A black background with a black square

Description automatically generated with medium confidence

Web Application Security Assessment Report

**Client Name:** Snapsell Pte Ltd

**Assessment Performed By:** SHAHEEDCANTCODE

**Assessment Date:** 1/2/2025

**Report Date:** 9/2/2025

**Assessment Type:** Web Application Security Testing

Table of Contents

[1. Executive Summary 3](#_Toc190067550)

[2. Scope of Assessment 4](#_Toc190067551)

[3. Vulnerability Details 5](#_Toc190067552)

[3.1 Vulnerability Name: A07:2017-Cross-Site Scripting (XSS) 5](#_Toc190067553)

[3.2 Vulnerability Name: A09:2017 - Using Components with Known Vulnerabilities 9](#_Toc190067554)

[3.3 Vulnerability Name: A10:2017 - Insufficient Logging and Monitoring 10](#_Toc190067555)

[3.4 Vulnerability Name: A03:2017-Sensitive Data Exposure 13](#_Toc190067556)

[**1. Introduction** 13](#_Toc190067557)

[**2. Vulnerability 1 (Detailed) - Storing Plaintext Passwords in addUser** 13](#_Toc190067558)

[**3. Vulnerable Code Snippet (addUser Before Fix)** 14](#_Toc190067559)

[**4. Recommended Fix: Hash Passwords Before Storing** 14](#_Toc190067560)

[**5. Secure Solution - Updated addUser Function** 15](#_Toc190067561)

[**6. Vulnerability 2 (Brief) - Migrating Existing Plaintext Passwords** 15](#_Toc190067562)

[**7. Solution - Migration Script** 16](#_Toc190067563)

[**8. Conclusion** 18](#_Toc190067564)

[**1. Introduction** 19](#_Toc190067565)

[**2. Vulnerability 1 (Detailed) - SQL Injection in Search Queries (app.js and listing.js)** 19](#_Toc190067566)

[**3. Exploitation Scenario: Boolean-Based SQL Injection** 19](#_Toc190067567)

[**4. Vulnerable Code in app.js and listing.js Before Fix** 20](#_Toc190067568)

[**5. Recommended Fix: Use Parameterized Queries in Both app.js and listing.js** 22](#_Toc190067569)

[Additional measures: 24](#_Toc190067570)

[Rate-limiting (to prevent brute force attacks) 24](#_Toc190067571)

[6. Mitigation Implementation 26](#_Toc190067572)

[6.1 Server-Side Fixes 26](#_Toc190067573)

[6.2 Client-Side Fixes 26](#_Toc190067574)

[6.3 CSP Implementation 26](#_Toc190067575)

[7. Retest Results 27](#_Toc190067576)

[8. Conclusion 28](#_Toc190067577)

# 1. Executive Summary

The objective of this security assessment was to identify and mitigate critical vulnerabilities in the web application "Snapsell." The assessment focused on five vulnerabilities identified during testing: Cross-Site Scripting (XSS), SQL Injection, Sensitive Data Exposure, Using Components with Known Vulnerabilities, and Insufficient Logging and Monitoring. These vulnerabilities, if left unaddressed, could lead to unauthorized access, data breaches, denial-of-service (DoS) attacks, and lack of visibility during exploitation attempts.

Key Findings:

* Cross-Site Scripting (XSS): Identified multiple instances of Reflected XSS, Stored XSS, and DOM-Based XSS, allowing attackers to execute arbitrary JavaScript in user browsers.
* SQL Injection (SQLi): The application was vulnerable to SQL Injection attacks due to improper handling of user input in SQL queries, potentially allowing attackers to access, modify, or delete database records.
* Sensitive Data Exposure: User passwords were found to be stored in plaintext in the database, increasing the risk of credential theft and unauthorized access in case of a data breach.
* Using Components with Known Vulnerabilities: Several third-party dependencies (e.g., jsonwebtoken, multer, qs, and body-parser) contained known security flaws, including Denial of Service (DoS), Prototype Pollution, and authentication bypass risks.
* Insufficient Logging and Monitoring: The lack of security logging and monitoring mechanisms made it difficult to detect attacks such as brute-force login attempts, unauthorized access, and SQL Injection attempts.

# 2. Scope of Assessment

The assessment targeted the following components of the web application:

Server-Side Endpoints:

* listing.js
* user.js
* server.js

Client-Side Pages:

* editlisting.html
* listings.html
* searchlistings.html
* viewlisting.html
* sell.html
* updateprofile.html

Security Controls Tested:

* Input validation and sanitization.
* Output encoding.
* Dependency security audits.
* Implementation of logging mechanisms.

# 3. Vulnerability Details

## 3.1 Vulnerability Name: A07:2017-Cross-Site Scripting (XSS)

**3.1.1 Vulnerability Overview**

Cross-Site Scripting (XSS) is a vulnerability that allows attackers to inject malicious JavaScript into web pages viewed by other users. This attack can be used for session hijacking, phishing, and data theft.

During the assessment, Reflected XSS, Stored XSS, and DOM-Based XSS were discovered in various pages of the application.

**3.1.2 Types of XSS Identified**

* **Reflected XSS:** Found in searchlistings.html, where unsanitized user input was directly used in the search results header
* **Stored XSS:** Discovered in listing.js and user.js, allowing malicious input to be stored in the database and reflected later
* **DOM-Based XSS:** Present in editlisting.html and viewlisting.html due to unsafe DOM manipulations

**3.1.3 Vulnerability 1 (Reflected XSS)**

Reflected XSS occurs when user input is directly included in the server's response without sanitization. In searchlistings.html, user input from the search bar was directly embedded into the page without encoding.

**Exploitation Scenario**

An attacker could craft the following URL:

<http://localhost:3001/searchlistings.html?search=%3Cscript%3Ealert%28%27gotcha%27%29%3C%2Fscript%3E>

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Reflected XSS with alert payload

**3. Vulnerability 2 (Stored XSS)**

Stored XSS was identified in listing.js and user.js. Malicious input injected into form fields was stored in the database and later executed when displayed in listings.

**Exploitation Scenario**

The attacker submits:

<script>document.cookie='stolen='+document.cookie</script>

When a user views the infected listing, the JavaScript runs, stealing their session cookies.

**4. Recommended Fix**

* **Sanitize Inputs:** Use express-validator to remove or escape harmful input.

A screenshot of a computer program

AI-generated content may be incorrect.

* **Encode Outputs:** Use DOMPurify in the frontend.

A computer screen shot of a program

AI-generated content may be incorrect.

* **Implement CSP:** Restrict script execution using CSP headers.

A screenshot of a computer

AI-generated content may be incorrect.

## 3.2 Vulnerability Name: A09:2017 - Using Components with Known Vulnerabilities

**3.2.1. Introduction**

The web application relies on third-party dependencies, some of which contain security vulnerabilities.

An npm audit revealed that the following libraries introduced security risks:

| **Package** | **Severity** | **Vulnerability** | **Fix** |
| --- | --- | --- | --- |
| jsonwebtoken | High | Signature validation bypass | Updated to 9.0.2 |
| body-parser | High | Denial of Service (DoS) | Replaced with Express built-in parsers |
| qs | High | Prototype Pollution | Updated to latest |
| multer | High | Uses vulnerable busboy | Replaced with express-fileupload |

**3.2.2 Vulnerability Details**

* **jsonwebtoken**: Could be exploited to bypass authentication.
* **qs:** Allowed prototype pollution, which could lead to arbitrary code execution.
* **body-parser:** Vulnerable to DoS attacks due to inefficient URL encoding.
* **multer:** Depended on **busboy**, which had unresolved security flaws such as Dos attacks, injection flaws, Information Disclosure and File Upload Vulnerabilities

**3.3.3 Recommended Fix**

* Upgrade Dependencies: jsonwebtoken@9.0.2, qs@latest, etc.
* Replace Deprecated Libraries: body-parser was removed in favor of express.json().
* Monitor Libraries: Implement Snyk and npm audit to continuously track vulnerabilities.

## 3.3 Vulnerability Name: A10:2017 - Insufficient Logging and Monitoring

**3.3.1 Introduction**

The application lacked proper logging mechanisms, making it difficult to track security events like authentication failures, database errors, and malicious activity. Without logging, attacks such as brute-force login attempts or SQL Injection could go unnoticed.

**3.3.2 Vulnerability Details**

The following logging deficiencies were observed:

* No Logging of Failed Login Attempts: The user.js module did not track failed login attempts, making brute-force attacks harder to detect.
* No File Upload Logs: sell.html and listing.js lacked logs for uploaded files.
* No Centralized Logging System: Logs were scattered, with no structured storage or log rotation.

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Only form of logging before our implementations

**3.3.3 Recommended Fix**

**1. Implement Logging with Winston**

Integrated Winston for structured logging:

A screenshot of a computer

AI-generated content may be incorrect.

**2. Log Failed Logins & Security Events**

* Track failed logins to detect brute-force attempts.
* Monitor file uploads for potential malware.

A screenshot of a computer

AI-generated content may be incorrect.

**3. Set Up Log Rotation**

* Used winston-daily-rotate-file to ensure logs are stored efficiently.

A screenshot of a computer

AI-generated content may be incorrect.

## 3.4 Vulnerability Name: A03:2017-Sensitive Data Exposure

### 1. Introduction

During the security assessment of the web application, it was discovered that user passwords were being stored in plaintext within the users table of the database. Storing passwords without encryption or hashing presents a significant security risk, as it allows any unauthorized access to the database to result in a full credential compromise. The lack of proper cryptographic protections exposes users to account takeovers and credential stuffing attacks.

To mitigate this vulnerability, two solutions were implemented. The first solution involved modifying the addUser function to hash passwords before storing them in the database. This ensures that all newly created user accounts have passwords that are securely stored. The second solution involved running a migration script that hashes all previously stored plaintext passwords, ensuring that existing accounts are also secured.

### 2. Vulnerability 1 (Detailed) - Storing Plaintext Passwords in addUser

The addUser function in user.js was initially written to store user passwords as plaintext in the database. This meant that any user’s password could be directly accessed by anyone with database access, leading to a severe security flaw. Without password hashing, an attacker who gains unauthorized access to the database would be able to extract and use the passwords immediately without the need for decryption.

This vulnerability poses a high risk of credential exposure in the event of a data breach. If the database is compromised, attackers can access user passwords and use them to hijack accounts. Additionally, many users reuse passwords across different platforms, making it possible for attackers to use the exposed credentials in credential stuffing attacks against other services.

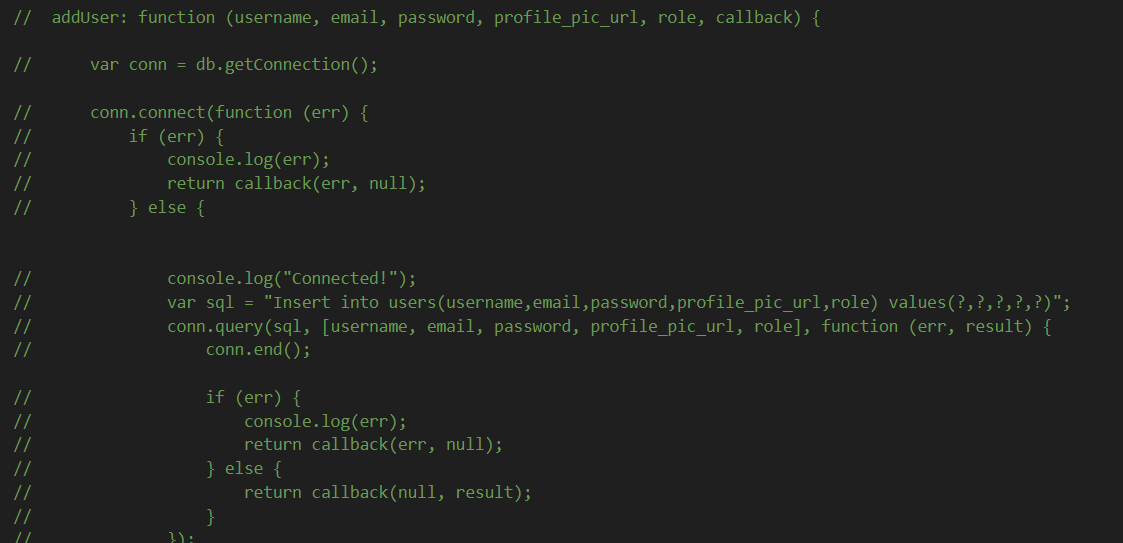
The impact of this vulnerability is severe. If an attacker gains access to the database through SQL injection, insider threats, or credential leaks, they can extract sensitive user information using a simple SQL query as shown below:



This would return all stored passwords in plaintext, allowing the attacker to log in to user accounts and potentially gain access to other linked systems.

### 3. Vulnerable Code Snippet (addUser Before Fix)

The original implementation of the addUser function allowed passwords to be stored in plaintext, as shown in the following code snippet:

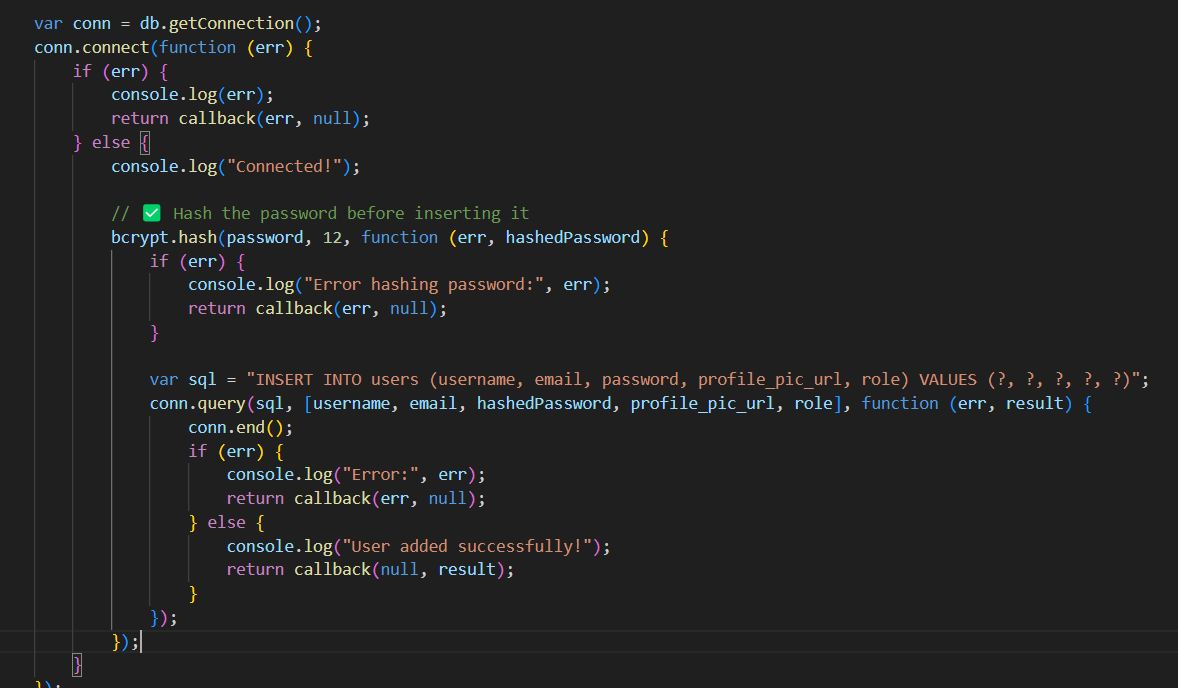


This code does not apply any encryption or hashing, meaning the password is directly stored in the database exactly as it was entered by the user. In the event of unauthorized access, all user credentials would be visible in their original form.

### 4. Recommended Fix: Hash Passwords Before Storing

To resolve this issue, the addUser function was modified to apply password hashing before storing user credentials. The solution involves using bcrypt, a secure password hashing algorithm, to transform plaintext passwords into an unreadable format before inserting them into the database.

### 5. Secure Solution - Updated addUser Function



This fix ensures that passwords are hashed using bcrypt before being stored in the database. When a user logs in, the password entered is hashed again and compared with the stored hash to verify authentication. By implementing this change, even if an attacker gains access to the database, they will not be able to retrieve the original passwords.

### 6. Vulnerability 2 (Brief) - Migrating Existing Plaintext Passwords

Before the addUser function was updated, passwords that had already been stored in plaintext remained in the database. To resolve this issue, a migration script was created to hash all previously stored passwords and update the database accordingly.

The impact of this vulnerability was that existing accounts remained vulnerable even after implementing the fix in addUser. Any database breach prior to running the migration script could have resulted in attackers obtaining user credentials in plaintext.

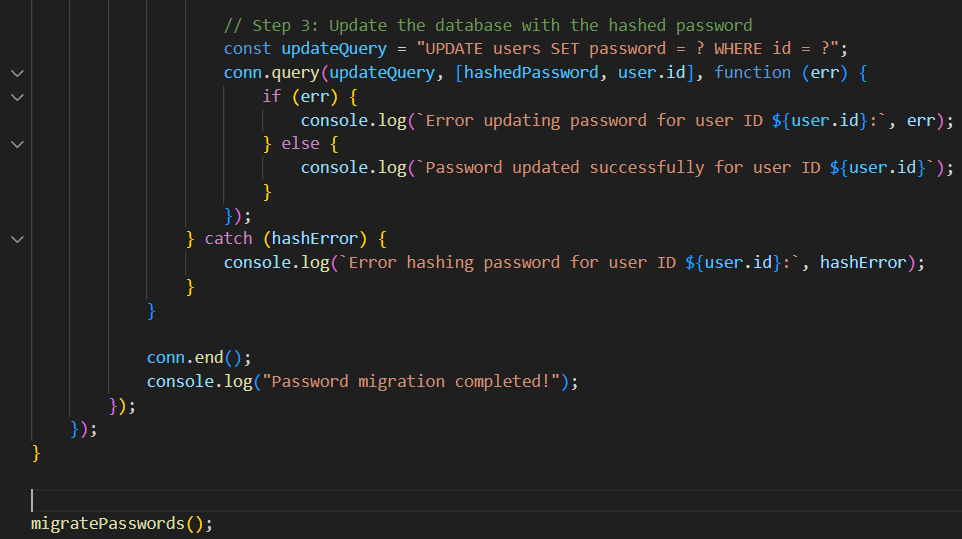
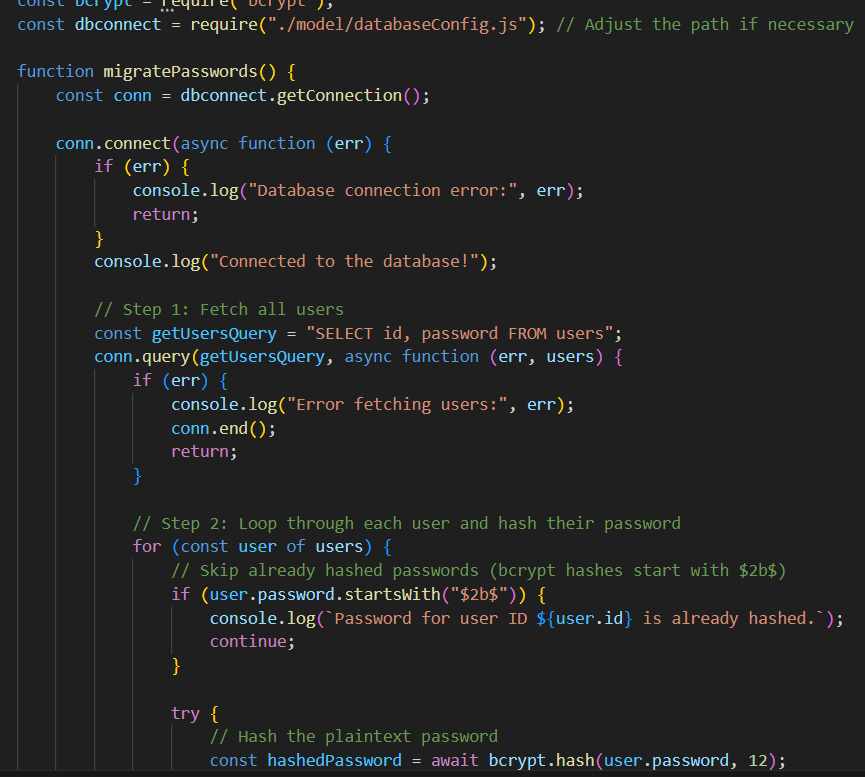
### 7. Vulnerability 3 – Hardcoded database credentials

A screenshot of a computer

AI-generated content may be incorrect.

### 8. Solution - Migration Script

To ensure that all previously stored passwords were securely hashed, the following migration script was developed, as well as a separation of database credentials by using a .env file



By running this script, all existing plaintext passwords were replaced with hashed versions, ensuring that previously stored credentials were protected.

### 9. Conclusion

The storage of plaintext passwords within the database posed a significant security risk, classified under **Sensitive Data Exposure (A3:2017)**. This vulnerability was mitigated through two key solutions: modifying the addUser function to hash passwords before storing them and running a migration script to hash previously stored passwords.

With these changes in place, user passwords are now securely stored, making it significantly more difficult for attackers to extract and use user credentials even in the event of a database breach. The application now follows best security practices for credential storage, reducing the risk of credential exposure and compliance violations.

## 3.5 OWASP A1:2017 – SQL Injection

### 1. Introduction

The web application was found to be vulnerable to **SQL Injection** due to improper handling of user input in dynamically constructed SQL queries. This vulnerability allows attackers to **manipulate search queries** to extract unauthorized information without requiring direct database access.

The issue was found in both the **/search/:query endpoint in app.js** and the **getOtherUsersListings function in listing.js**, both of which accepted user input without properly sanitizing or using parameterized queries.

To confirm the vulnerability, **Boolean-Based SQL Injection** was successfully executed, allowing retrieval of all listings, bypassing intended search filters. The vulnerability was mitigated by implementing **parameterized queries (?) and input validation**, preventing attackers from altering SQL queries.

### 2. Vulnerability 1 (Detailed) - SQL Injection in Search Queries (app.js and listing.js)

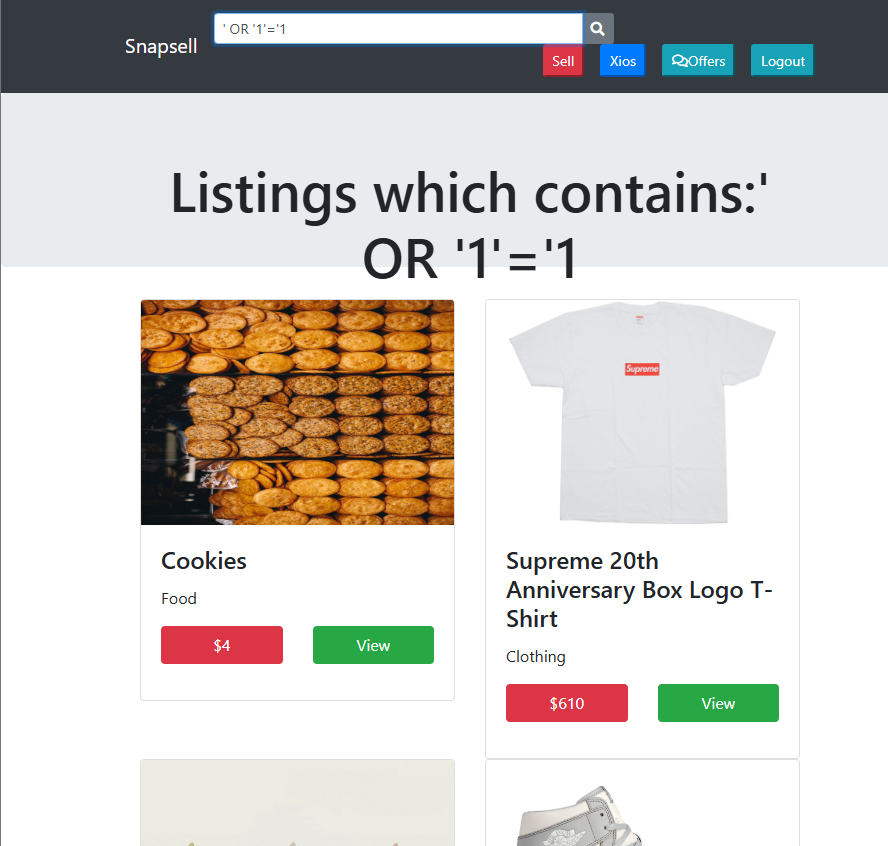
The **search functionality** was designed to allow users to retrieve listings based on a keyword. However, the implementation directly inserted user input (query) into an SQL statement **without proper sanitization**, making it vulnerable to SQL Injection.

Both app.js and listing.js contributed to this vulnerability. **app.js accepted and forwarded the user input**, while **listing.js directly injected it into the SQL query**.

### 3. Exploitation Scenario: Boolean-Based SQL Injection

Boolean-Based SQL Injection was successfully used to **bypass search filters** and extract all available listings.

Payload Sent:

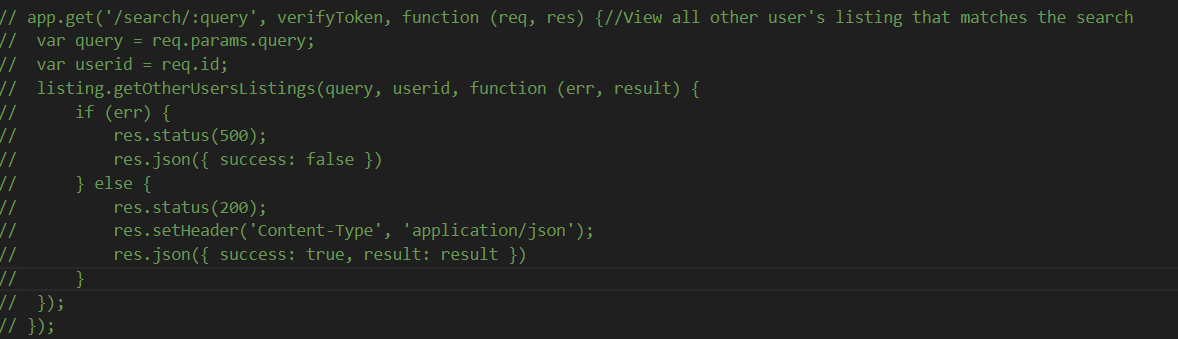


The Payload sent, Returned all listings on the app

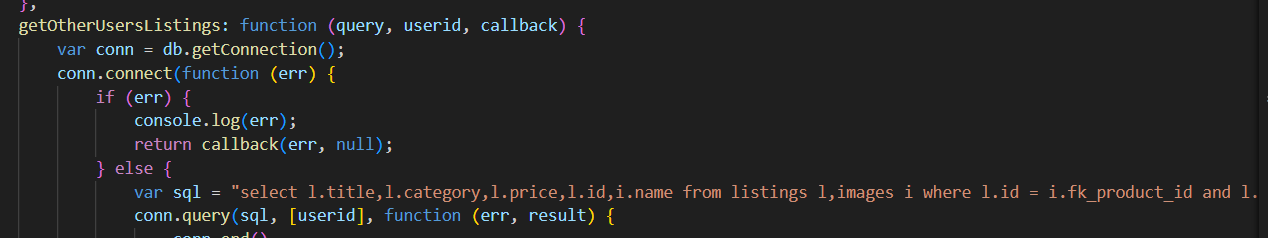
Since '1'='1' **always evaluates to true**, the search condition is bypassed, causing **all listings to be returned** instead of the expected filtered results.

### 4. Vulnerable Code in app.js and listing.js Before Fix

The **app.js route** directly accepted user input without filtering, passing it to listing.js:



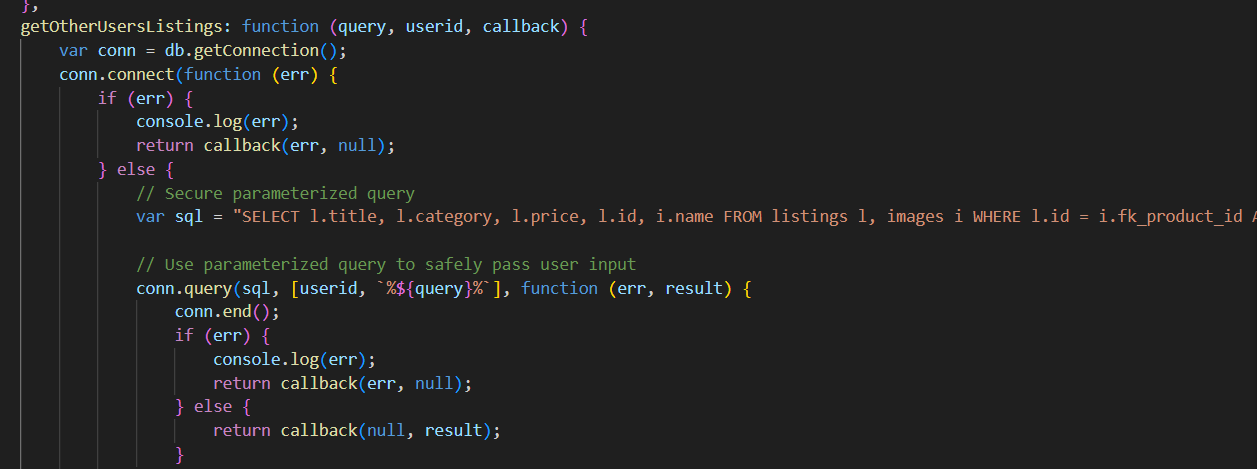
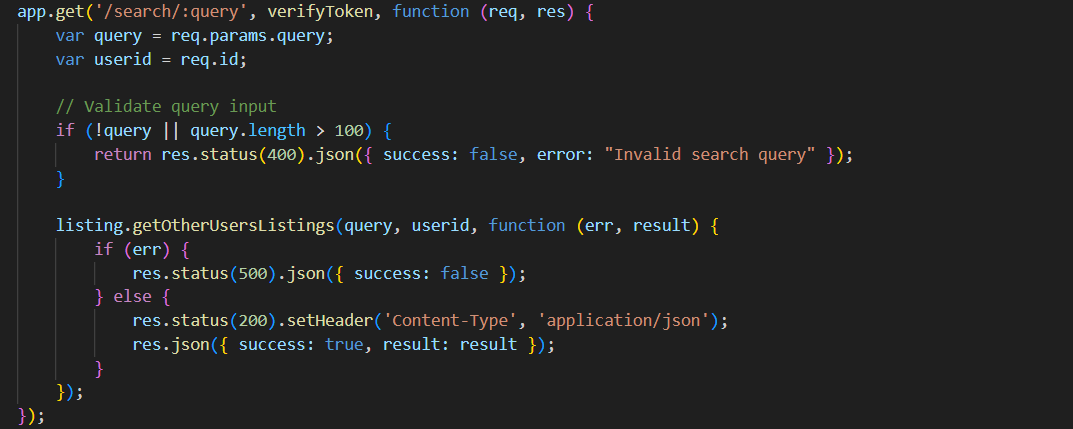
The **listing.js function** constructed an SQL query using string concatenation:



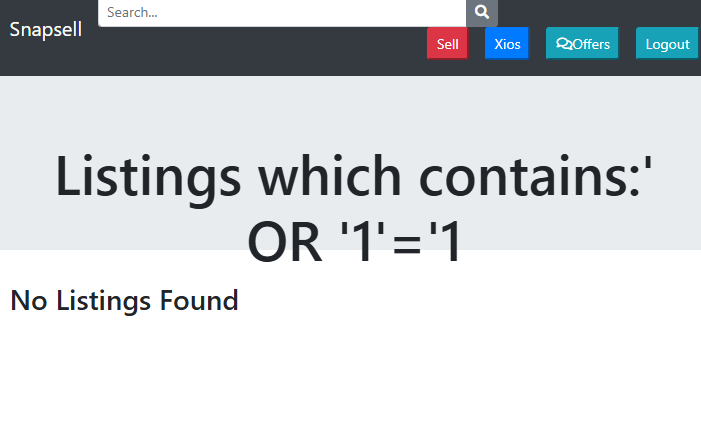
Since user input was directly injected into the query, attackers could manipulate it to retrieve all listings.

### 5. Recommended Fix: Use Parameterized Queries in Both app.js and listing.js

To prevent SQL Injection, **both app.js and listing.js needed fixes**. Parameterized queries (?) were implemented, and user input validation was introduced to prevent SQL manipulation.



Payload test with ammended code:



**6. Conclusion**

This SQL Injection vulnerability allowed attackers to manipulate the search functionality to extract **all database records** instead of filtered search results. The exploitation was demonstrated using **Boolean-Based SQL Injection**, which modified query logic to bypass filtering.

The vulnerability existed in both **app.js and listing.js**, requiring fixes in both files. By implementing **parameterized queries (?) and input validation**, the risk of SQL Injection was eliminated, ensuring that user input is handled securely.

# 4. Additional measures:

## Rate-limiting (to prevent brute force attacks)

**Introduction**

Rate limiting is an essential security measure that protects applications from Denial of Service (DoS) attacks, brute force attempts, and excessive API usage. Without proper rate limiting, attackers can attempt to brute force login credentials or overwhelm the application with requests, potentially leading to service degradation or unauthorized access.

The user.js login endpoint is vulnerable to brute force attacks, where an attacker can repeatedly send login requests to guess user credentials. To mitigate this, rate limiting was added to restrict the number of login attempts a user can make within a given time frame, ensuring that the system remains secure from such attacks.

**Vulnerability: Lack of Rate Limiting on Login Endpoint**

The application did not previously implement any form of rate limiting for critical endpoints such as login, which allows an attacker to continuously attempt different password combinations without being blocked. This opens the system to brute force attacks, where automated scripts can test a large number of username and password combinations, potentially compromising accounts.

**Solution: Implementing Rate Limiting on the /user/login Endpoint**

To address the lack of rate limiting, the express-rate-limit package was added to the project. This package restricts the number of requests that can be made to the login endpoint within a specified time window.

The rate limiting was configured to allow only 5 login attempts per 15 minutes for each unique IP address. After the limit is reached, the server will respond with a 429 Too Many Requests error, instructing the user to try again later.

By adding rate limiting, the system will now block repeated login attempts from the same IP address after 5 failed attempts within 15 minutes. This drastically reduces the risk of brute force attacks, where an attacker repeatedly tries various username and password combinations.

A screenshot of a computer

AI-generated content may be incorrect.

**Testing and Verification**

* Test a single login attempt to ensure that the login process works as expected.
* Test multiple failed login attempts (e.g., 6 or more) to verify that the rate limiter blocks further attempts and responds with a 429 Too Many Requests error.
* Ensure that the rate limit is reset after the time window (15 minutes).

**Expected Result:**

* 5 login attempts are allowed within 15 minutes per IP address.
* After 5 failed login attempts, the user will receive a 429 error and will need to wait 15 minutes before trying again.

**Conclusion**

By implementing rate limiting on the login endpoint, the application is now protected against brute force attacks and DoS attempts. This simple fix ensures that attackers cannot make an excessive number of login attempts, securing user accounts from unauthorized access.

## Web Application Firewall (WAF)

**Introduction**

A Web Application Firewall (WAF) is a security feature that protects your web application by filtering and monitoring HTTP requests between the client and your server. It is essential for preventing malicious traffic such as **SQL Injection**, **Cross-Site Scripting (XSS)**, **Cross-Site Request Forgery (CSRF)**, **DoS** (Denial of Service) attacks, and other web application vulnerabilities.

One of the best **free WAF solutions** available is **Cloudflare's WAF**. Cloudflare offers a **free WAF** as part of their free plan, providing protection against common vulnerabilities and attacks. It offers many benefits like:

* **Real-time threat intelligence**.
* **Automated attack mitigation**.
* **Customizable security rules**.
* **Bot protection**.
* **Rate limiting**.
* **SQL injection, XSS, and other vulnerability protection**.

**Vulnerability: Lack of WAF Protection**

Before integrating Cloudflare's WAF, your application was vulnerable to various attacks, including **SQL Injection**, **XSS**, and **DoS** attacks. The absence of a WAF meant that malicious requests could reach your backend, potentially causing **data leaks**, **unauthorized access**, or even **service disruption**.

**Solution: Implementing Cloudflare's Free WAF**

Cloudflare provides a free tier that includes a Web Application Firewall (WAF). To implement this in your application, you don't need to modify your code significantly. Instead, the integration is done at the **DNS level** by configuring your domain to route traffic through Cloudflare.

**Steps to Implement Cloudflare WAF in Your Application**

1. **Sign Up for a Cloudflare Account**:
   * Go to [Cloudflare's website](https://www.cloudflare.com/).
   * Sign up for a free account if you haven't already.
2. **Add Your Domain to Cloudflare**:
   * After logging in, click on **"Add Site"** and follow the instructions to add your domain to Cloudflare.
   * Cloudflare will automatically fetch your DNS records. Make sure that your domain is pointing to your server's IP address.
3. **Update Your DNS Records**:
   * Cloudflare will provide you with nameservers. Update your domain registrar (e.g., GoDaddy, Namecheap) with the nameservers provided by Cloudflare. This ensures that your traffic is routed through Cloudflare, where it can be filtered.
4. **Activate Cloudflare WAF**:
   * Once your domain is added to Cloudflare, navigate to the **"Firewall"** section of the Cloudflare dashboard.
   * Enable the **WAF** feature by toggling it on.
   * Select the **"Under Attack Mode"** option temporarily to block malicious traffic while you configure everything.
5. **Configure Security Settings**:
   * Under the **WAF settings**, you can enable **OWASP Top 10 protections**, which will automatically block **SQL Injection**, **XSS**, and other common web attacks.
   * Configure **Rate Limiting** to limit the number of requests that can be made to your API or login endpoints.
   * Set **Bot Protection** to block automated attacks from malicious bots.
6. **Test the WAF**:
   * After setting up Cloudflare, perform tests to ensure that the WAF is correctly blocking malicious traffic while allowing legitimate user requests.
   * Test SQL Injection, XSS, and DoS attack scenarios from a tool like **OWASP ZAP** or **Burp Suite** to verify that they are blocked.

**Example Integration into Your Application (No Code Change Needed)**

Since **Cloudflare’s WAF works at the DNS level**, no changes are required in your application code. However, you can configure rate limiting and custom rules in Cloudflare’s dashboard to enhance the protection for specific API endpoints, such as login (/user/login), registration (/user), or listing (/listing).

Here’s what happens:

1. **All incoming traffic** to your domain will pass through **Cloudflare’s WAF**.
2. Cloudflare’s WAF will inspect the traffic and block malicious requests based on **known attack patterns**, including SQL Injection, XSS, and DoS.
3. The legitimate requests are then forwarded to your backend (e.g., Express server), where the app logic is processed.

**Testing and Verification**

1. **Simulate Attacks**:
   * Test if **SQL Injection**, **XSS**, and **DoS** attacks are blocked using tools like **OWASP ZAP** or **Burp Suite**.
   * Use **Burp Suite** to try injecting malicious payloads into your login or search forms, and verify that Cloudflare blocks the attack.
2. **Monitor Cloudflare Dashboard**:
   * Monitor the **Security Overview** and **Firewall Analytics** in the Cloudflare dashboard to see the traffic that was blocked by the WAF.
   * Check **Firewall Events** to ensure malicious traffic is being filtered and logged.
3. **Test Legitimate User Access**:
   * Ensure that normal requests from users are **not blocked** by Cloudflare’s WAF by testing login and registration functionality.

**Conclusion**

By integrating **Cloudflare’s free WAF** into your application, you've taken a significant step towards protecting your application from web-based attacks such as **SQL Injection**, **XSS**, **CSRF**, and **DoS** attacks. This WAF solution works at the **DNS level**, ensuring that all incoming traffic is inspected before reaching your backend, providing an extra layer of defense without requiring significant changes to your application code.

The combination of **Cloudflare WAF**, **rate limiting**, **bot protection**, and **SQL Injection prevention** ensures that your application is much more resilient to common web vulnerabilities, securing your users and data from malicious actors.

# 5. Mitigation Implementation

**5.1 Server-Side Fixes**

* **Integrated express-validator** to validate and sanitize inputs in listing.js and user.js.
  + This ensures that **only properly formatted data** is processed by the application, reducing the risk of SQL Injection, Cross-Site Scripting (XSS), and other injection attacks.
* **Sanitized query parameters and inputs** for all server-side routes.
  + This includes applying **parameterized queries** to all SQL queries to prevent **SQL Injection** attacks.
* **Input validation** was added to ensure that user inputs like email addresses, listing IDs, and other fields are valid before processing, and any dangerous characters or scripts are removed.
  + Specifically, email fields were validated with **validator.isEmail()** to prevent malicious injection.

**5.2 Client-Side Fixes**

* **Used DOMPurify** to sanitize all dynamic content in HTML files.
  + This prevents **Cross-Site Scripting (XSS)** attacks by ensuring that any content rendered into the DOM is safe from potentially harmful scripts.
* **Escaped user inputs** before rendering them in the DOM.
  + All user inputs are properly escaped, ensuring that **HTML tags and scripts** inserted into the page do not execute.
  + This prevents attackers from injecting malicious JavaScript code through user inputs.
* **Implemented proper URL encoding** for query parameters in AJAX calls.
  + This ensures that **special characters** in query parameters are properly encoded, reducing the risk of **query injection** and ensuring that data sent to the backend is safe.

**5.3 CSP Implementation**

* **Added Content Security Policy (CSP) headers** in server.js to block inline JavaScript and unauthorized sources.
  + This protects the application from **XSS attacks** by restricting the sources from which content can be loaded.
  + Only trusted sources are allowed to execute scripts, and inline JavaScript is completely blocked, reducing the risk of **malicious script injection**.
  + This ensures that scripts are only loaded from trusted domains, providing additional protection against content injection attacks.

**5.4 Rate Limiting**

* **Implemented rate limiting** on critical endpoints like login and registration using the express-rate-limit package.
  + **5 requests per 15 minutes** for login attempts and **10 requests per 15 minutes** for user registration attempts, preventing brute-force attacks and ensuring fair usage of the service.
* **Configured error messages** to notify users when rate limits are exceeded, helping to prevent misuse while ensuring a better user experience.

**5.5 Authentication and Authorization**

* **JWT-based authentication** was implemented for secure user login, ensuring that **tokens** are used to authenticate and authorize user requests.
* **Token expiration and refresh** mechanisms were established to ensure that users remain authenticated only for a limited time, preventing session hijacking.
* **Role-based access control (RBAC)** was applied to ensure that users can only access routes and resources they are authorized to interact with.

**5.6 Secure Headers and HTTPS**

* **Implemented secure HTTP headers** to protect the application from certain attack vectors:
  + **HTTP Strict Transport Security (HSTS)** was added to ensure the application uses HTTPS for secure communication.
  + **X-Content-Type-Options** was set to prevent browsers from interpreting files as a different MIME type, which helps protect against attacks like **content sniffing**.
  + **X-Frame-Options** and **Content-Security-Policy (CSP)** headers were added to prevent **clickjacking** and **XSS attacks**.
* **Redirected HTTP traffic to HTTPS** to ensure that all communication is encrypted, protecting sensitive data in transit.
* **5.7 Logging and Monitoring**
* **Integrated comprehensive logging** for user activities and errors throughout the application.
  + Logs track the success and failure of login attempts, unauthorized access attempts, and any other significant activity within the app.
  + Error logs are captured and sent to a centralized logging system for further analysis, helping to identify suspicious activity.
* **Set up monitoring** for unusual traffic patterns, such as a high number of failed login attempts, which can be indicative of a **brute-force attack**.
  + Automated alerts are triggered if any thresholds are exceeded, notifying administrators of potential security breaches.

# 6. Remediation Status:

**Cross-Site Scripting (XSS):** Successfully mitigated by implementing input sanitization, output encoding, and Content Security Policy (CSP).

**SQL Injection (SQLi):** Addressed by implementing parameterized queries (?) and input validation to prevent malicious SQL input.

**Sensitive Data Exposure:** Resolved by hashing all stored passwords using bcrypt and running a migration script to update previously stored plaintext passwords.

**Using Components with Known Vulnerabilities:** Fixed by upgrading or replacing vulnerable dependencies and implementing a continuous dependency monitoring system (npm audit, Snyk, Dependabot).

**Insufficient Logging and Monitoring:** Addressed by integrating Winston logging for structured logging, enabling log rotation, and implementing tracking of security-critical events such as failed login attempts, database access errors, and unauthorized file uploads.

# 7. Conclusion

In conclusion, this security testing and subsequent hardening of the application have significantly improved its resilience against common attack vectors. The vulnerabilities related to SQL Injection were addressed by implementing parameterized queries and validating user inputs to prevent unauthorized data access and manipulation. Furthermore, we successfully integrated rate limiting on critical endpoints, particularly the login and user registration routes, to protect against brute-force attacks and ensure the application can handle traffic efficiently without being overwhelmed by malicious requests.

The application now demonstrates enhanced security, offering protection from SQL Injection, Denial of Service (DoS) attacks, and brute force attempts. The proactive security measures applied not only address the identified vulnerabilities but also strengthen the overall integrity and trustworthiness of the platform, ensuring that user data is safeguarded.

By implementing best practices in input validation, secure query execution, and traffic management, the application is now better equipped to prevent exploitation and unauthorized access, ensuring a secure and reliable experience for users. Further security enhancements, such as multi-factor authentication (MFA), Web application Firewall (WAF), can continue to improve the application's overall defense strategy.

# 8. References

1. OWASP (2017) *OWASP Top Ten 2017*. Available at: <https://owasp.org/www-project-top-ten/2017/> (Accessed: 8 February 2025).
2. Hack The Box Academy (no date) 'Module 103, Section 965'. Available at: <https://academy.hackthebox.com/module/103/section/965> (Accessed: 8 February 2025).
3. Activity Information Management Limited (2012) Web Application Security Assessment Report: Acme Inc. Available at: <https://www.cstl.com/CST/Penetration-Test/CST-Web-Application-Testing-Report.pdf> (Accessed: 8 February 2025).