent: *
Disallow: /
#
# Disallow even more
User-Agent: WasFlag1_1{DoYouComeHereOften?} << WasFlag1_1
Disallow:
#
# Hide project details from crawlers (2018-07-->
User-Agent: *
Disallow: /*/.git*
#
# Disallow off-site backup
User-Agent: *
Disallow: /backup/

```

- 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`

The screenshot shows the OWASP Juice Shop Score Board page. At the top, there are two progress bars: 'Score Board 1%' and 'Coding Score 0%'. Below them, six challenges are listed with their completion counts: 1/12, 0/12, 0/22, 0/25, 0/18, and 0/11. A 'Show all' button is available. Below the challenges is a navigation bar with categories: Broken Access Control, Broken Anti Automation, Broken Authentication, Cryptographic Issues, Improper Input Validation, Injection, Insecure Deserialization, Miscellaneous, Security Misconfiguration, Security through Obscurity, Sensitive Data Exposure, Unvalidated Redirects, Vulnerable Components, XSS, XXE, and a 'Hide all' button. The main table lists four challenges:

Name	Difficulty	Description	Category	Tags	Status
Bonus Payload	★	Use the bonus payload <iframe width="100%" height="16px" src="https://api.soundcloud.com/tracks/771984076XSS</iframe> in the DOM XSS challenge.		Shenanigans, Tutorial	<span>unsolved</span> <span>Tutorial</span>
Bully Chatbot	★	Receive a coupon code from the support chatbot.	Miscellaneous	Brute Force, Shenanigans	<span>unsolved</span> <span>Brute Force</span>
Confidential Document	★	Access a confidential document.	Sensitive Data Exposure	Good for Demos	<span>unsolved</span> <span>Good for Demos</span>
DOM XSS	★	Perform a DOM XSS attack with <iframe>	XSS	Good for Demos	<span>unsolved</span> <span>Good for Demos</span>

## 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](mailto: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](mailto: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'`

The screenshot shows a Firefox browser window with the Conduit theme. The URL is `localhost:8080/#/login`. The page displays a 'Sign in' form with fields for 'Email' (containing `wasdat-victim@example.com`) and 'Password'. A green 'Sign in' button is visible. The Network tab in the DevTools panel shows a POST request to `/api/users/login` with the following body:

```
{"user": {"email": "wasdat-victim@example.com", "password": "" OR '1'='1"}}
```

The response status is 200 OK, and the response body contains a JSON object with a key 'token' and value 'eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ3YXNkYXR0b2YtbmljbWVtZWQubWFwLmNvbSIsImV4cCI6MTYxMjEwOTUyMiwidXNlcm5hbWUiOiJ3YXNkYXR0b2YtbmljbWVtZWQubWFwLmNvbSJ9'. The status bar at the bottom indicates 4 requests, 1.02 kB transferred, and a finish time of 22ms.

Now the login is successful.

The screenshot shows a Firefox browser window with the Conduit theme. The URL is `localhost:8080/#/login`. The page displays a 'Sign in' form with fields for 'Email' (containing `wasdat-victim@example.com`) and 'Password'. A green 'Sign in' button is visible. The Network tab in the DevTools panel shows a POST request to `/api/users/login` with the following body:

```
{"user": {"email": "wasdat-victim@example.com", "password": "" OR '1'='1"}}
```

The response status is 200 OK, and the response body contains a JSON object with a key 'token' and value 'eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ3YXNkYXR0b2YtbmljbWVtZWQubWFwLmNvbSIsImV4cCI6MTYxMjEwOTUyMiwidXNlcm5hbWUiOiJ3YXNkYXR0b2YtbmljbWVtZWQubWFwLmNvbSJ9'. The status bar at the bottom indicates 4 requests, 1.42 kB transferred, and a finish time of 18.84 min.

## 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

The screenshot shows a browser window with a 'Conduit' tab open at `localhost:8080/#/login`. The main content is a 'Sign in' form for 'wasdat?'. The form has two input fields: 'Email' containing `wasdat-victim@example.com` and 'Password' containing `password`. A green 'Sign in' button is below the fields. Above the form, there's a message: 'Need an account? [Create one](#)' and a link to 'Logout'. Below the form, there's a footer note: 'conduit An interactive learning project from Thinkster. Code & design licensed under MIT.'

The right side of the screenshot shows the Firefox DevTools Network tab. It lists several requests and responses. One POST request to `/login` is highlighted, showing the JSON payload:

```

{
  "email": "wasdat-victim@example.com",
  "password": "password",
  "username": "boyplus"
}

```

The response for this request is a user object:

```

{
  "user": {
    "id": 1,
    "email": "wasdat-victim@example.com",
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eypjXQOjE2NDl0Mzc1NjYsIm5iZlBMTY0MjZnZU2Nwlanlp0iQTOGmzRkNQM2DzYS002WF1LgxjAANGE2y23MD.jyTnmiwXhej04ODAMjQ2nZU2NwlaWRlbnpdtk0jElmzjZKXNslp0cnVLCj0ExBiljWYnjZXNzn0WICe_Hogz4iR7canhzqIN5mpAzhf4i46TNQhu8DQQ",
    "username": "boyplus"
  }
}

```

## 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-mysql-cursorprepared.html> for MySQL of python