ST2515

BENEDICT ISAAC LIM (1928880), METTA TAN HOUYIK ()

dism/ft/2b/02

Secure Coding

Assignment 2 Report

Table of Content

**Type chapter title (level 1)1**

Type chapter title (level 2)2

Type chapter title (level 3)3

**Type chapter title (level 1)4**

Type chapter title (level 2)5

Type chapter title (level 3)6

# Executive Summary

In this report, we will be going through the various web security vulnerabilities present in Snapsell and classifying them under six main respective categories, SQL Injection, Broken Authentication, Sensitive Data Exposure, Broken Access Control and Cross-site Scripting. We conducted a risk assessment on each of the categories, providing background information and determining the likelihood of these vulnerabilities being present on modern web applications.

For each of the six categories, we identified two specific examples of these vulnerabilities in Snapsell, going in-depth on one of them while giving a brief overview of the other. Under SQL Injection, we showcased a vulnerability in the search field that allowed an attacker to view the login details of every user. Under broken authentication, we covered a vulnerability that allows an attacker to test a large amount of email and password combinations to find a successful one. For sensitive data exposure, we showed that the login details were stored in the database unencrypted, and how attackers could view them over the network using packet sniffers. Under broken access control, we covered several vulnerabilities that allowed an attacker to act outside of their user permissions and steal other users’ login tokens. Lastly, for cross-site scripting, we covered a stored XSS vulnerability in one of the listing APIs and a reflected XSS vulnerability in the search bar.

For both examples, we explained the methods and tools to exploit the vulnerability, displayed the snippet of code that caused the vulnerability, and lastly provided a recommendation to patch the vulnerability.

# SQL Injection

## Risk Assessment

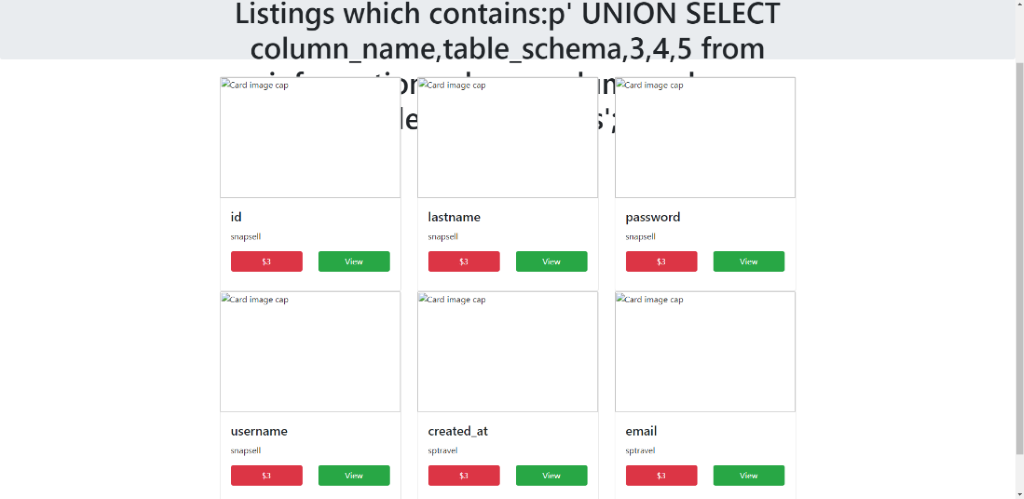
|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| High | Medium |
| Description: SQL Injection is a common attack vector that involves an attacker making malicious queries to a backend database. It exploits the lack of input validation and escaping of special characters to enable the attacker to modify SQL queries.  SQL Injection allows an attacker to retrieve information that was not intended to be displayed. This information includes confidential company data, private user data or any other data that should not be shown to the user. This results in the attacker gaining unauthorized access to sensitive data such as user details, passwords and credit card information.  Given that the Snapsell web application contains flaws that leave it vulnerable to SQL injection, a malicious user is able to view all the content in the tables stored in the MySQL database. This means that an attacker is able to view confidential information such as login details and offers made by other users. | SQL injection attacks occur fairly frequently and are simple to execute. Hence, injection attacks have earned themselves a position on OWASP’s Top 10 Web Application Security Risks.  A reason why Injection vulnerabilities are so common is the continued use of legacy web applications. As the codes that comprise these web applications are often outdated, it is extremely likely that attackers can exploit any vulnerabilities.  Additionally, an attacker’s job is made even more simple with the aid of scanners and automated fuzzing software.  As such, it is no surprise that Snapsell also contains vulnerabilities pertaining to SQL injection. |

## Detailed Example

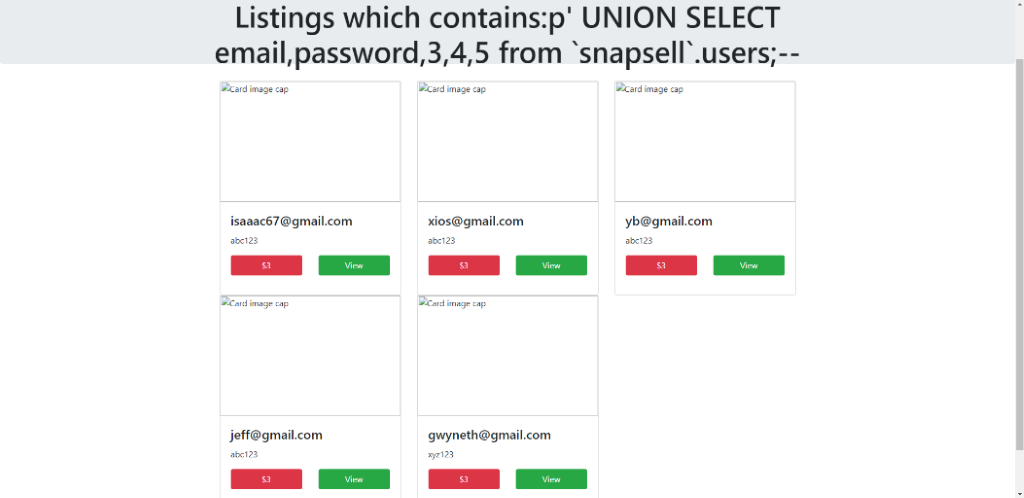
### Execution:

Entering **P' UNION SELECT column\_name, table\_schema,3,4,5 from information\_schema.columns where table\_name= 'users';-- -** into the search field will display all the column names from the ‘users’ table.





From this, we can enter **p' UNION SELECT email,password,3,4,5 from `snapsell`.users;-- -** to select the ‘email’ and ‘password’ columns to view the login details of every user.

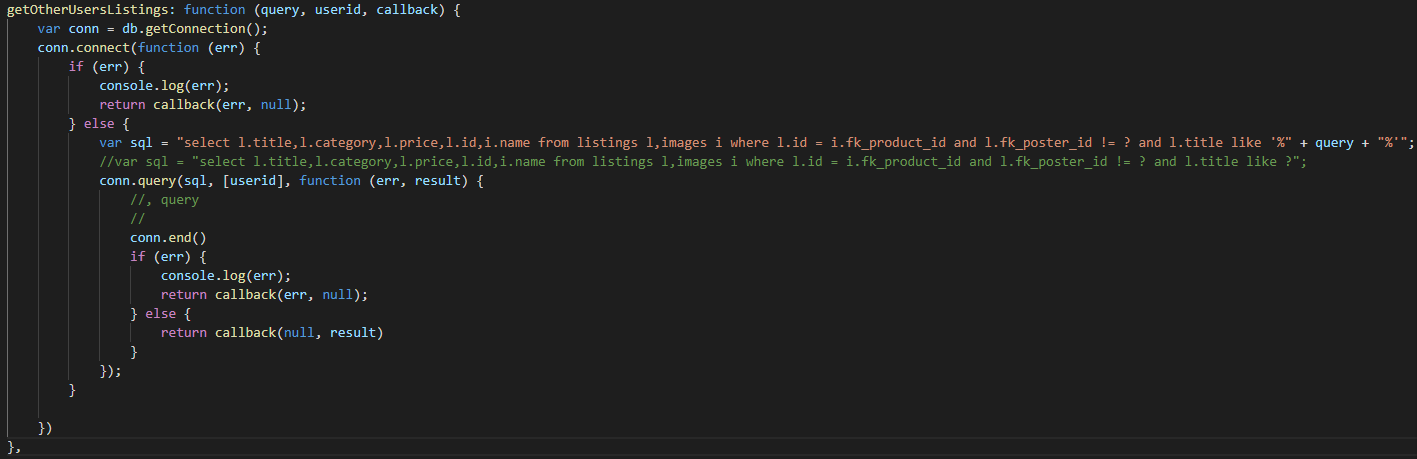


This will allow the attacker to login to Snapsell using the details of any user in the database.

### Reason:

There exists an SQL injection vulnerability in the ‘listing.js’ file.

**Code snippet: listing.js**

The line of code

var sql = "select l.title,l.category,l.price,l.id,i.name from listings l,images i where l.id = i.fk\_product\_id and l.fk\_poster\_id != ? and l.title like '%" + query + "%'";

makes the website vulnerable to SQL injection as it enables the attacker to enter malicious code into the field to be run. This will in turn allow the attacker to view the login details of every user.

## Recommendation

Parameterized queries are pre-compiled SQL statements where user input is automatically quoted and will not change the intent of the query. When a parameterized query is sent, the database knows the intended function of then query and will insert the parameters as values. Hence, the user input will be unable to affect the query, helping to mitigate SQL injection attacks.

Below is the updated code with parametrized queries in place to prevent SQL injection attacks.

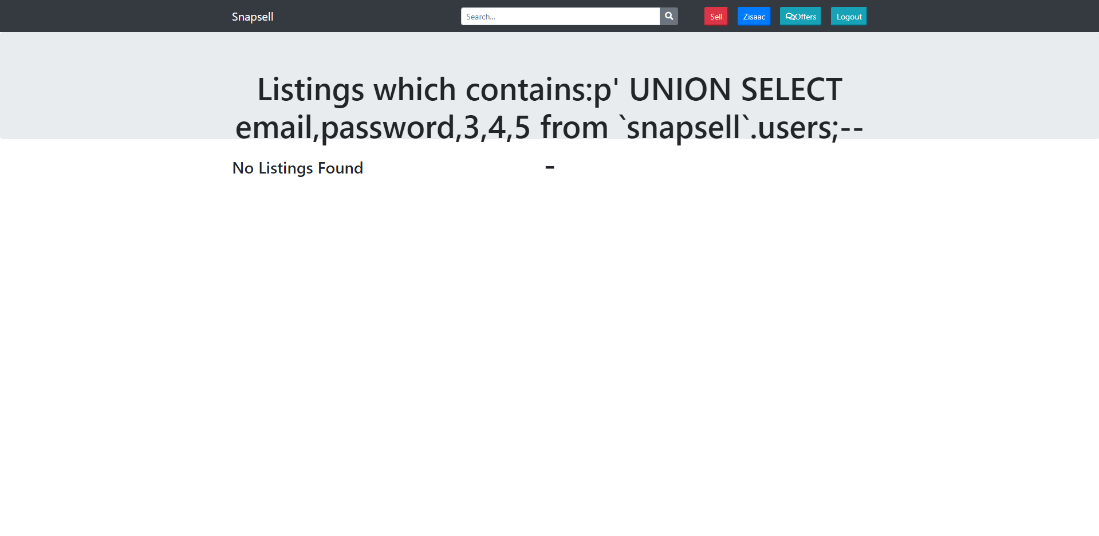
### Implementation:

**Code snippet: listing.js**

### 

### Execution:

Now, if the attacker were to enter “**p' UNION SELECT email,password,3,4,5 from `snapsell`.users;-- -**” into the search field, the database would look up the listing “**p' UNION SELECT email,password,3,4,5 from `snapsell`.users;-- -**” which would not produce any results.



## Brief Example

There exists an SQL injection vulnerability in the ‘offer.js’ file.

The ‘addOffer’ function in offer.js is vulnerable to SQL injection. Similar to that of the detailed example, the SQL statement accepts the user input directly, meaning it can be manipulated.

**Code snippet: offer.js**

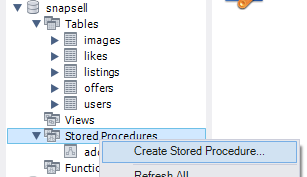
var sql = `insert into offers(offer,fk\_listing\_id,fk\_offeror\_id,status) values('${offer}', '${fk\_listing\_id}', '${fk\_offeror\_id}', '${status}')`;

## Recommendation

Stored procedures can be created and used in place of regualr SQL queries. They are special functions/methods that help to restrict access to specific sensitive columns in the database.

### Implementation:

To create a stored procedure, right-click on Stored Procedures and select ‘Create Stored Procedure’.



Below are the contents of the stored procedure we will be using.



To utilize the stored procedure, we will need to call it in the addOffer function in place of the SQL query.

**Code snippet: offer.js**

var sql=`call addOffer(?,?,?,?)`;

//var sql = `insert into offers(offer,fk\_listing\_id,fk\_offeror\_id,status) values('${offer}', '${fk\_listing\_id}', '${fk\_offeror\_id}', '${status}')`;

# Broken Authentication

## Risk Assessment

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| Very High | High |
| Broken authentication attacks occur when malicious individuals exploit flaws and weakness in a service’s session and credential management. This results from improper confirmation and management of a users’ credentials in order to prove their identities. These attackers use several techniques such as credential stuffing, brute force and dictionary attacks to exploit broken authentication in an application.  These attacks are carried out with the goal of impersonating and gaining the privileges of genuine users of services online. With such privileges, attackers are able to compromise the confidentiality, integrity and availability of data on targeted services as they will have gained access to private resources such as user login credentials, session details and even administrative controls.  Broken authentication attacks pose a high risk as an attacker can easily and quickly compromise and entire system simply by obtaining access to a small number of accounts (or just one account with administrator privileges). This allows for the possibility of large-scale negative impacts such as identity theft, unauthorised disclosure of confidential information, and social security fraud among others.  In the case of the Snapsell web application, broken authentication vulnerabilities potentially allow unauthorised users to view and alter the profiles of other accounts which contain their respective first and last names, emails, and usernames. | Broken authentication is highly common as a result of the intrinsic design and application of average access and identity controls such as Session management.  Given that session management is at the root of access controls and authentication, the detection of broken authentication vulnerabilities is possible through manual and automated means such as dictionary attacks.  The likelihood of broken authentication is further worsened by the extreme prevalence of session management in stateful applications and services. Since the use of session management has been standardised, many applications and services are possibly left vulnerable.  Moreover, broken authentication vulnerabilities can be exploited by automated attacks such as credential stuffing and brute force, making them easy for attackers to carry out. |
| Detailed Example:Setup: One example of a broken authentication attack is a dictionary attack. A dictionary attack is a form of brute force attacks, which involves attackers attempting to guess different pairings of usernames and passwords through a process of trial and error in order to gain access to other accounts on an application/service.  For this detailed example, Burp Suite Community Edition will be used as a tool to conduct a dictionary attack on the Snapsell web application.  By using Burp Suite as a proxy between a client and the server, requests made to the backend can be intercepted.    The screenshot above depicts a POST request from the client to Snapsell’s backend server that has been intercepted by Burp Suite. It can be seen that an email of ‘***isaaac67@gmail.com’*** and a password of ‘***password***’ have been sent in this request.    By sending the request to Burp Suite’s Intruder, attackers can position the payloads at the email and password portions of the request (highlighted in green).    Next, attackers can use text lists/dictionaries for both email and password payloads (left and right respectively). These lists and dictionaries can be found online or made by attackers themselves.  The image on the left shows a list of emails gathered from the Snapsell MySQL database, which an attacker may be able to obtain through other attacks such as SQL injection (even if the passwords are hashed).  The Image on the right shows a dictionary list of passwords taken from the leaked records of a company named RockYou. It contains a long list of passwords which can be used for brute force attacks such as this example. Execution Given that the community edition of Burp Suite throttles the speed of attacks, a smaller list of 10 passwords will be used purely for demonstration purposes instead of the RockYou dictionary (real scenarios involve larger lists).    After the execution of the attack is completed, attackers can view all requests with successful status codes (e.g. 201) as opposed to errors. This serves as an indication that the email and password combinations are valid. These combinations can then be used as login credentials to gain access to accounts not belonging to attackers. Reason: This vulnerability exists because the Snapsell web application does not limit the number of login requests that can be made by a user. This means that an attacker can freely make requests with no restrictions or cooldowns. Thus, this leaves the application susceptible to automated attacks.  **app.js**    **user.js**    As can be seen in the images from ***app.js*** and ***user.js***, there are no measures put into place to limit the number of login requests that can be made in a given time. | |
| Recommendations In order to patch this vulnerability, a restriction must be placed on the frequency of requests from a given client. Fortunately, this is possible with the assistance of the node package ***express-brute***. Installation: For the purposes of the Snapsell web application, express-brute requires three dependencies. They are ***sequelize***, ***express-brute-sequelize*** and ***mysql2.***  *npm install express-brute –save*  *npm install express-brute-sequelize –save*  *npm install mysql2 –save* Implementation: **app.js**    After declaring all the necessary variables for the node packages (ExpressBrute, SequelizeStore and Sequelize), another variable which stores a **Sequelize** object should be declared. This object contains the details of the MySQL database such as the schema name (snapsell), username (Ben), password (!QWER4321) and options (host IP address, dialect and logging).    Additionally, a function should be declared to handle any errors relating to the ***express-brute*** store.    Lastly, a **SequelizeStore** object should be declared. Within this new object, the **sequelize** variable declared earlier is passed in, along with the name of the MySQL table that will be used to store the frequency of failed logins (bruteStore) and a callback function.  This function declares a variable named **bruteforce** which contains a new **ExpressBrute** object. The **store** is passed in conjunction with options such as *freeRetries, minWait, maxWait, failCallback and handleStoreError* (see comments for descriptions).  The function also contains the API endpoint for logins. This remains mostly unedited from the original code except for presence of the the additional **bruteforce.prevent** middleware and the **req.brute.reset()** method which will reset the failed login counter to zero after a success. Execution:   On initial failed login attempt, an alert will indicate that the user has entered the wrong credentials.    After 5 failed login attempts, the alert will warn the user the he/she has had too many failed attempts and must try again after a specified amount of time (minimum of 30 seconds and maximum of 30 minutes). If a user tries to log in during this cooldown period, they will be restricted from doing so until it is over. This remains true even if the email/password entered is valid.    All the failed attempts that are made are store in a table named **brutestores** in the snapsell schema that was created by the **Sequelize** node package. For each failed attempt made, each user row (with a different ID) has its count incremented.    Once a user has successfully logged in, the row is then deleted from the database and the user can now use the web application. | |
| Brief Example Another vulnerability that pertains to broken authentication that affects Snapsell is its lack of strong password enforcement.  **snapsell.users**    As seen in the above screenshot of the **users** table in the **snapsell** schema, all current passwords are only six characters in length and extremely simple in nature. This is dangerous as attackers have a high chance of guessing passwords to user accounts.  Moreover, weak passwords increase the success rate of brute force attacks. This is because dictionaries that are used for such attacks contain commonly used words and phrases used in weak passwords. Additionally, having shorter passwords reduces the number of possible character combinations and permutations used in said passwords.  **app.js**    **user.js**    Shown in the snippets above, neither the login endpoint in **app.js** or the ‘loginUser’ function in **user.js** contain any code to enforce password length or complexity as values are immediately taken from a request’s body and inserted into the **users** table. | |
| Recommendations In order to enforce the use of strong passwords, we recommend to employ the use of the ***validator*** node package. Installation: npm install validator –save Implementation: **app.js**    By declaring the variable with ***var validator=require('validator'),*** we can use the ‘validator.isStrongPassword()’ function to enforce password to at least be 8 characters in length and have at least 1 lowercase, uppercase, number and symbol each.  If the entered password does not meet this criteria, the client will be returned a status code of 400 (bad request) and the user account will not be created. Execution: **POSTMAN** | |

# Sensitive Data Exposure

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| High | Very High |
| Sensitive data exposure takes place when confidential information is unintentionally revealed. This occurs when applications and APIs do not employ sufficient security measures to protect private information.  Malicious individuals can exploit flaws in an application/service’s software as well as oversights in encryption of said confidential information. Some a key example is man-in-the-middle attacks, where attackers intercept the data transmitted between clients and servers.  This is significant as sensitive data exposure can lead to many negative impacts and inconveniences. Attackers can use the private data that they uncovered for their personal gain. Notable examples include access to identity theft, selling of data to others and credit card fraud.  With reference to the Snapsell application, it contains certain vulnerabilities that allow for the possibility of sensitive data exposure. Two of which will be demonstrated below. | Stated in the OWASP Top Ten for 2017, attacks carried out with the intent of sensitive data exposure are the most frequent among the list. This is most often due a lack of encryption for confidential data that is stored and use of weak hashing algorithms.  Moreover, while security flaws for transmitted data can be easily detected, the same cannot be said for stored data. This increases the difficulty of securing any private information and preventing sensitive data exposure. |
| Detailed Example As mentioned earlier in this report, the Snapsell web application is vulnerable to SQL injection.    By entering an SQL query such as “*x' UNION SELECT username, email, password,4,5 from users; -- -*“ into the search bar, the details of all users – usernames, emails and passwords – are shown clearly instead of the intended listings.  **app.js**    In app.js, it can be seen that the details of newly created users (username, email, password, etc.) are taken from a request’s body and passed into the function ‘user.addUser()’ from user.js without any processing.  **user.js**    These values are then inserted into the MySQL database through an SQL query.  **MySQL Workbench**    As can be seen in the screenshot above, the data of newly created users is unobscured and stored in the MySQL database in plaintext. This is unsafe as an attacker would easily be able to view and understand this information if he/she were to ever gain access to it (as shown earlier through SQL injection). | |
| Recommendations: As the issue stems from the easily viewability and understandability of the user information (passwords in particular), implementing some form of data obfuscation will help ensure that malicious individuals will not be able to comprehend the passwords of users even if they manage to view them.  One such process to obscure data is hashing. Hashing is the process of mapping data of any size to fixed-sized values. This is useful for securing data as the hash values that are generated from hash function are irreversible. This means that attackers will be unable to gain any useful information from the hash values if they manage to obtain them. Thus, in order to protect the information of users of the Snapsell application, all password values should be hashed.    Using an API client such as Postman, a request can be made to the ‘/user’ endpoint to create a new user account and store its data in the MySQL database.  In order to hash the password values sent to the database, we recommend that the bcryptjs library is used as it provides a strong hashing function.  **app.js**    Open a BackEnd terminal and run ‘npm install bcryptjs --save’. Next, declare a constant for the bcrypt module.    After all values from the request body are obtained, a salt is generated with 12 rounds using the bcrypt.GenSalt() function. This salt is then used in conjunction with the created account’s password to generate a hash value using the bcrypt.hash() function.  Next, the hash value is passed into the user.addUser() from the user.js in the backend model.  **user.js**    Lastly, a connection is made to the MySQL database and a query is made which inserts a new row in the ‘users’ table which includes the newly generated hash value in place of plaintext passwords (as seen below).  **Postman**    **MySQL Workbench** | |
| Tool & Methods:  * Postman * bcryptjs | |
| Detailed ExampleExecution: Packet sniffers like Wireshark can be used to capture traffic going through a network. For example, if the attacker were to be on the same network as one of the Snapsell users, he can use Wireshark to view the login information and token of the user in plaintext.     Recommendation Currently, the server is using HTTP which is not secure. This is what allows packet sniffing to take place. Hence, we will be switching to the HTTPS protocol, which is the secure version of HTTP. HTTPS is encrypted to ensure security during data transfer between the browser and server. To use HTTPS, we will need an SSL certificate, which enables HTTPS and makes SSL/TLS encryption possible. The certificate contains the website’s public key and other related information, such as domain name, certificate authority, issue and expiration date. Any devices trying to connect to the server will need to obtain the public key for the certificate and verify the identity of the server. Implementation: To be able to serve Snapsell on HTTPS from localhost, we will need to create a self-signed certificate. Keep in mind that the browser will complain that the certificate is not trusted, as it is self-signed. Make sure OpenSSL is installed on your system. This can be checked by typing ‘openssl’ in a cmd terminal.  C:\Users\metta>openssl  'openssl' is not recognized as an internal or external command,  operable program or batch file.  From this, we can see that OpenSSL is not installed. There are various ways to install OpenSSL. More information can be found on the Internet. For this example, we will be using Git, an open-source distributed version control system.  Next, navigate to C:\Program Files\Git and run ‘git-bash.exe’. This will launch a terminal which we will use to set up OpenSSL.  First, we will change directory to the folder we want the certificate to be created in.  cd D:\SC\SCAY2021S1\_CA1\SCAssignment1\ssl  Next, enter openssl req -nodes -new -x509 -keyout server.key -out server.cert  You will be prompted to answer several questions. Answer accordingly, making sure to set Common Name as ‘localhost’ and to add your email address.    You should notice that 2 files have been created in the folder, the certificate and the private key.    Now, we will need to modify the server.js code to support HTTPS usage.  **Back-end Server:**  **Front-end Server:**  Next, we must edit the code in all of the files, replacing all instances of ‘http’ with ‘https’. Execution: To start Snapsell, we need to start up the back-end server first and click on ‘Proceed to localhost’.    Now, we can launch the front-end server and everything should function as normal. After the changes have been made, if we were to do a Wireshark capture as a user logs in, the data should be encrypted as it travels from browser to server. | |

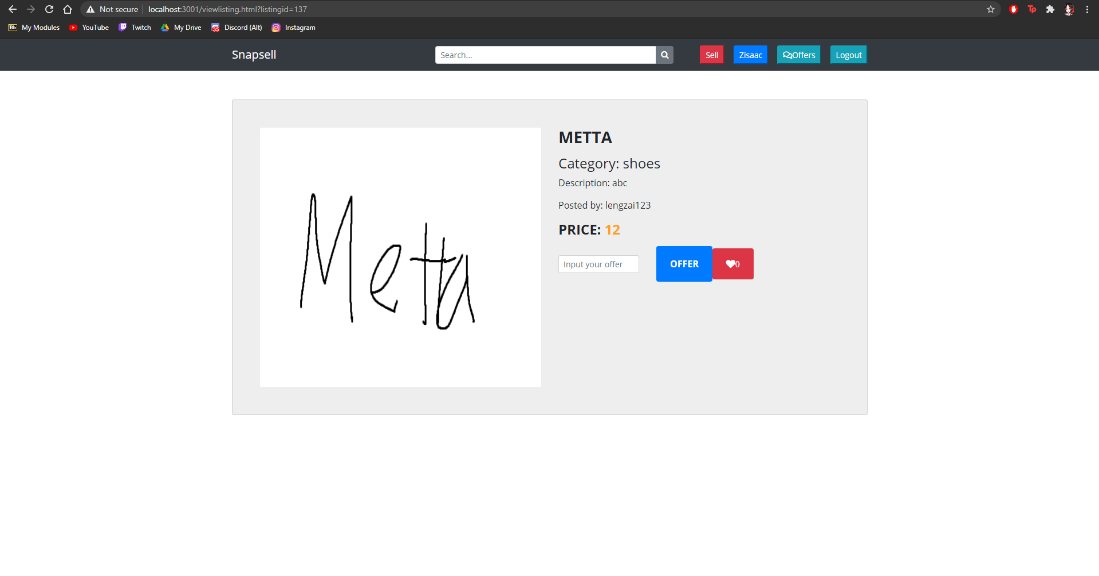
# Broken Access Control

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| High | High |
| Broken access control vulnerabilities allow users to act outside of their intended permissions through privilege escalation. These vulnerabilities usually stem from insecure coding or poor implementation of authorization features.  Attackers can access sensitive information from other users, manipulate data by performing functions of other roles, including those with greater permissions, and can even compromise the entire web server in some circumstances.  In the case of the Snapsell web application, broken access control vulnerabilities could potentially allow attackers to access or modify information or listings of other users. | Broken access control vulnerabilities are likely to be present in web applications. This is because they are difficult to detect in advance. Moreover, it is common for web developers to underestimate the difficulty of implementing secure access control mechanisms. Web applications need to have access control checks on not just the client side, but when the server is requesting any function as well.  These factors together contribute to Broken Access Control receiving a 5th place ranking on OWASP Top 10 Web Application Security Risks. |

## Detailed Example

### Execution:

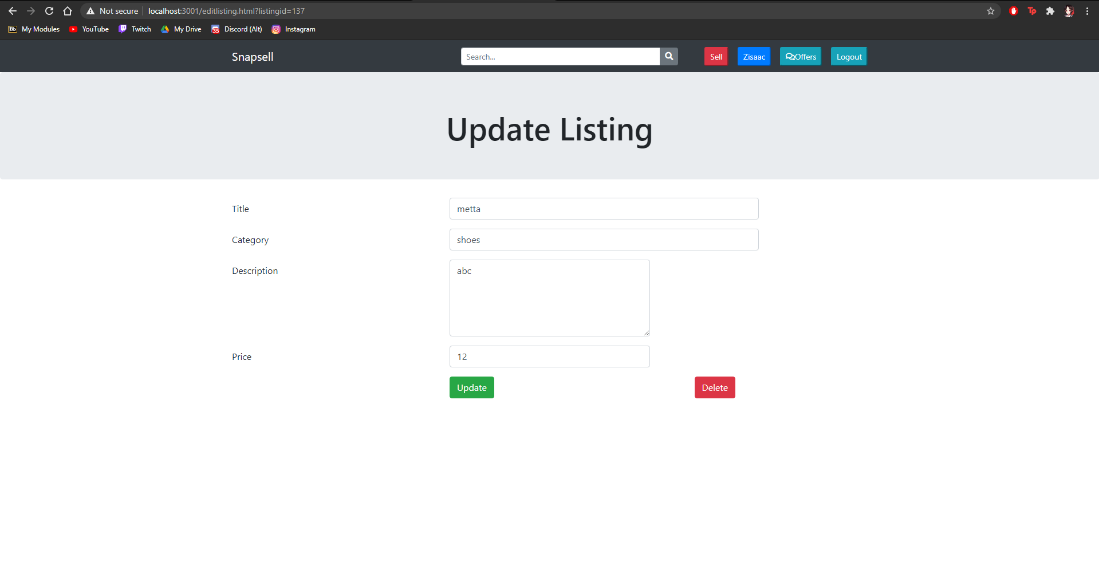
As mentioned above, Snapsell contains broken access control vulnerabilities. The web application has a significant flaw that allows users to edit or even delete the listings of other users. For example, let’s say Zisaac searched for the listing ‘metta’ by lengzaI123 and pressed ‘View’.



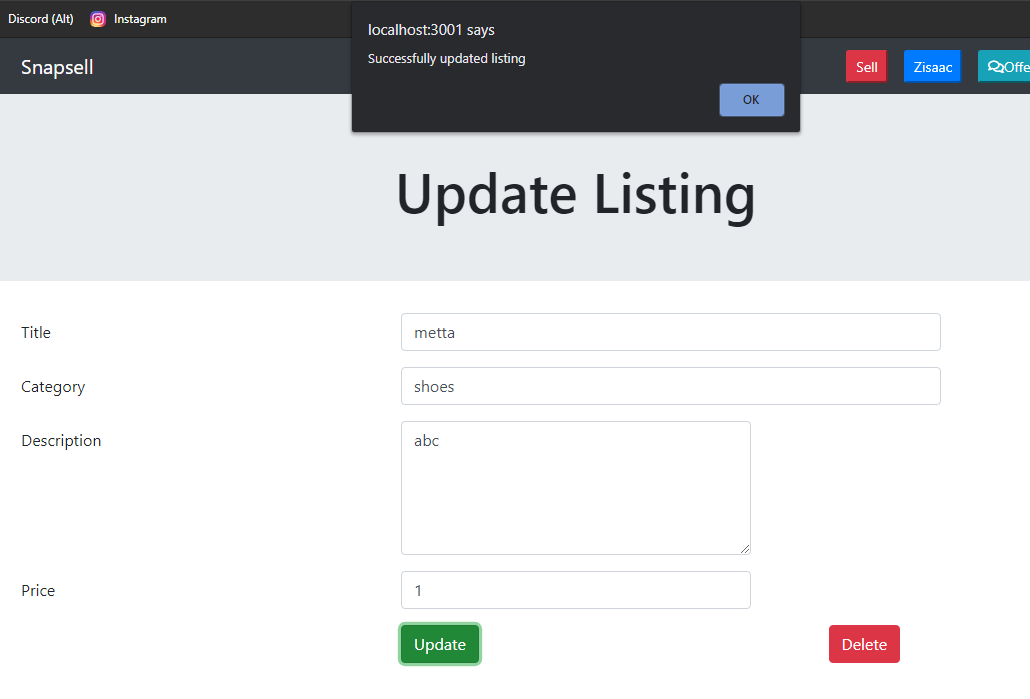
We can see in the URL that it displays the page name, ‘viewlisting.html’ and the ID of the listing, 137.



As an attacker, we can exploit this by changing ‘viewlisting.html’ to ‘editlisting.html’, which will bring up the listing editor page.

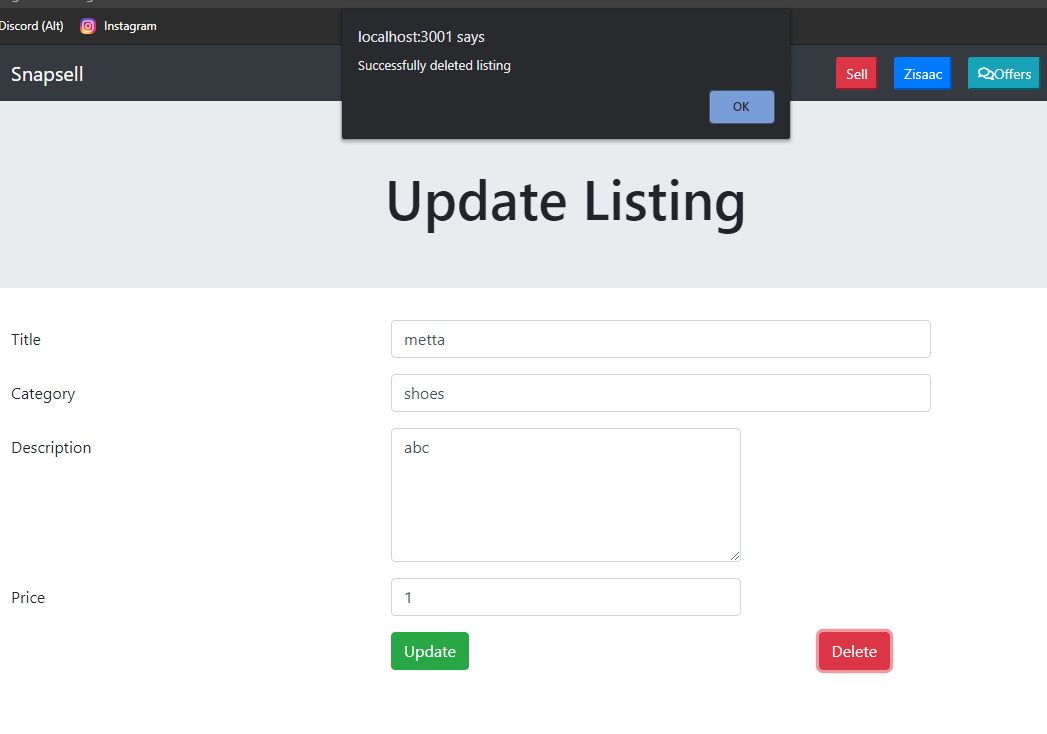


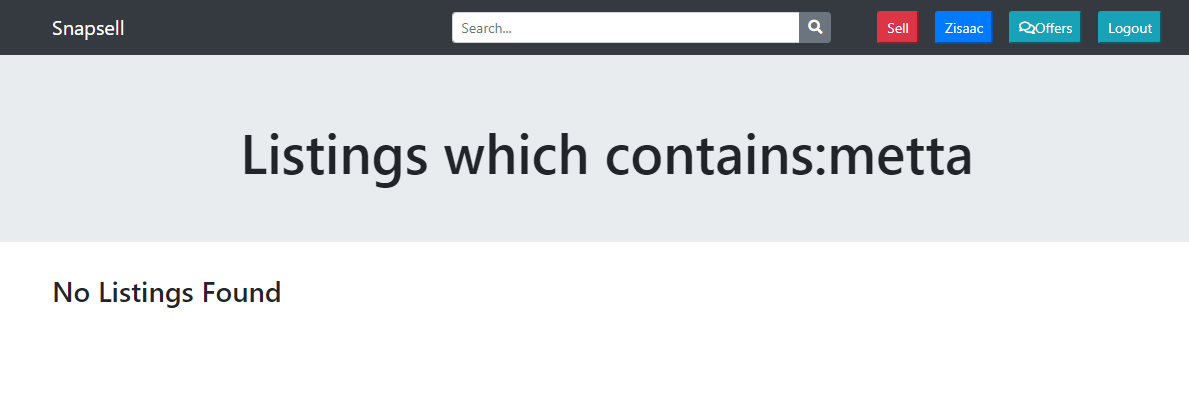
We can now modify or delete the listing of another user. First, we will change the price to 1 and press ‘Update’. As shown, the price has been changed to $1.





The attacker can even delete the listing if he chooses to do so.

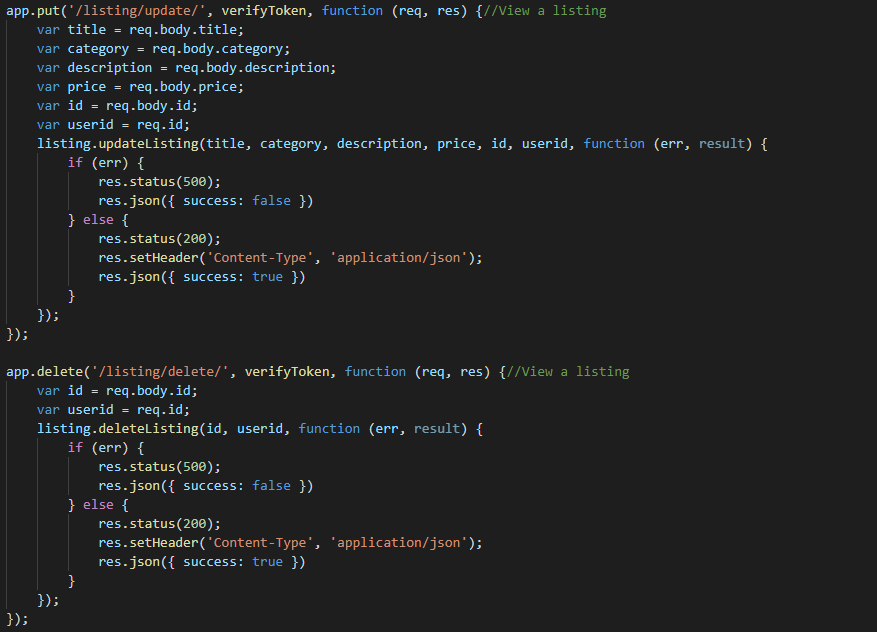




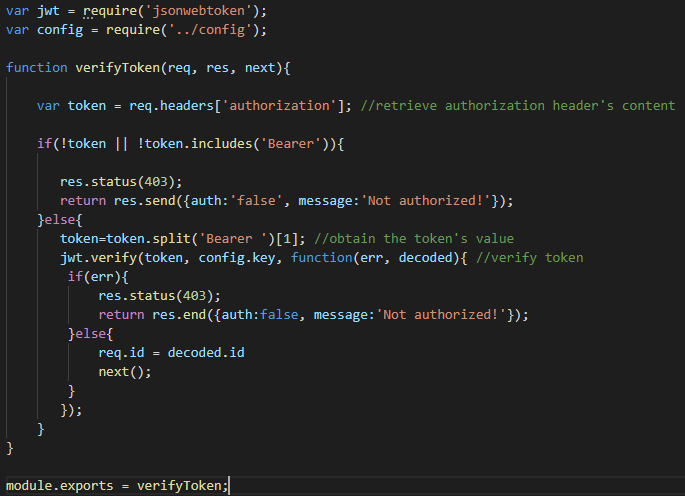
## Recommendation

### Implementation:

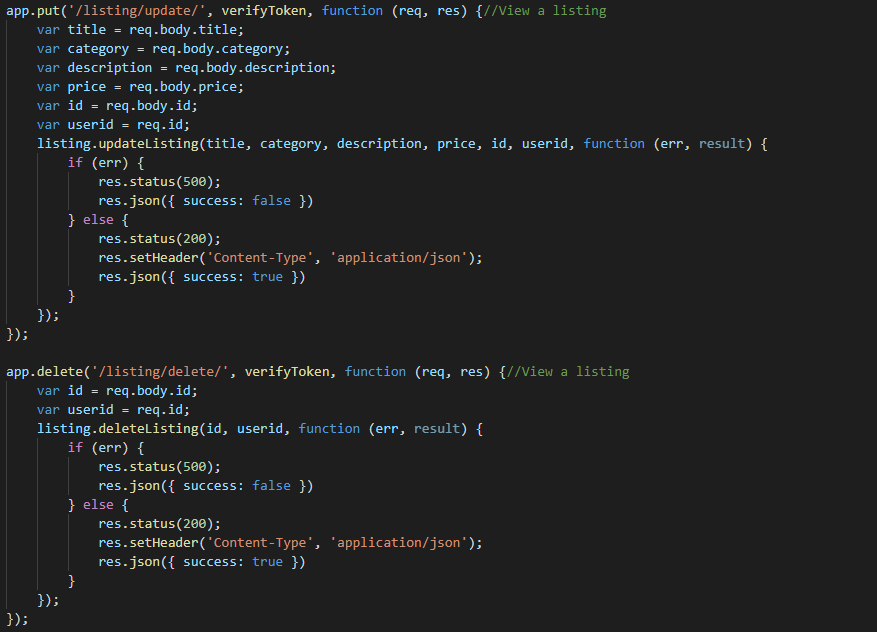
To patch this vulnerability, we recommend verifying the token of the logged in user to ensure that only the user who posted the listing can edit or delete it. Add the ‘verifyToken’ middleware which has already been created to both the update and delete listing api in app.js.



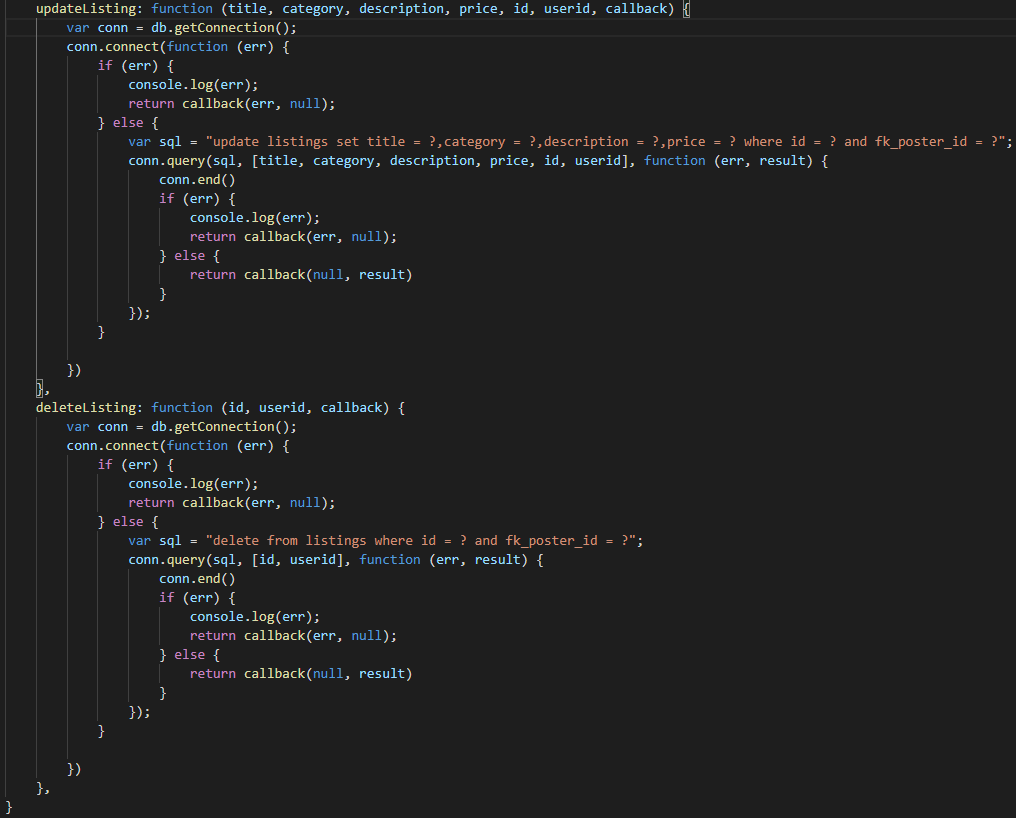
We also need to add the userid to the api which is extracted from the token as can be seen in verifyToken.js. It is called ‘req.id’.



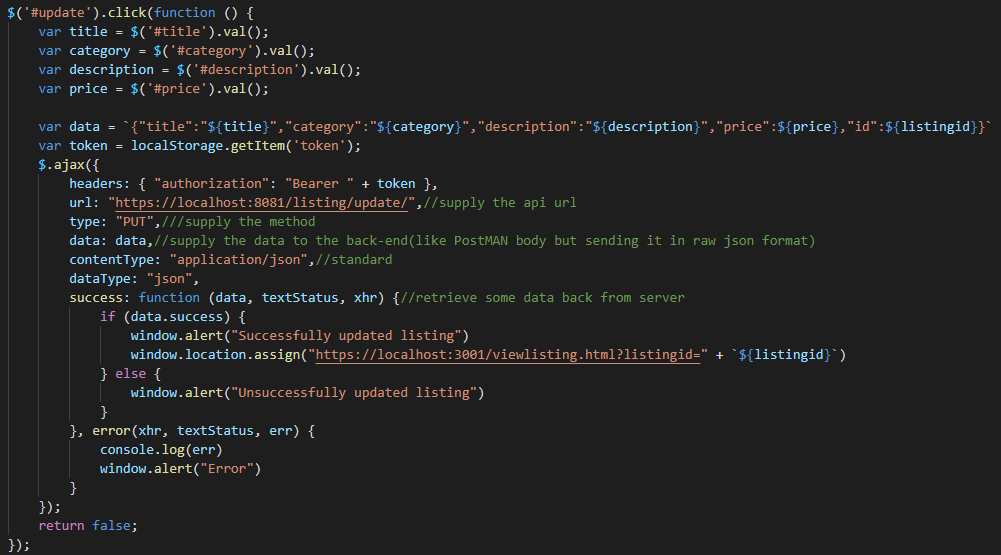
After making the changes to the api:

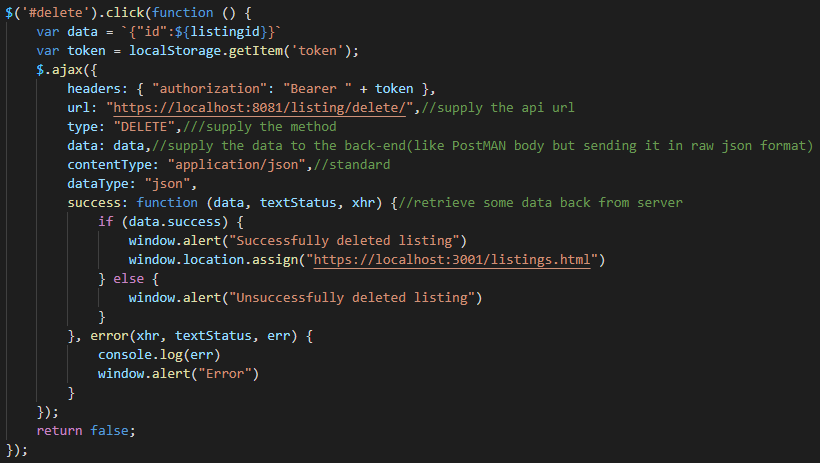


Next, we’ll need to make changes to the update and delete endpoints and SQL queries in listing.js to include the user id.



Moving on to the front-end, we will need to modify the editlisting.html page to edit the update and delete functions to verify the token before allowing the user to modify or delete listings. Add the token variable and headers to both functions.





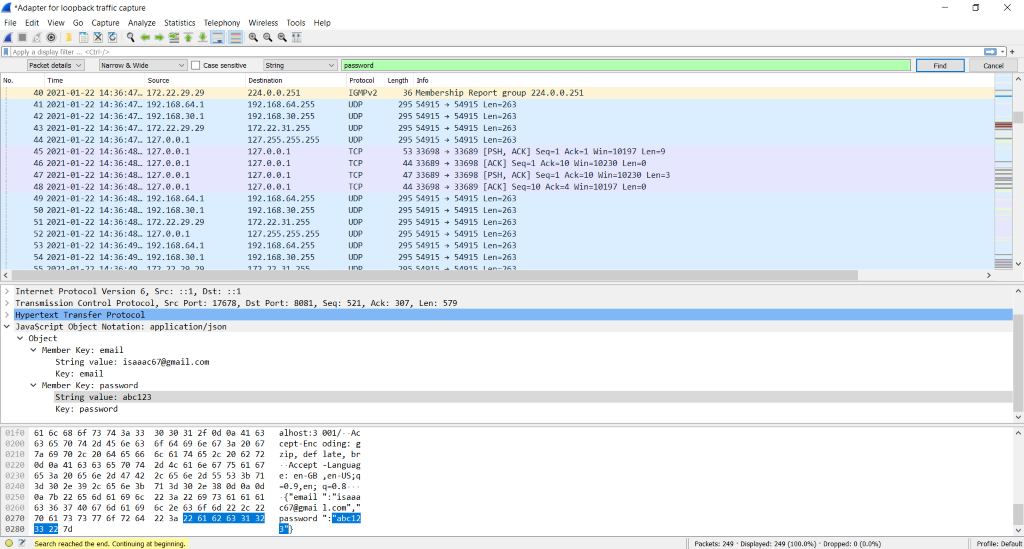
With these modifications to the code in place, if an attacker were to attempt to modify or delete another user’s listings, no changes would be applied when he presses the ‘Update’ or ‘Delete’ buttons.

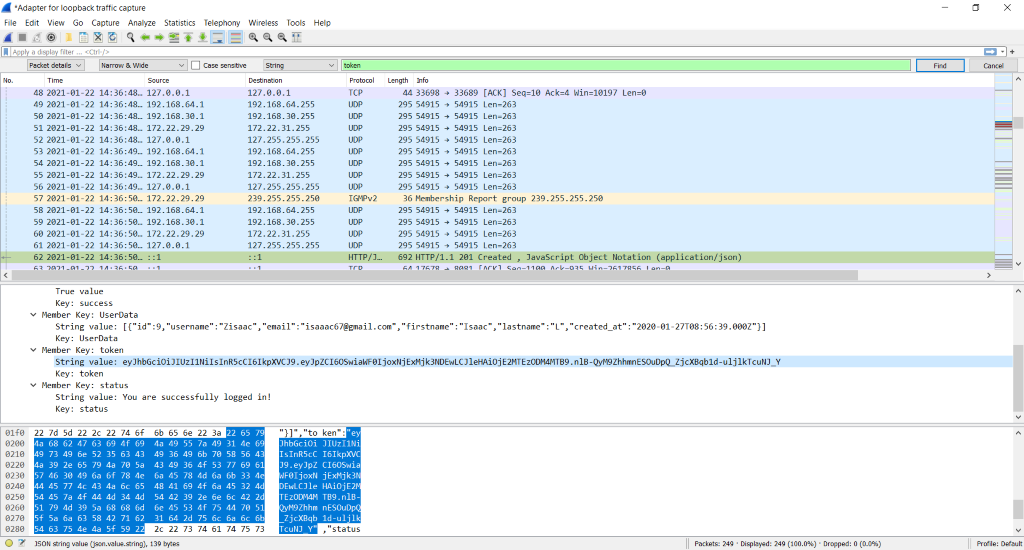
## Brief Example

If the attacker is on the same network as the victim, he can use a program like Wireshark to extract the login details and token of the victim.

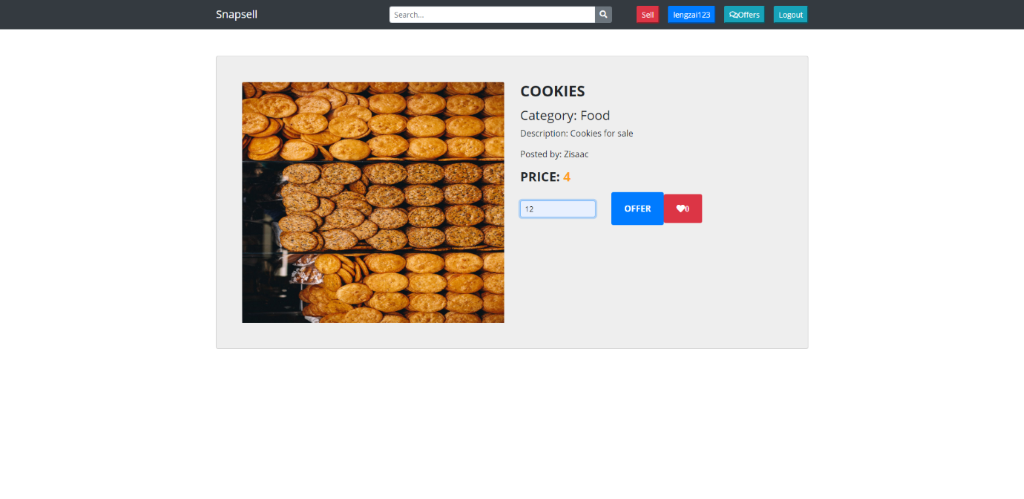
### Execution:

For example, if Zisaac were to log in to Snapsell while lengzai123 was running a capture, all of his login details including his token would be captured. We can then search up ‘password’ and ‘token’ to find the login password and token of Zisaac.

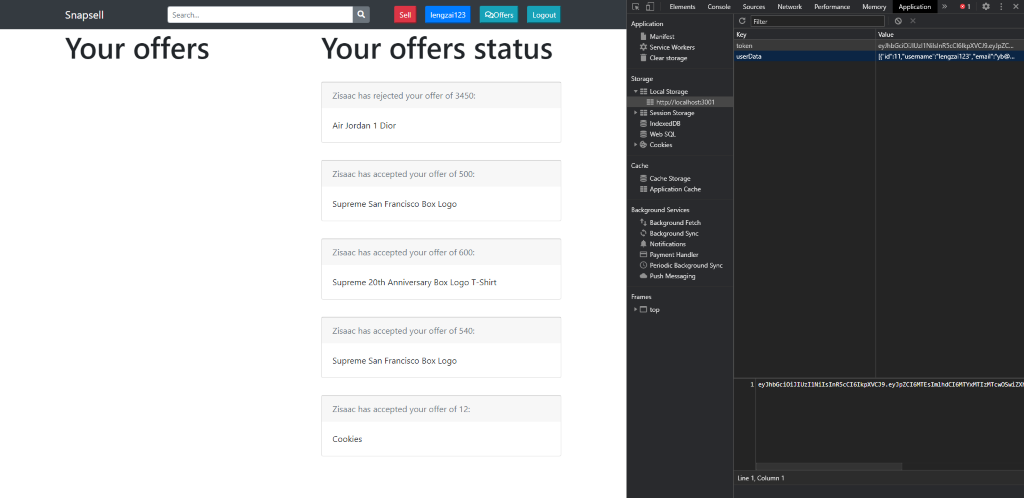




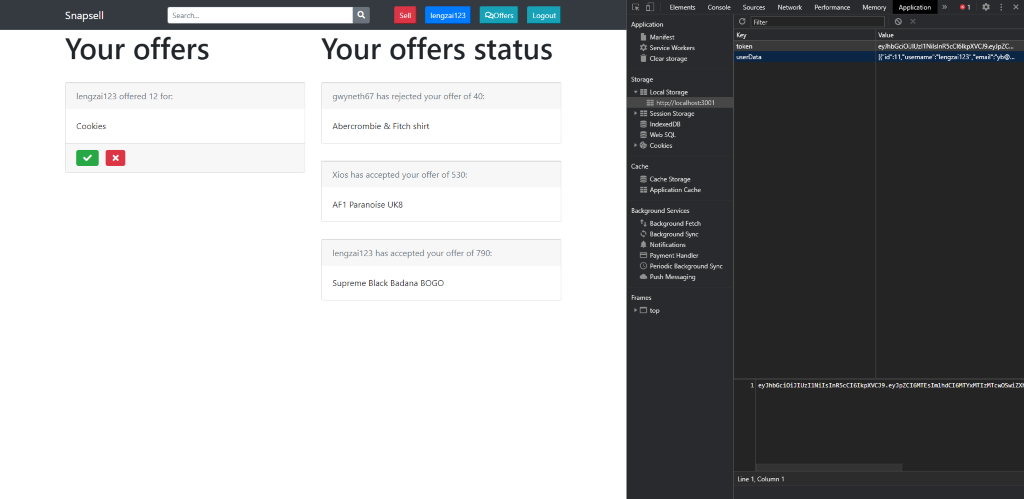
Using the lengzai123’s account, we will make an offer of $12 to Zisaac for his cookies.



Next, we will go to the ‘Offers’ page and bring up the Developer Console. Go to the Application tab where it shows the Local Storage.



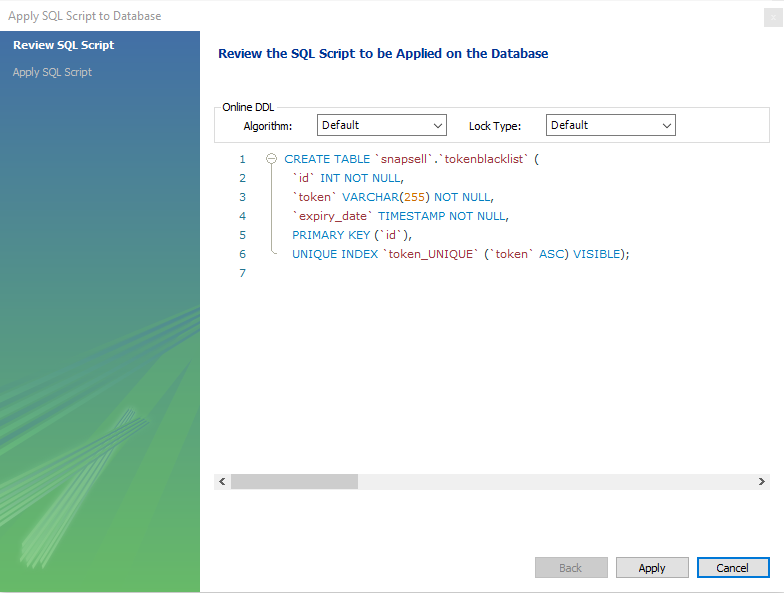
Now using Zisaac’s token which we found through Wireshark earlier, we can replace lengzai123’s token with it. After reloading the page, it shows Zisaac’s offers instead of lengzai123’s. From here, we can accept the offer for Cookies under Zisaac’s name.



## Recommendation:

Since this issue is caused by JWTs still being valid even after a user logs out, a solution must be implemented to invalidate said user’s token once the ‘Logout’ button is clicked.

To achieve this, the MySQL database will be used to monitor tokens belonging to users that have logged out. This is necessary as it is impossible to invalidate a JWT on command.

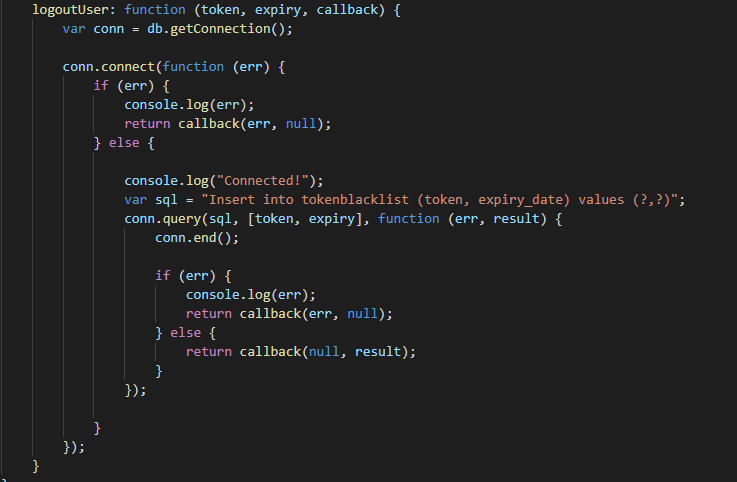




Seen above is the **create** statement for the table (tokenblacklist) that will act as a blacklist for logged out tokens. It consists of two columns (excluding id), ***token*** (VARCHAR)and ***expiry\_date*** (TIMESTAMP). As both names suggest, the former will store JWT values while the latter will store the expiry date of each token.

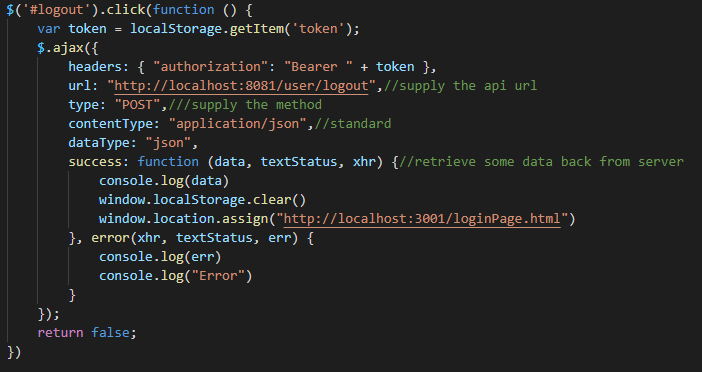
Once the table is created, edits can be made to **user.js** to include a function to insert the JWT and expiry date into the table.

**users.js**

****

Next, adjustments have to be made for the ajax sections for log out in the html files to include the JWT in ‘authorization’ header. (listings.html will be used as an example).

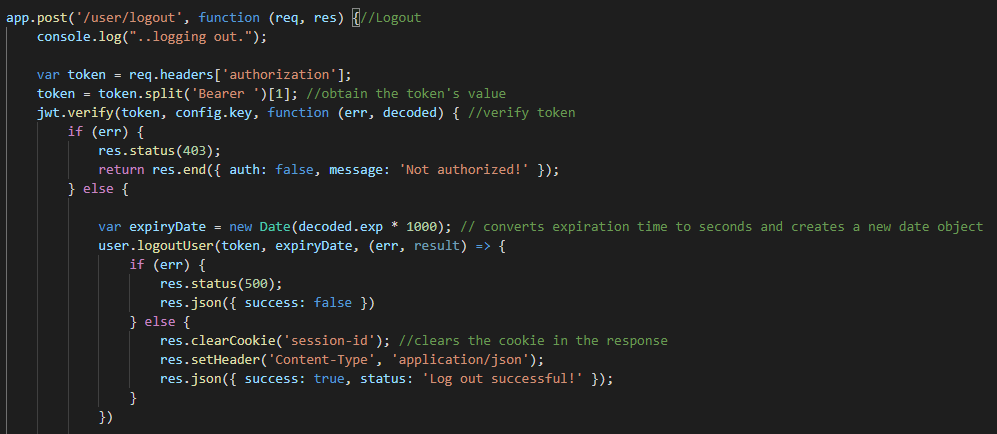
**listings.html**



After that, the ‘/user/logout/’ API will have to be edited to decode the token (similar to in verifyToken.js). Thus, we will need the ***jsonwebtoken*** node package and the ***config.js.***

**app.js**





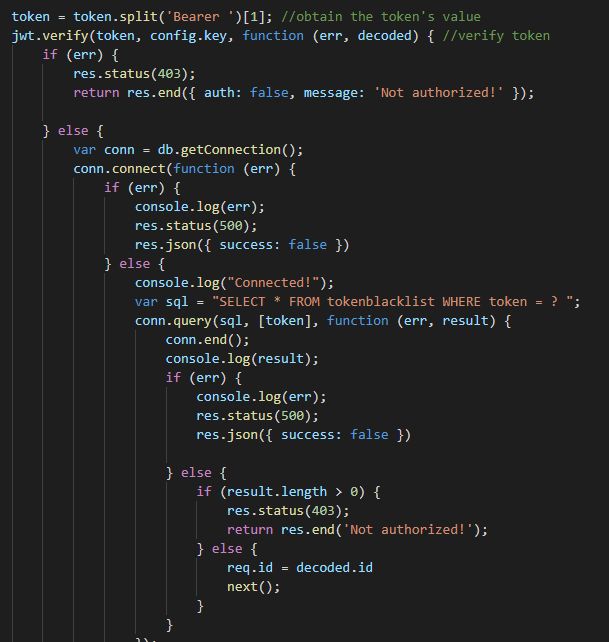
Once decoded, the token value and its expiry date are passed into the ‘user.logoutUser()’ function, where it will it be sent to the database.

**tokenblacklist**



Next, edit verifyToken.js to include a SELECT SQL query to retrieve any JWTs from the ***tokenblacklist*** table that have the same value as the client’s current token.

**verifyToken.js**

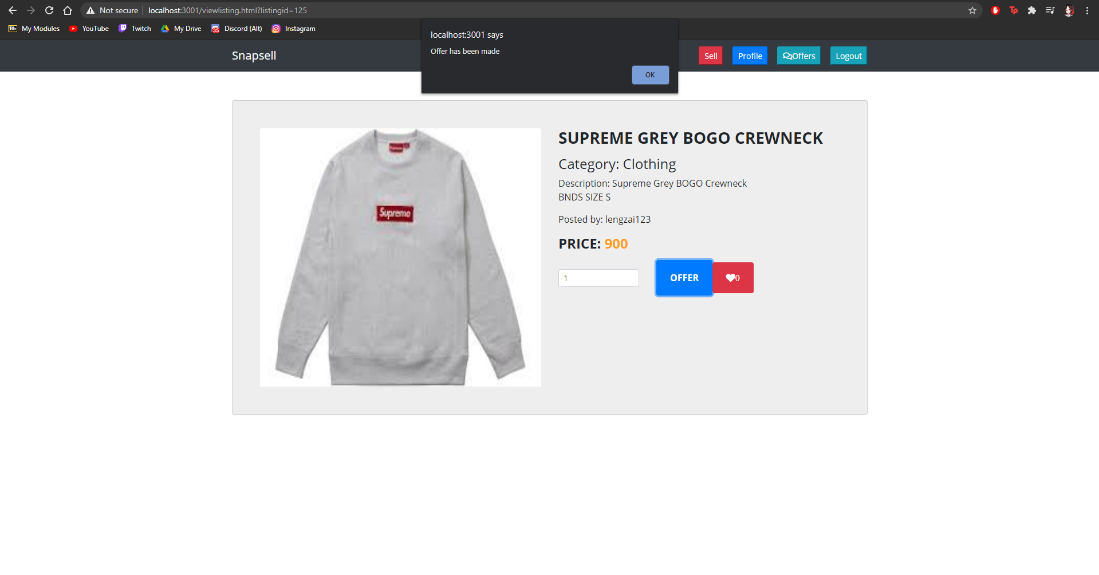


If there is a result returned, it means that the client’s current token is present in the blacklist and is a stolen JWT. Thus, the token will not be accepted a “Not authorized!” is returned. If there is no result, the backend continues to operate as normal.

Lastly, the expiry date stored in the table can be used in a MySQL EVENT to delete a row after its corresponding token has expired.

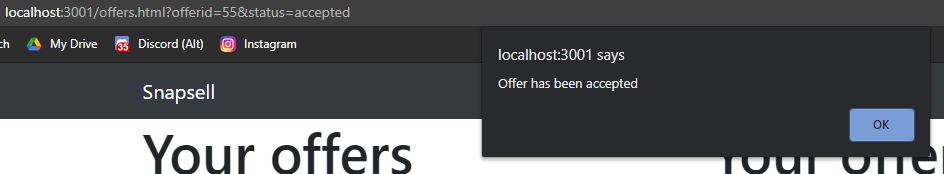
## Brief Example

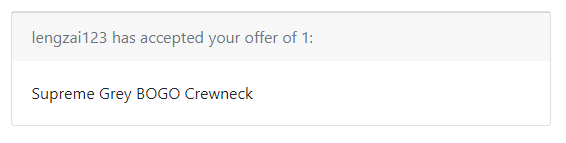
A broken access control vulnerability in Snapsell allows an attacker to accept offers of other users. The attacker can exploit this by making an offer of an incredibly low price, e.g. $1 and then accepting it under the listing owner’s name without his knowledge or permission. This attack can be carried out without needing the login details or token of the victim.



### Execution:

The vulnerability lies in the URL, which can be changed to [**https://localhost:3001/offers.html?offerid=55&status=accepted**](https://localhost:3001/offers.html?offerid=55&status=accepted)where 55 is the id of the attacker’s offer. This URL will send a request to the server to change the status of the offer with ID 55 to ‘accepted’ in the database.



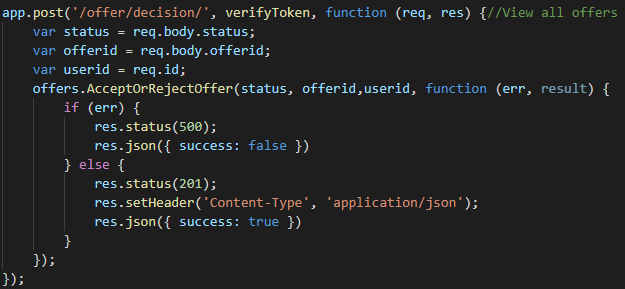


## Recommendation

To patch this vulnerability, we recommend verifying the token of the logged in user to ensure that only the user who posted the listing can edit or delete it. Add the ‘verifyToken’ middleware which has already been created to both the update and delete listing api in app.js.

### Implementation:

The solution to fix this vulnerability is similar to that of the previous example. We will need to apply token verification to the offers api in charge of accepting or rejecting the offer. We also need to pass in userid as a parameter.

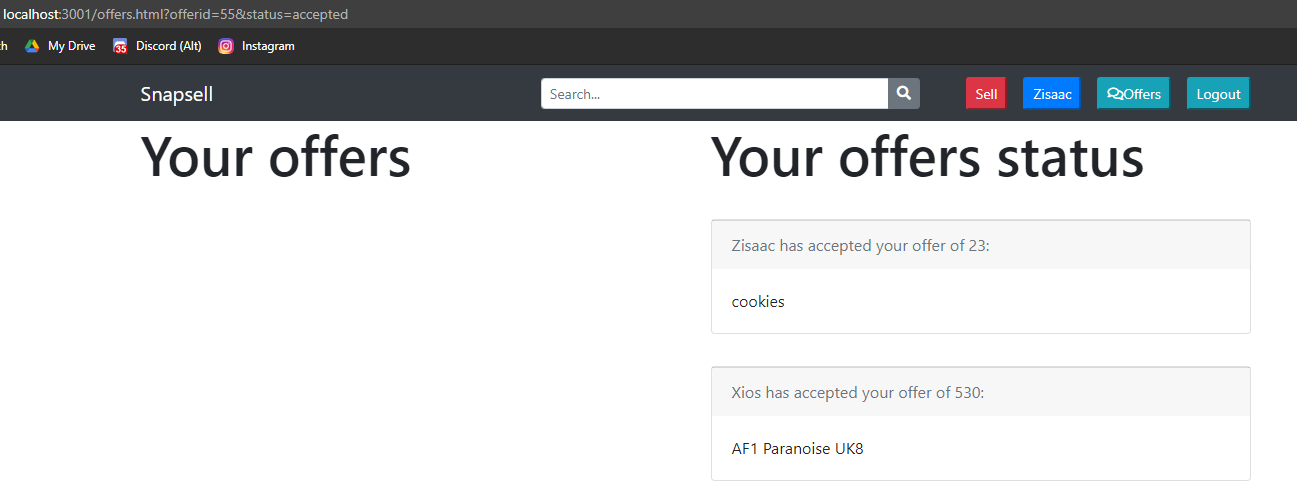


Next, we will need to make changes to the AcceptOrRejectOffer function in ‘offer.js’. Since the offers table does not contain a column to store the id of the user who posted the listing, we will need to make a separate SQL query to retrieve the poster id. Then, we will compare the poster id with the id of the currently logged in user. The function will only allow the user to update the status of the offer if the id matches.



### Execution:

With these changes in place, when the URL is changed to [**https://localhost:3001/offers.html?offerid=55&status=accepted**](https://localhost:3001/offers.html?offerid=55&status=accepted)**,** Snapsell just redirects to the offers page of the currently logged in user.



# Cross-site Scripting (XSS)

## Risk Assessment

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| High | High |
| Cross-site scripting, or XSS is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. In this attack, the attacker manipulates a vulnerable website so that it returns malicious Javascript code to other users’ browsers. When the code is executed in the victim’s browser, the attacker can potentially compromise their interaction with the application. There are 2 types of XSS attacks, stored XSS and reflected XSS. Stored XSS occurs when malicious javascript code is injected into a web application, while reflected XSS happens when malicious javascript code is reflected off a web application and onto a user’s browser. | XSS vulnerabilities are quite common in web applications. They are found in around two-thirds of all applications, hence earning themselves the number 7 spot on the OWASP Top 10 Web Application Security Risks. XSS attacks have maintained a position in the OWASP Top 10 Web Application Security Risks for over a decade.   XSS attacks are rampant largely due to the fact that most browsing experiences are heavily reliant on JavaScript. Although perceived to be less dangerous than other vulnerabilities due to JavaScripts limited access to the user's operating system and files, XSS attacks can deal significant damage through the user's browser. |

## Detailed Example (Stored XSS)

The API to post new listings to the database is vulnerable to stored XSS attacks.

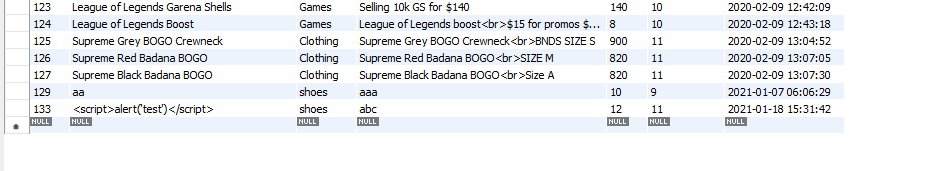
**Code snippet: listing.js**



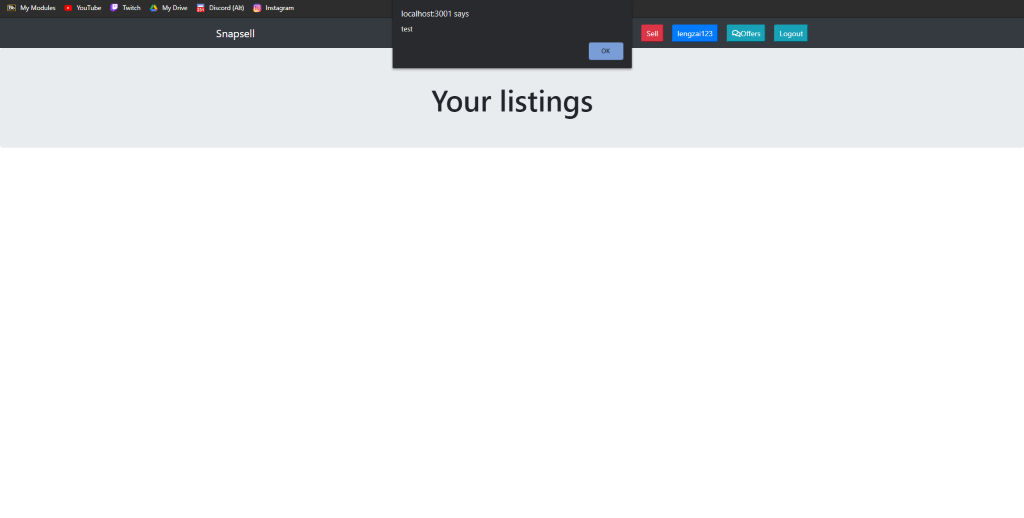
### Execution:

Due to this, an XSS attack can be conducted through the sell.html page. When the attack enters Javascript code into any of the fields and submits the listing, it will be passed into the database and run as code. Enter **<script>alert('test')</script>** into the ‘Title’ field and fill in the other fields normally.

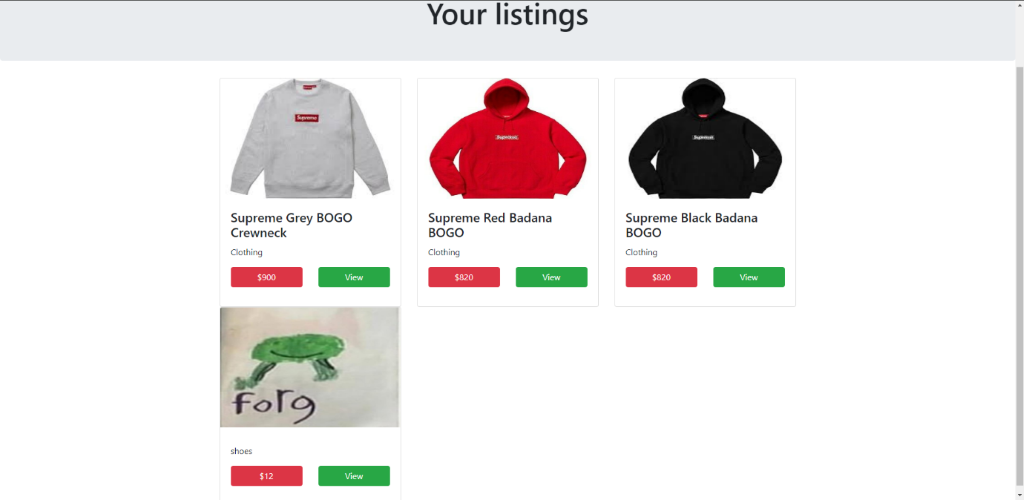




In this example, the code will create a pop-up alert that reads ‘test’ every time the user visits the ‘Your listings’ page.



The listing added by the attacker still appears as a regular listing.



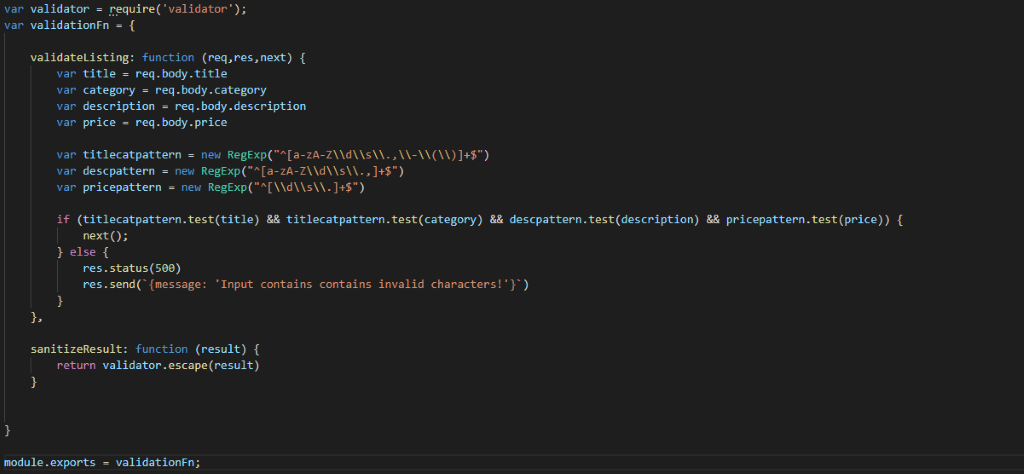
## Recommendation

The reason such XSS attacks can occur is due to the lack of input validation. Hence, when the database receives the input, it recognises the input as normal code and runs it. One method of preventing this is input validation and sanitization.

### Implementation:

For this, we created a new file called ‘validation.js’. We also need to install the validator module by entering **npm install validator** into the terminal.

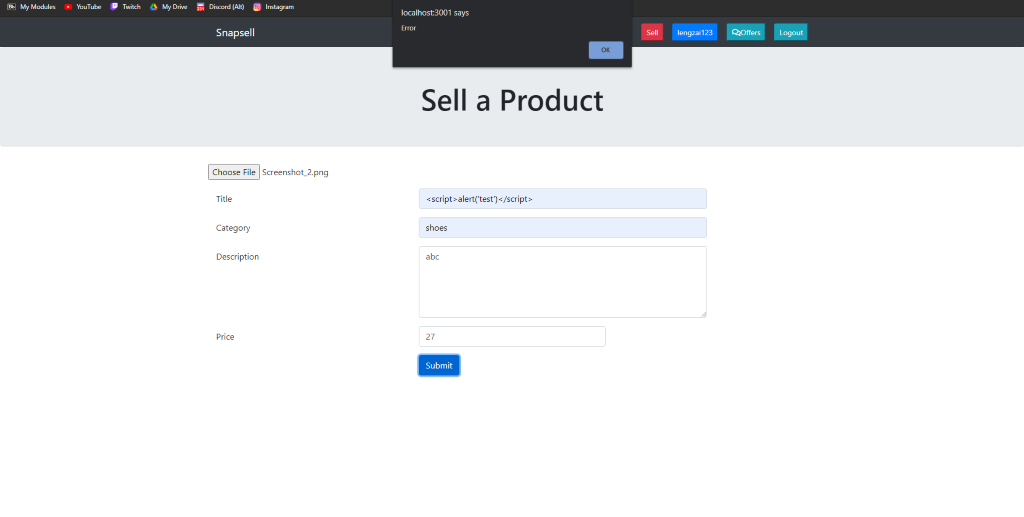
Below is the contents of the ‘validation.js’ file.



The code contains a function that matches user input with a regular expression before sending it to the database. The function is exported into the ‘app.js’ file and applied as middleware. This helps to ensure that no malicious code is submitted to the database. The result is also sanitized to ensure that malicious code is not returned to the user.

### Execution:

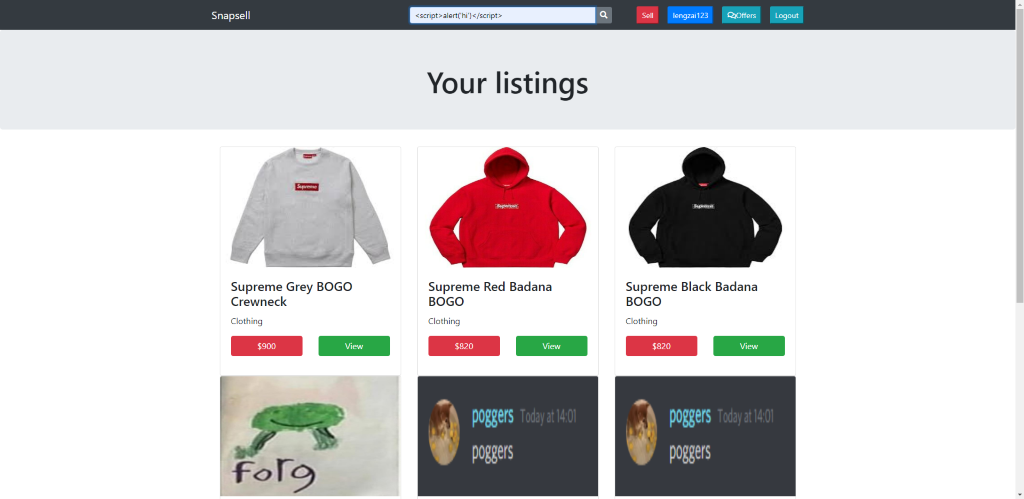
Now, when we enter the same script code into the ‘Title’ field and try to submit the listing, an error is returned.

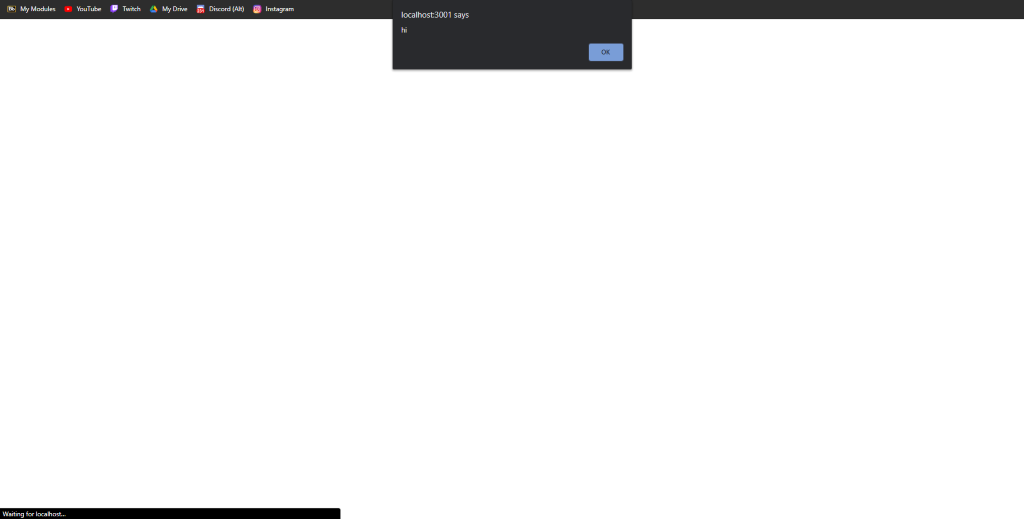


## Brief Example (Reflected XSS)

### Execution:

A reflected XSS attack can be carried out on the search bar. By entering **<script>alert('hi')</script>** into the search field and sending it, the attacker can cause an alert to appear on the user listings page.





## Recommendation

The reason a reflected XSS attack can be carried out on the search bar is due to the lack of output sanitization. Output sanitization is necessary here to ensure that no malicious script will be run when the search user input is appended to the back of the query api URL (line 36). Hence, we can apply output sanitization to the search bar vulnerability to prevent reflected XSS attacks from occurring.

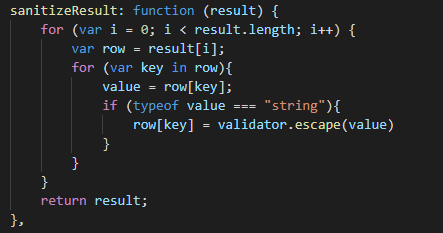
### Implementation:

To achieve this, we will need the ‘validator’ module, which can be installed by entering **npm install validator** in the back-end terminal.

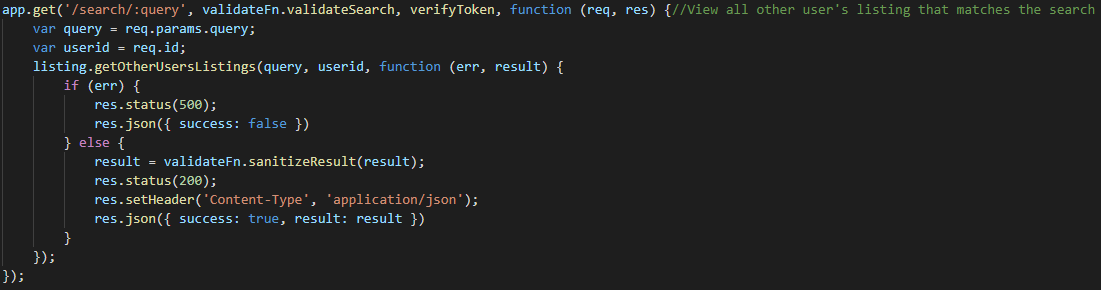
**Code snippet: validation.js**

Ensure that the line var validator = require('validator'); is present at the beginning of the file.

In the validation.js file, we will add a new function in the validationFn variable to sanitize the result.



Next, we need to add the line result = validateFn.sanitizeResult(result); to the search query API to apply the output sanitization.



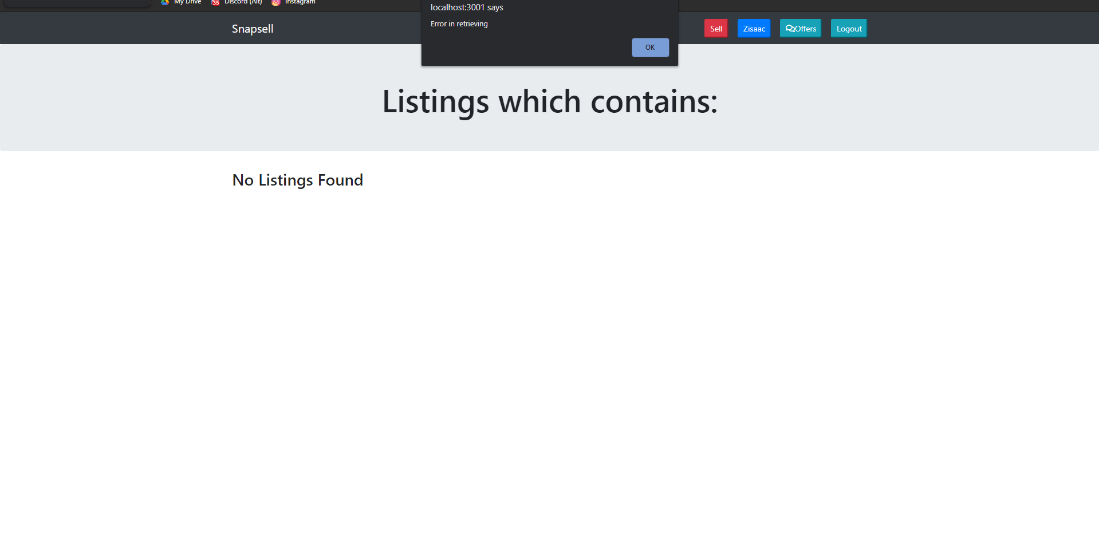
**Code snippet: searchlistings.html**

In the $(document).ready() function, we can use a regex to sanitize the search input string before it is appended to the back of the api URL. The line of code var search = search.replace(reg, "-"); parses through the user input search string and replaces instances of any of the characters <,>,&,’,”,/ with a harmless character, ‘-’.

### 

### Implementation:

After applying output sanitization to the searchlistings.html page, an error is returned when a malicious query such as **<script>alert(‘hi’)</script>** is entered into the search bar.



# Conclusion

Security vulnerabilities can be found on numerous applications across the web today. The most common types of web security vulnerabilities include SQL Injection, Broken Authentication, Sensitive Data Exposure, Broken Access Control and Cross-site Scripting, due to their nature of being difficult to prevent or simply due to developers overlooking them. These vulnerabilities can be exploited in several ways, potentially causing massive damage to web applications involving loss of sensitive information. Therefore, it is crucial that web developers are aware of the types of attacks that can be conducted on their applications, and practice secure coding to ensure that web applications such as Snapsell are safe.

# References

Cheatsheetseries.owasp.org. 2021. SQL Injection Prevention - OWASP Cheat Sheet Series. [online] Available at: <https://cheatsheetseries.owasp.org/cheatsheets/SQL\_Injection\_Prevention\_Cheat\_Sheet.html> [Accessed 7 February 2021].

Owasp.org. 2021. Cross Site Scripting (XSS) Software Attack | OWASP Foundation. [online] Available at: <https://owasp.org/www-community/attacks/xss/> [Accessed 7 February 2021].

Academy, W. and scripting, C., 2021. What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy. [online] Portswigger.net. Available at: <https://portswigger.net/web-security/cross-site-scripting> [Accessed 7 February 2021].

Academy, W. and control, A., 2021. Access control vulnerabilities and privilege escalation | Web Security Academy. [online] Portswigger.net. Available at: <https://portswigger.net/web-security/access-control#:~:text=Broken%20access%20control%20vulnerabilities%20exist,to%20be%20able%20to%20access.> [Accessed 7 February 2021].

Owasp.org. 2021. Broken Access Control for Software Security | OWASP Foundation. [online] Available at: <https://owasp.org/www-community/Broken\_Access\_Control> [Accessed 7 February 2021].

Us.norton.com. 2021. Sensitive data exposure: What is it and how it’s different from a data breach. [online] Available at: <https://us.norton.com/internetsecurity-privacy-sensitive-data-exposure-how-its-different-from-data-breach.html#:~:text=Sensitive%20data%20exposure%20occurs%20when,attacker%20accesses%20and%20steals%20information.> [Accessed 7 February 2021].

Hdivsecurity.com. 2021. What is and how to prevent Sensitive Data Exposure | OWASP Top 10 (A3). [online] Available at: <https://hdivsecurity.com/owasp-sensitive-data-exposure> [Accessed 7 February 2021].

Ropesec.com. 2021. What is a Sensitive Data Exposure Vulnerability?. [online] Available at: <https://ropesec.com/articles/sensitive-data-exposure/> [Accessed 7 February 2021]. Immuniweb.com. 2021. OWASP Top 10 Security Vulnerabilities in 2021 | ImmuniWeb. [online] Available at: <https://www.immuniweb.com/resources/owasp-top-ten/> [Accessed 7 February 2021].

Owasp.org. 2021. SQL Injection | OWASP. [online] Available at: <https://owasp.org/www-community/attacks/SQL\_Injection> [Accessed 7 February 2021].

Auth0 - Blog. 2021. What Is Broken Authentication?. [online] Available at: <https://auth0.com/blog/what-is-broken-authentication/#:~:text=Broken%20authentication%20is%20an%20umbrella,to%20impersonate%20legitimate%20users%20online.&text=Both%20are%20classified%20as%20broken,IDs%20or%20stolen%20login%20credentials> [Accessed 7 February 2021].

Hdivsecurity.com. 2021. What is and how to prevent Broken Authentication | OWASP Top 10 (A2). [online] Available at: <https://hdivsecurity.com/owasp-broken-authentication> [Accessed 7 February 2021].

SECURITY, C., 2021. Broken Authentication. [online] Contrastsecurity.com. Available at: <https://www.contrastsecurity.com/knowledge-hub/glossary/broken-authentication#:~:text=Authentication%20is%20%E2%80%9Cbroken%E2%80%9D%20when%20attackers,of%20broken%20authentication%20is%20widespread> [Accessed 7 February 2021].

Owasp.org. 2021. A3:2017-Sensitive Data Exposure | OWASP. [online] Available at: <https://owasp.org/www-project-top-ten/2017/A3\_2017-Sensitive\_Data\_Exposure> [Accessed 7 February 2021].

Inside Out Security. 2021. What is a Brute Force Attack? Definition | Varonis. [online] Available at: <https://www.varonis.com/blog/brute-force-attack/> [Accessed 7 February 2021].

npm. 2021. *validator*. [online] Available at: <https://www.npmjs.com/package/validator> [Accessed 7 February 2021].

Medium. 2021. How to log out when using JWT. [online] Available at: <https://medium.com/devgorilla/how-to-log-out-when-using-jwt-a8c7823e8a6> [Accessed 7 February 2021].