

Security Analysis of InvenTracker

Kien Le, Parakh Dayal, Patrick Cunningham

InvenTracker, created for nu-inventory-tracker-company pty, uses several technologies to store inventory and user data. The application can be used to track inventory, item locations, stock levels, and suppliers. To maintain high security and privacy, leading security practices are employed that provide mitigations for many types of attacks. The web application was built with a Node.js framework, with different imported modules and dependencies to achieve the desired functionalities. In doing so, the application complies with industry best practices to increase security.

Password Hashing

Hashing refers to the process of converting data into another unique, fixed size value that is irreversible (Vaideeswaran, 2024). By hashing information, even if someone gained access to the data, they would find it extremely difficult to determine the original information, thus protecting privacy. In InvenTracker, the Node Express bcrypt module hashes passwords. The database stores these hashed values so that user inputted passwords can be hashed and compared against the stored hash to determine if they are correct. The bcrypt module was chosen because it is based on the Blowfish cipher, which is known to be a secure hashing algorithm, and it allows the user to modify the compute time simply by changing a cost value. While bcrypt is a good hashing algorithm, Argon2 is currently considered to be the best password specific hashing algorithm since it is resistant to side channel attacks and memory hard. Nevertheless, bcrypt is still a strong hashing algorithm and a good choice for the use case since it is still widely used and is more secure than alternatives like PBKDF2 (Gupta, 2024).

bcrypt requires a salt by default, which provides additional security. Salting refers to a technique where a unique, random string is added to each password before hashing (Hughes, 2024). This process helps protect against rainbow table attacks that are used to crack passwords and provides unique hashes for identical passwords. This means that users with the same passwords will have different values in the

database, since different salts will be used for each password. Furthermore, using a salt helps slow down brute force attacks since the attacker must crack each password individually due to the unique salts. As a result, incorporating a salt when hashing enhances the overall security of the database and allows user passwords to be stored securely.

While hashing and salting are common security features, there is a security concern around timing attacks being used to leak information about the hash. Timing attacks use the time taken to compare hashes to help crack passwords, but modern CPUs use many different safeguards against these types of attacks. Thus, this security concern is not particularly relevant. Another concern revolves around using general purpose hashing functions (e.g. SHA-256), which can leave systems more vulnerable to GPU accelerated attacks. However, InvenTracker's approach to this security concern is the most appropriate since it uses a password specific hashing function (i.e. bcrypt), which is better at mitigating GPU accelerated attacks due to the configurable work factor and built-in salting.

Access Control / Permissions

Since the application is used to monitor, update, and delete stock, it is imperative to incorporate access controls. Access controls are implemented to prevent unauthorised users from gaining access to sensitive data or tampering with the inventory system. For example, protections are added to sensitive routes, such as requests to update, add, and delete items by assigning different roles to specific users and actions. Furthermore, the backend of the application uses role based checks to protect the sensitive routes. The code uses a function that checks if a user's current role has the privileges required to access the route. Thus, if an actor gained access to an unprivileged user account, they would be unable to tamper with the inventory records or access sensitive data. Note that the main page cannot be accessed unless the user logs in with a valid username, password, and 2FA, which helps protect the application from unauthorised users.

A drawback associated with this approach is if the role checking function has a flaw, it can compromise the entire access control system. To address this issue, significant testing was undertaken to fix any bugs and patch vulnerabilities. Furthermore, it can be difficult to scale up this type of role based access control when there are

hundreds, or even thousands, of users. Instead, attribute based access controls could help solve this issue. However, we determined that this was unnecessary for the use case due to the limited number of roles.

A security concern related to this approach could be a lack of audit trails and logging making investigations into unauthorised access difficult. Detailed logs help determine the extent of an attack and what actions a malicious actor performed, both of which are necessary for proper analysis and to effectively mitigate future attacks. Nevertheless, the approach to access controls through role-based checking helps protect sensitive routes and sections of InvenTracker from unauthorised users.

Authentication

Since users have different roles and access permissions, authentication is essential. InvenTracker uses Sessions and Multi Factor Authentication (MFA) to implement authentication.

Sessions:

The application uses Node Express Session (NES), which is a lightweight framework that enables Express Session authentication middleware. The middleware checks user roles in the backend rather than riskily doing it on the frontend. NES allows the application to use session cookies to maintain the state between the client and server for authentication and session management. The cookies are configured to use the 'HTTPOnly' flag, increasing security since client-side JavaScript scripts are prevented from accessing the cookie. This mitigates the threat of Cross-Site Scripting attacks to read and steal session tokens. The application's cookies are also set to limit sessions to 1 hour, which helps reduce the risk of session hijacking. Another advantage of using Express Session is that, by default, it uses tokens with high entropy. This allows sessions to be resilient against brute force attacks.

However, despite the advantages, Node Express still has some shortcomings. Express Sessions rely on a single long-life Session ID. This means that for each API request, the frontend must send this single access token to the backend for authentication. This increases the risk of Man-In-The-Middle attacks to steal the Session ID since requests can expose this critical token. Additionally, NES does not

have any recourse against Cross-Site Request Forgery attacks (Podar, 2020).

Despite the limitations of NES, it is the most appropriate framework to use for authentication. It is very lightweight, fast to deploy, and popular in the industry despite offering basic security coverage. This makes Node Express an inferior choice for production applications, it is an excellent choice for the use case since it is not yet in production and is being run locally. During production, employing HTTPS would remove most of the security concerns. Configuring the secure cookie flag and .env to force HTTPS will redirect users to the HTTPS version of the website. Free services like 'Let's Encrypt' can be used to obtain SSI/TLS certificates, which help mitigate MITM attacks since the server must present a certificate to prove its identity. This certificate can then be checked against a Certificate Authority to verify the server's identity. Thus, employing HTTPS will help remove MITM attacks that attempt to steal Session IDs.

Two Factor Authentication:

In the contemporary digital landscape, protecting user accounts from unauthorised access is a critical challenge. CISA (2022) mentions how Multi Factor Authentication (MFA) lowers the likelihood of being hacked by 99%, particularly since attackers can now get past complex passwords. InvenTracker uses a two factor authentication (2FA) approach to achieve MFA by allowing a user to be approved with any combination of a known password, email access, and trusted device access.

When a valid username and password are entered, the system generates a unique, time sensitive code that is sent to the user's email address as shown in Figure 1. This code is generated using a cryptographically secure random function found in the crypto module mitigating random number generator prediction attacks. Since the MFA code must be entered, only individuals who have access to the email and know the password can login. Please note that the implemented method uses existing Gmail infrastructure, which is generally regarded as secure.

The image shows a simulated email interface. At the top left is a circular profile picture with a white 'V' on a blue background. To its right is a redacted email address, represented by a red bar followed by '@gmail.com'. Below this, the text 'to:' is followed by another redacted name, also represented by a red bar.

Your authentication code is: 394546

Do not share this code with anyone.

If you did not attempt to login, I would suggest changing your password as soon as possible.

Figure 1: An image showing the email one time code sent to a user.

When a user logs in, they can choose to remember their device for future logins. In doing so, a random 32-byte string is generated and given to the user as a cookie. Then, when the user next logs in, rather than being prompted for a one time code, the stored cookie will automatically complete the login process for the user. The creation time for this byte string is stored, however, and a cookie time limit of 30 days is imposed to prevent expired cookies from being used to gain access to the system.

InvenTracker focuses on security without compromising the user experience using a 2FA approach. Tran-Truong et al. (2025) generally defined authentication factors as: something a user is, something a user knows, and something a user has. The user's password falls into the "something a user knows" category, while the one time email code and device token are "something a user has". By addressing these two categories, the application increases security significantly. However, InvenTracker intentionally does not employ 3FA since this would impact the user experience too much and be inefficient to use on a regular basis for an inventory application.

2FA still has some security and privacy concerns. To apply 2FA, emails are stored and cannot be hashed since the values are directly required, and hashes of device tokens are stored. This risk of storing emails would have likely been an issue regardless, since emails may be sent to users to notify them about changes like low stock levels. Thus, the risk associated with storing emails that are linked to names would have been a privacy concern irrespective of 2FA. Hashed device tokens are largely safe to store, although computationally expensive attacks could find a user device token given a data breach or database compromise. Figure 2 demonstrates how the emails are attached to users in the database.

username	passw...	role	email
Filter	F	Filter	Filter...
user	\$2b\$10\$2IS1...	user	[redacted]@gmail.com
admin	\$2b\$10\$.h1LT...	admin	[redacted]@gmail.com
kienle	\$2b\$10\$.LNji...	admin	[redacted]@gmail.com

Figure 2: An image of the database showing user emails.

Since the disadvantages of using MFA (mainly regarding user experience) are significantly outweighed by the benefits, InvenTracker has taken the most appropriate approach by implementing 2FA. Whilst there are some privacy concerns surrounding 2FA, the risks are negligible with device tokens, or will likely be present without this security feature as in the case of emails. Leading technology companies, such as Microsoft, support email based MFA, thus showing how InvenTracker is following industry standards to security (Microsoft, 2016).

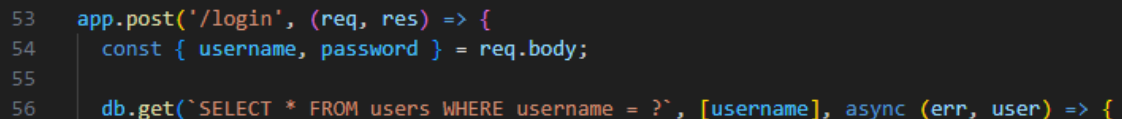
Handling User Responses

To handle user response, InvenTracker employs several approaches to data storage and transfer, and assumes that user inputs cannot be trusted. The application scrutinises user inputs using parameterised SQL queries in prepared statements, and enforces limited login attempts.

Parameterising Queries:

InvenTracker parameterises user inputs before placing them inside prepared SQL statements. This is because parameterised queries are one of the best methods of preventing SQL injection (OWASP, 2019). Parameterised queries within prepared statements help prevent attackers from changing the intent of a query (Engreitz, 2021). Values are bound to the query rather than being interpolated into an SQL string, preventing attackers from embedding SQL queries within their input. If an attacker maliciously entered a username like “tom or ‘1’=’1’”, then the parameterised query would look for a username that matches literally with the entire string. This safeguards the database from SQL injection. Prepared statements are simpler to write and understand than dynamic queries, while parameterised queries force

developers to define all SQL code before passing parameters. This still has potential vulnerabilities since state of the art software like PostgreSQL have found SQL injection vulnerabilities in sanitisation processes (PostgreSQL, 2025). Overall, the security benefits of preventing SQL injections far outweigh the extra developmental costs.



```
53   app.post('/login', (req, res) => {
54     const { username, password } = req.body;
55
56     db.get(`SELECT * FROM users WHERE username = ?`, [username], async (err, user) => {
```

Figure 3: An image showing an example of a Prepared Statement.

Limiting Login Attempts:

Limiting the number of login attempts for a potential user helps the system mitigate brute-force attacks (Lucas, 2025). In InvenTracker, user accounts are locked for 30 seconds after five incorrect attempts. This strategy helps slow down attacks, but a downside to this approach is that legitimate users who forget their password may become frustrated after they exhaust all attempts trying to log into their account. A commercial off-the-shelf application that also provides this functionality is 'Login LockDown'; although Login LockDown blocks IP addresses instead of accounts. But, Paixão Kröhling (2017) stated that blocking IP addresses cannot confidently identify a client and effectively stop an attack. Thus, the approach of limiting login attempts based on accounts is more appropriate.

Data Transfer:

To facilitate data transfer, the REST API is used. Login credentials, inventory stock data, and other updates are all sent using HTTP POST requests for client-server communication. The application uses Node Express to set a cookie on the user's browser after login and automatically send it back to the server to authenticate the session. Static files such as Cascading Style Sheets, frontend Javascript scripts, and EJS files are also provided using GET requests.

The advantage of using the REST API is that it is easy to implement and straightforward to understand. This makes it easy to maintain, while the widespread industry adoption of REST means that it is compatible across many different platforms. However, using a public API increases the attack surface, which can be a security concern. Another disadvantage of using REST is that repeated authentication

checks on every request can increase the load on the server. However, these trade-offs are worth it due to the increased application security.

Data Storage:

User and inventory data is stored on the backend using SQLite3. For SQL queries, parameterised inputs are utilised to protect against SQL injection, as shown by Figure 4. One of SQLite3's advantages is that it is lightweight and quick to set up. This makes it easy to maintain the code and helps lessen the load on the server. Inventory items are stored in an 'inventory' table in the database, while usernames and password hashes are stored in a 'user' table. Session data, including cookies, are stored in memory through the Express Session module. Also, sensitive environment variables and information (e.g. session secret keys) are stored securely in a .env file, rather than in the source code. This approach increases security by reducing the attack surface since actors cannot access sensitive data, even if the code is compromised.

```
355
356     // If valid and not duplicate, insert the item
357     db.run(
358         `INSERT INTO inventory (name, quantity, location, supplier) VALUES (?, ?, ?, ?)`,
359         [name, parsedQuantity, location, supplier],
360         (err) => {
361             if (err) return res.status(500).send('Error adding item');
362             res.redirect('/');
363         }
364     );
365 }
366 );
367 });
```

Figure 4: Example of Parameterised SQL Query

A major advantage of this approach is that the in-memory session storage can improve response times by reducing database queries. However, if the server process becomes compromised or crashes, all the active sessions could become exposed or lost entirely. Furthermore, there is no need for complex configurations or maintenance since the database is contained in a single file. However, high write workloads can cause database locking issues, which reduce the overall responsiveness of the application. SQLite3's lower overhead helps combat this disadvantage since it is efficient for applications with moderate read/write demands, like InvenTracker. In combination with the bcrypt hashing algorithm, the privacy concern of weak password storage is also mitigated. Thus, this approach is the

most appropriate for data storage since it balances functionality with privacy and security concerns.

Handling Potential Attacks

Mitigating Command Line Injection (CLI) Attacks:

Since no shell commands are used in the web application and no shell interaction exists in the source code, there is no direct command line injection attack vector available for attackers to use.

Mitigating Buffer Overflow Attacks:

InvenTracker executes JavaScript inside a managed environment in the browser and in Node.js on the server. Unlike other low-level languages like C and C++, JavaScript is relatively memory-safe because it uses automatic garbage collection and lacks direct memory access, thus reducing the risk of buffer overflow attacks. Furthermore, prepared and parameterised SQL queries help prevent SQL injection attacks by avoiding unsafe string concatenation in database queries. Additionally, limits on user inputted values are applied so that an attacker cannot set excessive values that crash the database.

A disadvantage with using JavaScript is that the automatic garbage collection can cause some latency, which negatively impacts performance. However, the potential latency that can be caused is negligible. JavaScript in Node.js is also single threaded, which can cause performance bottlenecks under high loads.

Mitigating Cross Site Scripting (XSS) Attacks:

InvenTracker mitigates XSS attacks using EJS templates. By default, EJS escapes any data inside the “<%= %>” tags. This involves converting special characters, including ‘<’ and ‘&’, into their HTML equivalents. Thus, malicious user inputs like “<script>alert(1)</script>” are displayed as plaintext rather than being executed as a script.

Another benefit of using EJS is that it is straightforward to use and integrates easily with Express, thus making it a good approach for the use case. Since EJS is one of the most widely used templates for Node.js, there is extensive documentation and widespread community support. This makes it easy to diagnose problems using open-source tutorials and implement plugins developed by the community.

However, a security concern related to EJS is that if `<%- %>` is accidentally used instead of `<%= %>` when developing the code, then a direct XSS vulnerability is created. This is because `<%- %>` renders whatever content is provided, even malicious scripts, whereas `<%= %>` automatically escapes malicious characters. To address this security concern, all the code in InvenTracker was checked to ensure only `<%= %>` is used.

Conclusion

In summary, the approach to hashing and salting, access control, authentication, handling user responses, and mitigating potential attacks is the most appropriate. The application collects, stores, and transfers data using technologies and techniques that are widely used in the industry by leading companies. Furthermore, likely types of attacks are mitigated, sometimes using multiple methods, thereby enhancing overall security. Ultimately, InvenTracker balances functionality with privacy and security concerns, all while ensuring that leading security practices are followed.

References

Cybersecurity and Infrastructure Security Agency. (2022). Multifactor Authentication (MFA). *Cybersecurity and Infrastructure Security Agency*.

<https://www.cisa.gov/topics/cybersecurity-best-practices/multifactor-authentication>

Engreitz, J. (2021). Parameterized Queries. *Hex*.

<https://hex.tech/templates/parameterized-query/>

Gupta, D. (2024). Comparative Analysis of Password Hashing Algorithms: Argon2, bcrypt, scrypt, and PBKDF2. *Deepak Gupta*.

<https://guptadeepak.com/comparative-analysis-of-password-hashing-algorithms-argon2-bcrypt-scrypt-and-pbkdf2/>

Hughes, A. (2024). Encryption vs. Hashing vs. Salting - What's the Difference?. *PingIdentity*.

<https://www.pingidentity.com/en/resources/blog/post/encryption-vs-hashing-vs-salting.html>

Lucas, S. (2025). Login Lockdown – Limit Login Attempts & Prevent Brute Force Attacks. *Auto Page Rank*.

<https://autopagerank.com/login-lockdown-limit-login-attempts/>

Microsoft. (2016). How to use two-step verification with your Microsoft account. *Microsoft*.

<https://support.microsoft.com/en-au/account-billing/how-to-use-two-step-verification-with-your-microsoft-account-c7910146-672f-01e9-50a0-93b4585e7eb4>

OWASP. (2019). SQL Injection Prevention Cheat Sheet. *OWASP Cheat Sheet Series*.

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

Paixão Kröhling, J. (2017). Why IP-based rules are bad, but we still use it. *Medium*.

<https://medium.com/qaclana/why-ip-based-rules-are-bad-but-we-still-use-it-99b4f356bc83>

Podar, R. (2020). Should you use Express-session for your production app?. *SuperTokens*.

<https://supertokens.com/blog/should-you-use-express-session-for-your-production-app>

PostgreSQL. (2025). CVE-2025-1094. *PostgreSQL*.

<https://www.postgresql.org/support/security/CVE-2025-1094/>

Tran-Truong, P. T., Pham, M. Q., Son, H. X., Nguyen, D. L. T., Nguyen, M. B., Tran, K. L., Van, L. C. P., Le, K. T., Vo, K. H., Kim, N. N. T., Nguyen, T. M., & Nguyen, A. T. (2025). A systematic review of multi-factor authentication in digital payment systems: NIST standards alignment and industry implementation analysis. *Journal of Systems Architecture*, 162, 103402-.

<https://doi.org/10.1016/j.sysarc.2025.103402>

Vaideeswaran, N. (2024). Hashing in Cybersecurity. *CrowdStrike*.

<https://www.crowdstrike.com/en-us/cybersecurity-101/data-protection/data-hashing/>