Hedera Transaction Tool Code review

Hedera Hashgraph
Version 1.0 – November 20, 2024

Prepared By
Angel Roman
Victor Lasa

Prepared For
Joe Blanchard
Simi Hunjan

1 Executive Summary

Synopsis

During the month of October 2024, Hedera Hashgraph engaged NCC Group to conduct a security code audit of Hedera Transaction Tool. The application is a demo application that allows a user to generate keys, create, sign, and submit transactions to a Hedera network. This software is designed for use solely by the Hedera Council and staff. More detailed information can be seen on the Hedera Transaction Tool Overview.

Scope

NCC Group's evaluation included:

- **Hedera Transaction Tool**: Code Review was performed on the commit id `a08e595924d80558258a2c2c2df8906a80e93ef0` of the project hosted on https://github.com/hashgraph/hedera-transaction-tool. A branch of the code was made at this point in the following branch: https://github.com/hashgraph/hedera-transaction-tool/tree/ncc-external-code-audit

Key Findings

The assessment uncovered a set of common application flaws. The most notable findings were:

- Weak Password Complexity Requirements: The only requirement when first setting up the tool was to provide a 10-character password. Furthermore, users could change their password once they were logged into their account to any other password with no restrictions. As this password was used to encrypt the private keys stored by the application, a weak password could allow attackers with local access to compromise the private keys stored in the application's database.
- Secrets Committed to Repository: Inspection of the source code revealed that sensitive secrets were stored in configuration files and inadvertently committed to the git repository. This exposure allows anyone with access to the affected source code repositories to potentially exploit these secrets, gaining unauthorized control over the associated services and environments.
- Weak Default Secrets in Config Files: Default, weak secrets, including the secrets to sign JWT tokens, were stored in the configuration files and were not documented, allowing attackers to forge authentication tokens and compromise any user account and role if these default secrets were not changed during deployment.

Strategic Recommendations

Deprecate Exposed Secrets: Revoke all exposed secrets to prevent unauthorized access and potential security breaches. Implement robust secret management practices moving forward, including the use of secure vaults or secret managers, to safeguard against future exposures. Before making the repository public, remove any sensitive credential from the entire git repository history.

Hardening Guides: It is recommended to develop comprehensive security hardening guides linked in the `READMEs` to provide users deploying the application with best practices for secure configuration.

Setup Wizard: To address the findings related to default configurations, and to ease of deployment for the backend with secure options, consideration should be given to implement an interactive setup wizard through scripts or through a web interface, guiding users through the configuration process, prompting for necessary secrets and settings while integrating with secret management tools. This would ensure that secrets are securely

stored and managed. Additionally, enforce validation checks to meet security best practices, and ensure that the configuration options cannot be run after the setup phase is finished.

On-Chain Multi-Sig: To mitigate the risks of private key compromise on the client side, it is recommended to configure the accounts that can perform sensitive operations with on-chain multi-signature, by leveraging Hedera's native capabilities, such as the Threshold Keys, that allow to set up thresholds of required keys to sign a transaction in the account, eliminating single points of failure if a private key is compromised.

Memory Leaks Mitigations: To mitigate the threat of private key leakage in the case of an attacker accessing the memory of the client laptop with the application open, it is recommended to nullify sensitive strings after use and, where possible, use buffers or byte arrays over strings, allowing manual zeroing of the secrets after use. Also, secure libraries like sodium-native provide functions for enhanced memory management. It is also recommended to consider Trusted Execution Environments (TEE) like Intel SGX, that would allow isolating the memory of the sensitive data even from system-level access, which is considered the state of the art for private key management.

Hardware Wallets Support: Consider supporting Hardware Wallets for signing, which would increase the security by storing the private keys outside the application entirely.

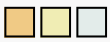
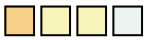
The code base should undergo an additional round of review once it reaches Release Candidate status. Additionally, any new functionality should be individually assessed during development once implemented.

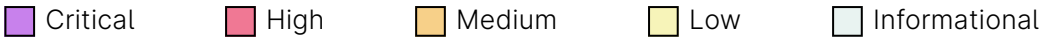
2 Dashboard

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	3	
Low issues	5	
Informational issues	4	
Total issues	**12**	

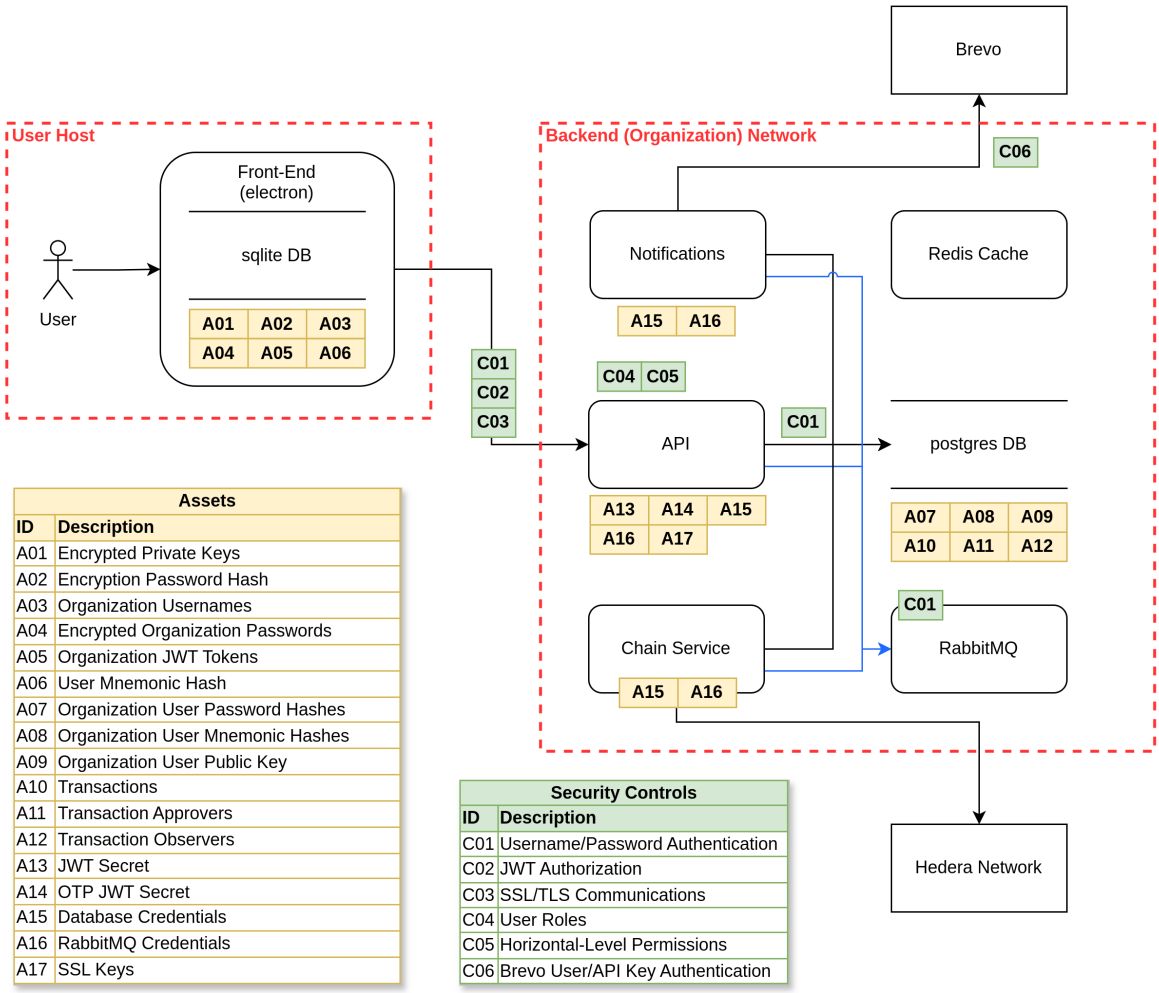
Category Breakdown

Access Controls	2	
Auditing and Logging	1	
Authentication	1	
Cryptography	3	
Data Exposure	4	
Other	1	



3 Hedera Transaction Tool Overview

NCC Group created the following diagram depicting a simplified view of the logic components, overlaying trust boundaries, assets and security controls. This diagram is loosely based on recognized conventions such as those described in The Security Development Lifecycle, Chapter 9, (M. Howard and S. Lipner, 2006). Other conventions may have been used based on project-specific requirements.



Assets	
ID	Description
A01	Encrypted Private Keys
A02	Encryption Password Hash
A03	Organization Usernames
A04	Encrypted Organization Passwords
A05	Organization JWT Tokens
A06	User Mnemonic Hash
A07	Organization User Password Hashes
A08	Organization User Mnemonic Hashes
A09	Organization User Public Key
A10	Transactions
A11	Transaction Approvers
A12	Transaction Observers
A13	JWT Secret
A14	OTP JWT Secret
A15	Database Credentials
A16	RabbitMQ Credentials
A17	SSL Keys

Security Controls	
ID	Description
C01	Username/Password Authentication
C02	JWT Authorization
C03	SSL/TLS Communications
C04	User Roles
C05	Horizontal-Level Permissions
C06	Brevo User/API Key Authentication

Figure 1: Hedera Transaction Tool Diagram

As described in the README, the Hedera Transaction Tool application is a demo application that allows a user to generate keys, create, sign, and submit transactions to a Hedera network. This software is designed for use solely by the Hedera Council and staff. The software is being released as open source example code only, and is not intended or suitable for use in its current form by anyone other than members of the Hedera Council and Hedera personnel.

The application was composed by the following components:

- **Front-End**. Client application using Electron, that provided the user of the application with all the interfaces to use the application. The client application had a SQLite database where the private keys added by the user were saved (encrypted with a password set by the user). This database also held the encryption password hash, as well as credentials for external organizations (backends).
- **Back-End**. Server-side application that allowed connecting different users in order to create transactions that needed to be signed by other users, while also allowing the user

to define approval rules, requiring the signature of configurable number of users to submit a signed transaction. The approval process was performed off-chain, without relying on any of the native Hedera account abstraction capabilities.[1]

This component was divided between 3 microservices, all connected with RabbitMQ messages:

- API: Defined the API endpoints and the main logic of the backend
- Notifications: In charge of email notifications via a SMTP connection to Brevo
- Chain: In charge of all the connections to the Hedera network.

This component also had a Postgres database, where the organization users and password hashes were saved, as well as their public keys, required to verify their signatures for the validity of the created or signed transactions, as well as the transaction history of the users along with their state.

The backend defined roles (admin and normal user) and presented horizontal-level controls to control the visibility of pending transactions (observers), as well as who had permission to approve a transaction (approvers), or who could edit transaction details (owner).

The user sessions in the application were managed by JWT tokens.

1. Hedera Account Abstraction at the Protocol Level https://hedera.com/blog/account-abstraction-at-the-protocol-level

4 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Secrets Committed to Repository	New	AFA	Medium
Weak Password Complexity Requirements	New	UVW	Medium
Weak Default Secrets in Config Files	New	XRW	Medium
Code Comments Suggest Incomplete or Missing Code	New	2VC	Low
Exposed Mnemonic Hashes	New	F9D	Low
Excessive Default JWT Token Lifetime	New	KBG	Low
Insecure Organization Credentials Storage	New	FCD	Low
Insecure Communications With Message Broker	New	RXH	Low
Empty or Missing Catch Statement	New	HEN	Info
Sequential IDs Used	New	WQ3	Info
Weaknesses in Private Keys Encryption and Password Hashing Practices	New	Q47	Info
Reusable Password Reset Tokens	New	DKK	Info

5 Finding Details

Medium

Secrets Committed to Repository

Overall Risk	Medium	**Finding ID**	NCC-E017270-AFA
Impact	Medium	**Component**	Code Review
Exploitability	High	**Category**	Data Exposure
		Status	New

Impact

Attackers with access to the source code would be able to extract valid API keys for Brevo, that could be used to perform a denial of service on the Brevo platform by sending an unlimited number of emails, or to perform targeted phishing attacks using the email address configured of the application. Other secrets disclosed user emails and potentially valid passwords.

Description

Inspection of the code revealed that secrets were stored in configuration files. The presence of these secrets means that anyone with access to the affected source code repositories could potentially gain control over the affected services and environments by using those secrets.

Furthermore, propagation of such secrets to source control places a significant security burden on the development and build of infrastructure, making the insider threat scenario almost impossible to mitigate.

Below are some examples of secrets discovered in the configuration files:

```
apiVersion: v1
kind: Secret
metadata:
  name: brevo-secret
  namespace: hedera-transaction-tool
type: Opaque
data:
  username: cG[REDACTED]b20=
  password: eH[REDACTED]FJ5
```

Figure 2: back-end/k8s/back-end/brevo-secret.yaml

The secrets above were confirmed as a valid Brevo username and API key, that allowed to send emails with `no-reply@hederatransactiontool.com`.

As a proof of concept, the email below was received with these keys:

```
[...]
Subject: Test Email from Brevo SMTP
Origin-messageId: <cafb775d-1d6d-b740-aab1-d16523fe9a6a@hederatransactiontool.com>
To: <test@[REDACTED]>
[...]
From: "Transaction Tool" <no-reply@hederatransactiontool.com>
[...]
```


Other examples were found in the files with extension ".http" that contained examples of requests, disclosing valid emails and potentially valid passwords:

```
### Log in Ted
[...]
POST http://localhost:3001/auth/login
content-type: application/json

{
  "email": "[REDACTED]@outlook.com",
  "password": "NI[REDACTED]z5"
}
[...]
```

Figure 3: back-end/apps/api/src/auth/auth-requests.http

```
[...]
POST http://localhost:3000/users/reset-password
content-type: application/json

{
  "email": "[REDACTED]@swirldslabs.com"
}

### Verify reset password for current temp user
POST http://localhost:3000/users/verify-reset
content-type: application/json

{
  "email": "[REDACTED]@swirldslabs.com",
  "token": "12147954"
}
[...]
```

Figure 4: back-end/apps/api/src/users/user-requests.http

Other potential secrets could end up as default weak secrets (see finding "Weak Default Secrets in Config Files"), as shown in the example below:

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
  namespace: hedera-transaction-tool
type: Opaque
data:
  database: cG9zdGdyZXM=
  username: cG9zdGdyZXM=
  password: cG9zdGdyZXM=
```

Figure 5: hedera-transaction-tool-main/back-end/k8s/back-end/postgres-secret.yaml

This secret was a base64-encoded string:

```
$ base64 -d <<< 'cG9zdGdyZXM='
postgres
```


Recommendation

Secrets should not be stored in code repositories. Instead, they should be stored separately, ideally in a password vault or storage mechanism designed for the purpose. The passwords currently stored in the code should be considered compromised and should be rotated as they are removed from the code base and transferred into the appropriate secret management platform.[2] [3]

It is recommended to remove the sensitive data from the repository before it is made public, altering the repository's history with tools like `git filter-repo` or the BFG Repo-Cleaner. More information can be found in the Github Documentation.

It would be prudent to integrate a static analysis process into the CI/CD pipeline or development process to ensure that credentials are kept out of code in the future. Any secrets that are committed to source and detected should be eliminated from the source should be rotated, as source control servers will retain the historical data.

It is also recommended to add the following lines to the .gitignore file to avoid committing files with secrets:

```
*-secret.yaml
*.http
```

Location

- /back-end/k8s/back-end/jwt-secret.yaml
- /back-end/k8s/back-end/rabbitmq-secret.yaml
- /back-end/k8s/back-end/otp-secret.yaml
- /back-end/k8s/back-end/postgres-secret.yaml
- /back-end/k8s/back-end/brevo-secret.yaml
- /back-end/apps/api/src/user-keys/user-keys-requests.http
- /back-end/apps/api/src/auth/auth-requests.http
- /back-end/apps/api/src/transactions/approvers/approvers-requests.http
- /back-end/apps/api/src/transactions/comments/comments-requests.http
- /back-end/apps/api/src/transactions/observers/observers-requests.http
- /back-end/apps/api/src/transactions/transactions-requests.http
- /back-end/apps/api/src/transactions/groups/transaction-groups-requests.http
- /back-end/apps/api/src/transactions/signers/signers-requests.http
- /back-end/apps/api/src/users/user-requests.http

2. SANS Top 25 Software Flaws: Number 11, Hardcoded Credentials: https://www.sans.org/blog/top-25-series-rank-11-hardcoded-credentials/
3. CWE-798: Use of Hard-Coded Credentials: https://cwe.mitre.org/data/definitions/798.html

Medium

Weak Password Complexity Requirements

Overall Risk	Medium	**Finding ID**	NCC-E017270-UVW
Impact	High	**Component**	Code Review
Exploitability	Medium	**Category**	Authentication
		Status	New

Impact

An attacker may guess or brute-force weak user passwords, especially in the event of a password database breach. These passwords were used to encrypt private keys, increasing the impact were this issue to be exploited.

Description

The Hedera Transaction Tool application did not enforce enough restrictions on user passwords. The only requirement when first setting up the tool was to provide a 10-character password without any other restriction such as using several character classes. Furthermore, users could change their password once they were logged in to their account to any other password with no restrictions whatsoever.

As a result of the observed restrictions, it was possible for users to set their passwords to simple values such as “1234567890” when first creating their account, and later changing it to a single character (e.g. “a”).

If a user does use a weak password, it is more likely that an attacker could guess their password and gain access to their account. Alternatively, in the event of a password database breach, an attacker is more likely to recover a weak password from a brute-force attack.

This password was used to encrypt private keys, which were stored in the application’s local database:

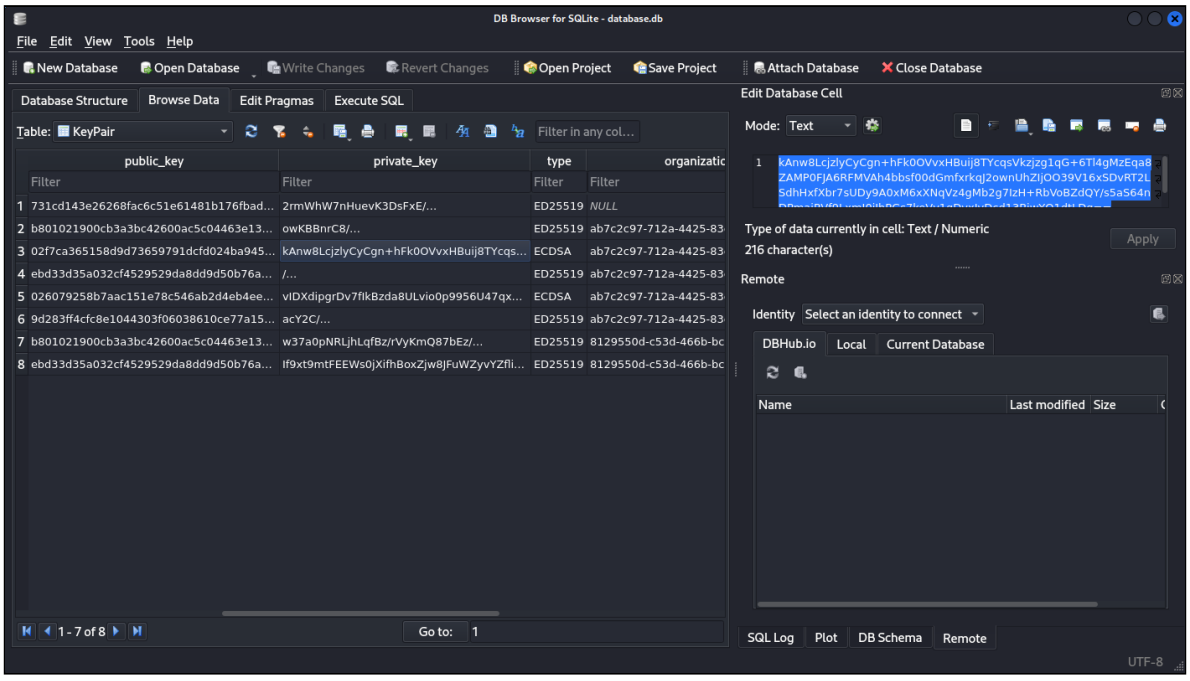


Figure 6: ~/.config/hedera-transaction-tool/database.db

This meant that private keys were as secure as the user's password. If the user's password was weak, decrypting their private keys would be trivial, as shown in the example below:

```
$ node decrypt.js
Usage: node decrypt.js <base64_encrypted_data> <password>

$ node decrypt.js 'kAnw8LcjzlyCyCgn+hFk0OVvxHBuij8TYcqsVkzjzg1qG+6Tl4gMzEqa8ZAMP0FJA6RFMVAh4bbs
↳ f00dGmfxrkqJ2ownUhZIjOO39V16xSDvRT2LSdhHxfXbr7sUDy9A0xM6xXNqVz4gMb2g7IzH+RbVoBZdQY/
↳ s5aS64nDPmaiRVf9LxmJ0jJhPGs7keVv1gDvxIvDcd13BiwYQ1dtLDg==' 'a'
Decrypted Data:
  859f21e0846dab501af11c16f56bf1043e2eac4d5a7a55d05a014d3a6b4d03be
```

It can be observed that the decrypted private key above matched the one from the Hedera Transaction Tool:

Figure 7: Private Key

The code for the `decrypt.js` script can be found in the Supplemental Data - Decrypt Script section at the end of this document.

This script was created using the parameters set in the `/automation/utils/crypto.js` file, which contained the functions used to encrypt private keys, as mentioned in finding "Weaknesses in Private Keys Encryption and Password Hashing Practices".

Recommendation

Ideally, complexity checks should be performed on the client-side using a tool like `zxcvbn`[4] to provide immediate user feedback. Subsequently, these checks should be repeated on the server-side to ensure that the password rules have not been bypassed on the client side and to prevent the reuse of previous credentials.

Do not require users to regularly update passwords at unreasonably short intervals, as this increases the likelihood of weaker passwords overall.[5]

Finally, consider providing users with the option to use multi-factor authentication for all applications.

Location

- /front-end/src/main/services/localUser/auth.ts
- /front-end/src/main/services/localUser/keyPairs.ts

4. Low-Budget Password Strength Estimation: https://github.com/dropbox/zxcvbn
5. NCSC Password policy: updating your approach https://www.ncsc.gov.uk/collection/passwords/updating-your-approach

Medium

Weak Default Secrets in Config Files

Overall Risk	Medium	**Finding ID**	NCC-E017270-XRW
Impact	High	**Component**	Code Review
Exploitability	Low	**Category**	Cryptography
		Status	New

Impact

If the JWT secret is not changed or a weak secret is used, an attacker could create JWTs with arbitrary payloads that pass verification, leading to unauthorized access and potential data breaches. Several other sensitive secrets with weak default values were also identified.

Description

The default secret used for the JSON web signature (JWS) that protected the JSON web tokens (JWTs) for the Hedera Transaction Tool had a default value hardcoded in the configuration files. As a result, it would be possible to create JWTs with an arbitrary payload that passed verification in any instance that had not changed this secret.

An attacker in possession of the HMAC shared key used to sign the JWT can create tokens with an arbitrary payload. As such the HMAC secret needs to be particularly strong and, since it is used programmatically and thus does not need to be regularly entered like a traditional password, the value can be long and complex without usability consequences.

Note that if the implementation uses a JWT library that includes a default HMAC secret, its value must be changed irrespective of the strength of the original.

The HMAC shared key had a default value of `<enter any string>`, which was a valid secret, that if unchanged, was used to sign the JWT tokens. This was found in two instances within the same `.env` file in the GitHub repository:

```
# JSON web token settings
JWT_SECRET=<enter any string>
# Temporary, expiration in days
JWT_EXPIRATION=365

# One time password settings
OTP_SECRET=<enter any string>
# OTP expiration in minutes
OTP_EXPIRATION=20
```

Figure 8: /back-end/apps/api/example.env

As shown in the snippet above, the OTP secret was also set to the same default value. This secret was used to sign the JWT when issuing a password change request.

It is worth noting that there were no mentions found to changing these secrets in the documentation apart from copying the example `.env` file before running the project, making it more likely that organizations deploying the Hedera Transaction Tool have default, insecure JWT and OTP signing tokens.

Similarly, several default weak secrets were found in the Kubernetes configuration. A couple of examples are presented below:

```
apiVersion: v1
kind: Secret
metadata:
  name: jwt-secret
  namespace: hedera-transaction-tool
type: Opaque
data:
  key: MTIzdGVzdDQ1NnRlc3Q3ODl0ZXN0
```

Figure 9: /back-end/k8s/back-end/jwt-secret.yaml

This secret was the following base64-encoded string: `123test456test789test`.

```
apiVersion: v1
kind: Secret
metadata:
  name: rabbitmq-secret
  namespace: hedera-transaction-tool
type: Opaque
data:
  RABBITMQ_USERNAME: dG9vbC1iYWNrLWVuZA==
  RABBITMQ_PASSWORD: cGxhaW5wYXNz
  RABBITMQ_ERLANG_COOKIE: c2ltcGxlX2Nvb2tpZV9zdHJpbmc=
```

These secrets were the following base64-encoded strings, respectively: `tool-back-end`, `plainpass` and `simple_cookie_string`.

There were other instances where secrets were considered more secure, yet they were still exposed, as mentioned in finding "Secrets Committed to Repository".

Recommendation

Eliminate any hard-coded secrets from configuration files and ensure secrets are not included in the source code, requiring a unique, strong secret for the HMAC signature to be generated for each deployment, that conforms to RFC 7518.

It is preferable to include the full range of byte values for the secret (in other words, it should not be limited to printable characters) to make full use of the available key space.[6] [7]

It is recommended the use of RS256 instead of HS256. RS256 uses asymmetric encryption (public/private key pair), which, in case of extending the application to a microservices' architecture, would allow securing the private key with access only to the authentication service, and would allow distributing the public key between the rest of the services, allowing them to verify tokens without access to the private key, facilitating better scalability in distributed systems.

These parameters and their importance should be clearly mentioned in the application's documentation or, better yet, automatically generated on deployment.

Use secret management tools to securely manage and inject JWT and OTP secrets into the application.

Implement a policy for regular rotation of JWT and OTP signing secrets to minimize the impact of potential secret exposure.

6. JWA RFC 7518: https://tools.ietf.org/html/rfc7518#section-3.2
7. IETF JWT Best Current Practices: https://datatracker.ietf.org/doc/html/rfc7518

Location

- /back-end/scripts/example.env
- /back-end/apps/api/example.env
- /back-end/apps/chain/example.env
- /back-end/apps/notifications/example.env
- /back-end/k8s/back-end/brevo-secret.yaml
- /back-end/k8s/back-end/jwt-secret.yaml
- /back-end/k8s/back-end/otp-secret.yaml
- /back-end/k8s/back-end/postgres-secret.yaml
- /back-end/k8s/back-end/rabbitmq-secret.yaml

Low

Code Comments Suggest Incomplete or Missing Code

Overall Risk	Low	**Finding ID**	NCC-E017270-2VC
Impact	Low	**Component**	Code Review
Exploitability	Low	**Category**	Other
		Status	New

Impact

Code comments that indicate that functionality is incomplete or broken may lead to scenarios where security critical areas, that may prevent a vulnerability or improve the security of the solution, are left missing and otherwise forgotten about.

Description

A search for code comments with the terms “FIXME” or “TODO” and derivatives thereof identified a number of instances where developers have noted incomplete or missing code. Such comments usually indicate that further work is needed to implement a feature or, more often, to fix bugs and introduce error checking. These code changes or additions are often not made, and these sections of code can lead to erroneous or vulnerable behavior.

/back-end/apps/api/src/auth/auth.service.ts

```
/* Log the user out of the organization, remove his cooke and blacklists his token */
  logout(response: Response) {
    response.clearCookie('Authentication', {
      httpOnly: true,
      sameSite: ['production', 'testing'].includes(this.configService.get('NODE_ENV'))
        ? 'none'
        : 'lax',
      secure: ['production', 'testing'].includes(this.configService.get('NODE_ENV')),
    });
    //TODO implement token blacklisting
  }
```

/back-end/apps/api/src/transactions/comments/comments.controller.ts

This API functionality did not seem to be in use by the application.

```
@UseGuards(JwtAuthGuard, VerifiedUserGuard)
//TODO add serializer
export class CommentsController {
  constructor(private commentsService: CommentsService) {}

  @Post()
  //TODO need some sort of guard or check to ensure user can comment here
  createComment(
    [SNIPPED]
  ) {
    return this.commentsService.createComment(user, transactionId, dto);
  }
[SNIPPED]

  //TODO add update and remove routes
```


/back-end/libs/common/src/models/transfer-transaction.model.ts

```
// Get the Fee Payer
    const accounts = super.getSigningAccounts();

    // Go through all the transfers and add the accounts to the array.
    // All accounts are included, whether it is a receiving account that
    // requires a signature or not. This is because the 'signature required'
    // flag can change at any time, and this list of accounts could then be wrong
    // if stored (as it is in the transaction datatable). So all accounts are
    // included.
    //TODO if all accounts are included, how do we differentiate between those
    // that need to be sign and not? Meaning if I am on account 44, and account 44
    // is NOT flagged to require sig for receiving, when I pull transactions for account 44
    // I will pull any transaction that 44 is the receiver, and it will be displayed on the
    ↪ client
    // indicating I need to sign it. That is not right. So, then what? When the user signs in
    // they will pull accountIds for their keys? or do they pull full account info? if account
    ↪ info, easy
    // if not, I'd have to do another check?
    // Also, if I am logged in already, then someone flips the flag, how would I get notified
    ↪ of the
    // transactions I now need to sign? In order to flip the flag, an account update needs to
    ↪ go through.
    // And if that is done outside of this app, ugh. But it really shouldn't be able to be
    ↪ done outside of a
    // multi sig solution. So let's say it has to be done inside the app. But let's also say
    ↪ that I didn't sign
    // the account update, but it passed due to other sigs. Someone somewhere would then need
    ↪ to pull/push the
    // new transactions. If the server pushes, it would need to go through each transaction
    ↪ for the updated account
    // and find all keys on that transaction for that account and push the transaction. Or
    ↪ could it just
    // get the users for the keys on the updated account and notify the user to pull new data?
```

Recommendation

Review all instances of FIXME, TODO, and other such comments identified above, and implement the necessary code changes and additions in order to improve the overall robustness of the application.[8]

Where changes are not able to be implemented in a timely manner, said missing or broken functionality should be formally tracked within the development team's issue/bug tracking system.

Consider using static analysis to track when such comments are added to code to prevent their usage and ensure they are captured in the development backlog instead.

Location

- /back-end/apps/api/src/auth/auth.service.ts
- /back-end/apps/api/src/transactions/comments/comments.controller.ts
- /back-end/libs/common/src/models/transfer-transaction.model.ts

8. Todo Comments Considered Harmful: https://c2.com/cgi/wiki?TodoCommentsConsideredHarmful

Low

Exposed Mnemonic Hashes

Overall Risk	Low	**Finding ID**	NCC-E017270-F9D
Impact	Low	**Component**	Code Review
Exploitability	Medium	**Category**	Data Exposure
		Status	New

Impact

Exposing hashed mnemonic passphrases could allow attackers to perform offline brute-force attacks, potentially compromising private keys and user accounts. It is acknowledged that brute forcing a 24 words mnemonic passphrase securely generated is an impractical attack, except in cases where the mnemonic may have been generated insecurely. For this reason, the overall risk and impact were set to Low.

Description

The Hedera Transaction Tool was found to save both in client and server side, and then expose to all users, hashes of the mnemonic passphrases used to generate or import private keys.

Although the hash itself did not reveal the original mnemonic, an attacker with access to the hash could attempt offline brute-force or dictionary attacks to recover the mnemonic, compromising the security of the associated account.

These hashes were being returned for each user when issuing a GET request to the `/user/*/keys` endpoint:


```
[
  {
    "id": 3,
    "userId": 7,
    "mnemonicHash": "$2b$10$byOLuN.OFoFbSjnsTfkc/.FRIKqwzSxbPbL1IuHPYoUViglR0v4QS",
    "index": 0,
    "publicKey": "b801021900cb3a3bc42600ac5c04463e13f4b46005acce60ba7cb9971206fc09",
    "deletedAt": null
  },
  {
    "id": 5,
    "userId": 7,
    "mnemonicHash": "$2b$10$bM..nrwNsQRXEvrWx6bQiexL1E2YXMRKlY69zmwNjFpUQ7SHrJrF2",
    "index": 1,
    "publicKey": "ebd33d35a032cf4529529da8dd9d50b76a9a799e35cb3cb04bfd827ecc319281",
    "deletedAt": null
  }
]
```


These hashes were stored both by the frontend, in the local SQLite database on the `KeyPair` table, `secret_hash` field; and on the backend database, in the `user_key` table, `mnemonicHash` field. They were hashed by bcrypt with 10 iterations which, while not considered weak, is considered the minimum when hashing a password (see finding "Weaknesses in Private Keys Encryption and Password Hashing Practices").

As an example, the hash `$2b$10$sdgBu8AuKlBatsCXQw8.ver89g2GgMoE3SbAIdUP7t.0YzdsdeiXm` correspond to the join of the mnemonic words with a comma:

`lift,nest,armed,program,leader,march,message,place,similar,job,forest,decline,buzz,speak,spray,stairs,hire,heart,pulse,wife,brisk,face,future,scare`

Additionally, as highlighted in finding "Sequential IDs Used", user identifiers were sequential integers, further increasing the risk of enumeration attacks.

Recommendation

Consider not storing the mnemonic hashes on the client and server side, as the public key of the first account should be enough to make any functionality to recover the data associated to an account. Mnemonic hashes from API responses should be removed too. There is no legitimate need for the application to return mnemonic hashes to the end user. This data should be never exposed via public endpoints.

Location

- /automation/pages/OrganizationPage.js, `generateAndStoreUserKey` function
- /automation/utils/databaseQueries.js, `insertUserKey` function
- /back-end/apps/api/src/user-keys/user-keys.service.ts

Low

Excessive Default JWT Token Lifetime

Overall Risk	Low	**Finding ID**	NCC-E017270-KBG
Impact	Medium	**Component**	Code Review
Exploitability	Low	**Category**	Access Controls
		Status	New

Impact

An attacker who steals a token will maintain unauthorized access for an extended period.

Description

The application uses long-lived JWT tokens by default when accessing an organization. Best practice is to regularly rotate session tokens over short time periods to reduce the risk of compromise. For instance, if a session token is accidentally saved to a backup, committed in source code, or shared through insecure channels like chat or email, regular rotation ensures that any exposed session token is limited in its usefulness to an attacker. The default expiration time set in `back-end/apps/api/.env` was 365 days (1 year):

```
# Temporary, expiration in days
JWT_EXPIRATION=365
```

Since the README and documentation did not recommend changing this value, it is likely that users of the application will leave it unchanged.

As the validity of a JWT is generally not revoked on demand but relies on the expiry time which is set as a built-in property of the token, a relatively short time frame is recommended.

Recommendation

Set the expiry period of the JWT to as short a period as possible, appropriate to the use case.[9] For example, a session timeout of 12 hours ensure that the token is valid through the day, but requires to log in again the next day. If necessary, use a token refresh mechanism to extend access when a token approaches expiry if long-lived tokens are needed.

Location

- /back-end/apps/api/.env

9. RFC – JSON Web Token (JWT) – Registered Claim Names: https://tools.ietf.org/html/rfc7519#section-4.1

Low

Insecure Organization Credentials Storage

Overall Risk	Low	**Finding ID**	NCC-E017270-FCD
Impact	Medium	**Component**	Code Review
Exploitability	Low	**Category**	Data Exposure
		Status	New

Impact

Storing user credentials on the client side increases the risk of credential exposure. Failing to remove these credentials upon logout could allow unauthorized access when reopening the application, compromising security and contradicting user expectations of a complete logout.

Description

The application's frontend stored usernames and encrypted passwords for each organization in a local database, which is against best practices on token management. Passwords were encrypted using the user's password (see finding "Weak Password Complexity Requirements"), in a similar way to the private keys.

The screenshot below shows the encrypted credentials in the local database:

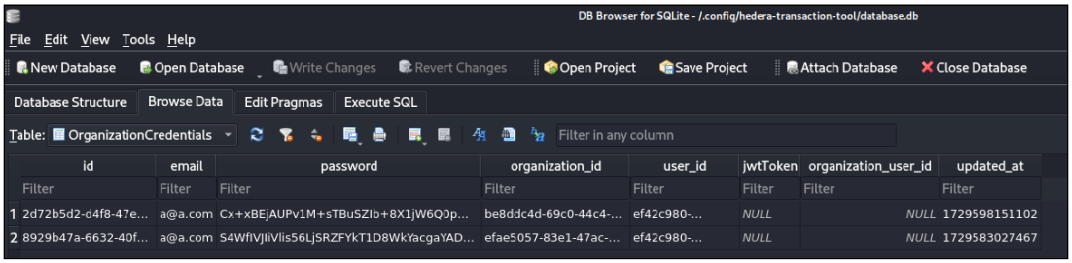


Figure 10: User and encrypted password stored in frontend database

The snippet below was in charge of storing these credentials in the database:

```
export const addOrganizationCredentials = async (

[...]
    await prisma.organizationCredentials.create({
      data: {
        email,
        password,
        organization_id,
        user_id,
      },
    });
[...]
};

export const updateOrganizationCredentials = async (
[...]
) => {
  const prisma = getPrismaClient();

  try {
    if (password) {
```

```
      const useKeychain = await getUseKeychainClaim();
      if (useKeychain) {
        const buffer = safeStorage.encryptString(password);
        password = buffer.toString('base64');
      } else if (encryptPassword) {
        password = encrypt(password, encryptPassword);
      } else {
        throw new Error('Password is required to store unencrypted key pair');
      }
    }

    const credentials = await prisma.organizationCredentials.findFirst({
      where: { user_id, organization_id },
    });
[...]
    await prisma.organizationCredentials.update({
      where: { id: credentials.id },
      data: {
        email: email || credentials.email,
        password: password || credentials.password,
      },
    });
[...]
};
```

Figure 11: /front-end/src/main/services/localUser/organizationCredentials.ts

Furthermore, when users logged out of an organization, the associated credentials were not removed from the database. This resulted in unintended behavior: if the application was closed and then reopened, users were automatically logged back into the last organization accessed upon providing the encryption password.

Users may not be aware that their credentials persist, creating a potential risk if another person were to gain access to the device. Additionally, the current design undermined typical logout expectations by failing to fully terminate the user session.

Recommendation

The user password should never be stored on the application, even encrypted. Instead, an encrypted JWT token should be saved instead.

Once a user logs out from an organization, the process should remove all user credentials/ JWT tokens as well as any information related to the organization from the local database. This would both enhance data security and align with standard security practices, ensuring that only authorized users can initiate a new session upon reopening the application.

The logout process should also update the local database to remove the relevant organization's credentials.

Location

- /front-end/src/main/services/localUser/organizationCredentials.ts - `addOrganizationCredentials` and `updateOrganizationCredentials` functions

Low

Insecure Communications With Message Broker

Overall Risk	Low	**Finding ID**	NCC-E017270-RXH
Impact	Medium	**Component**	Code Review
Exploitability	Low	**Category**	Cryptography
		Status	New

Impact

An attacker with the ability to intercept network traffic would be able to monitor the affected services, potentially gaining access to credentials along with any other network traffic sent and received between the services and the message broker.

Description

The communications between the microservices composing the backend and the message broker (RabbitMQ) were not using SSL/TLS encryption. As a result, any information sent with the messages, including the RabbitMQ credentials, were transmitted in clear text.

As an example, the snippet below show one of the services connecting without any SSL options:

```
51 app.connectMicroservice({
52   transport: Transport.RMQ,
53   options: {
54     urls: [configService.getOrThrow<string>('RABBITMQ_URI')],
55     queue: API_SERVICE,
56   },
57 });
```

Figure 12: back-end/apps/api/src/setup-app.ts

The `RABBITMQ_URI` was set using the `amqp` protocol (without SSL) on port 5672 (port for cleartext communications)

```
7 # RabbitMQ Messaging URI
8 RABBITMQ_URI=amqp://rabbitmq:5672
```

Figure 13: back-end/apps/api/example.env

Recommendation

To mitigate these risks, it is recommended to implement SSL/TLS encryption for all internal communications between the services and RabbitMQ. This involves deploying and managing certificates to establish secure connections and updating service configurations to enforce encryption.

In this case, set the SSL options and use `amqps` (AMQP over SSL) protocol on port 5671.

It is also recommended to harden the RabbitMQ service by requiring mutual TLS authentication and configuring granular permissions for each queue, ensuring that each service can only subscribe to the queues necessary for its operation.

Location

- /back-end/apps/api/src/setup-app.ts

- /back-end/apps/chain/src/setup-app.ts
- /back-end/apps/notifications/src/setup-app.ts
- /back-end/libs/common/src/modules/api-proxy.module.ts
- /back-end/libs/common/src/modules/chain-proxy.module.ts
- /back-end/libs/common/src/modules/notifications-proxy.module.ts
- /back-end/k8s/back-end/rabbitmq-definition.yaml

Info

Empty or Missing Catch Statement

Overall Risk	Informational	**Finding ID**	NCC-E017270-HEN
Impact	Low	**Component**	Code Review
Exploitability	Low	**Category**	Auditing and Logging
		Status	New

Impact

The presence of empty or missing catch blocks in the codebase increases the risk of undetected errors, which could lead to unexpected behavior, security vulnerabilities, or system instability. By silently allowing exceptions, failures may occur without appropriate handling, potentially compromising the application's integrity and reliability.

Description

The codebase contained instances of empty or missing catch blocks. Empty or missing catch statements are generally considered poor coding practice, as the program is silently swallowing an error condition and then continuing execution. Failures that are met with no response should be avoided.

In general, if a catch statement has been reached, the program should at the very least log the error to a file, allowing the error to be identified later.

As an example, on `front-end/src/renderer/pages/UserLogin/UserLogin.vue` it can be seen a try with a finally missing a catch block:

```
126 if (userData) {
127       try {
128         globalModalLoaderRef?.value?.open();
129         await user.login(userData.id, userData.email.trim(), false);
130         if (isUserLoggedIn(user.personal)) {
131          user.setPassword(inputPassword.value);
132         }
133
134         if (user.secretHashes.length === 0) {
135          router.push({ name: 'accountSetup' });
136         } else {
137          router.push(
138            router.previousPath ? { path: router.previousPath } : { name: 'transactions' },
139          );
140         }
141       } finally {
142         buttonLoading.value = false;
143         globalModalLoaderRef?.value?.close();
144       }
145     }
```

Figure 14: /front-end/src/main/services/localUser/organizationCredentials.ts

Although it does not represent a security issue if an unexpected error occurs it won't be captured and logged by a catch block.

Recommendation

Ensure that all conditions are met with a suitable response. A trivial example, in which a non-existent file is read, is given below:[10]

```
const fs = require('fs');

try {
  const data = fs.readFileSync('log.txt', 'utf8');
  console.log(data);
} catch (e) {
  if (e.code === 'ENOENT') {
    console.log(`File not found exception: ${e.toString()}`);
  } else {
    console.log(`An error occurred: ${e.toString()}`);
  }
}
```

Location

- /front-end/src/renderer/pages/UserLogin/UserLogin.vue - line 127
- /front-end/src/renderer/pages/Transactions/components/ReadyToSign.vue - line 154
- /front-end/src/renderer/pages/Transactions/components/ReadyForReview.vue - line 160
- /front-end/src/renderer/pages/Transactions/components/ReadyForExecution.vue - line 123
- /front-end/src/renderer/pages/Transactions/components/InProgress.vue - line 119

10. Why are empty catch blocks a bad idea?: https://stackoverflow.com/questions/1234343/why-are-empty-catch-blocks-a-bad-idea

Info

Sequential IDs Used

Overall Risk	Informational	**Finding ID**	NCC-E017270-WQ3
Impact	None	**Component**	Code Review
Exploitability	Medium	**Category**	Data Exposure
		Status	New

Impact

The application made use of sequential identifiers which allowed programmatic access to user and transaction data without request limits, posing a potential risk of data enumeration and exposure despite the lack of sensitive information disclosure.

Description

The Hedera Transaction Tool backend made use of sequentially incremental identifiers. As a result, it was possible to programmatically access information about every user and transaction. It is worth noting that no apparent restrictions were in place to limit the number of requests being issued to the server.

Several endpoints were found to make use of sequential integer values, such as the ones used to retrieve information about every user and transaction:

- User's public keys:

```
GET /user/7/keys HTTP/1.1
Host: localhost:8090
Connection: keep-alive
Accept: application/json, text/plain, */*
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) hedera-
↳ transaction-tool/0.3.7 Chrome/128.0.6613.36 Electron/32.0.1 Safari/537.36
sec-ch-ua-platform: "Linux"
Origin: http://localhost:8083
Referer: http://localhost:8083/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US
Cookie: Authentication=eyJhbGciOi[REDACTED]bhPcsdJ28U
```

- Transaction information:

```
GET /transactions/9 HTTP/1.1
Host: localhost:8090
Connection: keep-alive
Accept: application/json, text/plain, */*
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) hedera-
↳ transaction-tool/0.3.7 Chrome/128.0.6613.36 Electron/32.0.1 Safari/537.36
sec-ch-ua-platform: "Linux"
Origin: http://localhost:8083
Referer: http://localhost:8083/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US
Cookie: Authentication=eyJhbGciOi[REDACTED]bhPcsdJ28U
```

The first endpoint also disclosed hashes for the mnemonic passphrases used when generating or importing private keys, as explained in finding "Exposed Mnemonic Hashes".

Even though no other sensitive information was being disclosed, using sequential object identifiers is usually considered to be a bad practice, especially since UUIDs were already being generated but not actively used:

```
export const register = async (email: string, password: string) => {
  const prisma = getPrismaClient();

  return await prisma.user.create({
    data: {
      id: randomUUID(),
      email: email,
      password: bcrypt.hashSync(password, 10),
    },
  });
};
```

Figure 15: hedera-transaction-tool-main/front-end/src/main/services/localUser/auth.ts

Recommendation

Consider using a universally unique identifier (UUID)[11] to represent object IDs, rather than incremental integers. This will significantly increase the difficulty for attackers attempting to exploit insecure direct object reference vulnerabilities.[12]

The OWASP guidelines suggest the use of the intermediate mapping table model. Dynamically-generated identifiers, rather than incremental integers, should be used to reference existing objects through a mapping table that is associated to the user session or user account.

Location

- /back-end/apps/api/src/user-keys/user-keys.controller.ts
- /back-end/apps/api/src/transactions/transactions.controller.ts

11. Universally unique identifier - Wikipedia: https://en.wikipedia.org/wiki/Universally_unique_identifier
12. OWASP - Insecure Direct Object Reference Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html

Info

Weaknesses in Private Keys Encryption and Password Hashing Practices

Overall Risk	Informational	**Finding ID**	NCC-E017270-Q47
Impact	High	**Component**	Code Review
Exploitability	Low	**Category**	Cryptography
		Status	New

Impact

Attackers who gain access to the application database had an increased likelihood of compromising user passwords and private keys through offline brute-force attacks. While the functions used (`PBKDF2` and `bcrypt`) are not considered insecure, they may not provide sufficient protection against determined attackers with access to modern hardware.

It is important to note that sufficiently strong passwords may still be impractical or impossible to crack under the current configurations, and therefore this finding was rated with an informational risk. However, the application accepted weak passwords (see finding "Weak Password Complexity Requirements") which would increase the exploitability of this finding.

Description

The frontend application implemented `PBKDF2` with `SHA-512` using 2,560 iterations to derive an `AES-256-GCM` encryption key from the user password to encrypt the private keys. While `AES-256-GCM` is a strong symmetric encryption algorithm, the `PBKDF2` parameters are considered inadequate for highly sensitive data such as private keys. Keychain passwords were also encrypted with the same method.

`PBKDF2` with `SHA-512`, though widely used, is no longer considered sufficient in configurations with low iteration counts like `2560`, as it is vulnerable to brute-force attacks using modern hardware (e.g., GPUs or ASICs). The number of iterations used in this case falls below current best practices for deriving keys for encryption of sensitive materials, such as private keys.

The code snippet below shows the code in charge of encrypting the private keys

/automation/utils/crypto.js:

```
1  const crypto = require('node:crypto');
2
3  function deriveKey(password, salt) {
4    const iterations = 2560;
5    const keyLength = 32;
6
7    return crypto.pbkdf2Sync(password, salt, iterations, keyLength, 'sha512');
8  }
9
10 function encrypt(data, password) {
11   const iv = crypto.randomBytes(16);
12   const salt = crypto.randomBytes(64);
13
14   const key = deriveKey(password, salt);
15
```

```
16    const cipher = crypto.createCipheriv('aes-256-gcm', key, iv);
17
18    const encrypted = Buffer.concat([cipher.update(data, 'utf8'), cipher.final()]);
19
20    const tag = cipher.getAuthTag();
21
22    return Buffer.concat([salt, iv, tag, encrypted]).toString('base64');
23  }
```

In addition, the user passwords themselves were hashed using `bcrypt` with 10 iterations before being stored in the database. While `bcrypt` is a secure password hashing algorithm 10 iterations is considered the minimum requirement by today's standards, which, for extremely sensitive material, such as the password protecting Hedera private keys, can be considered relatively weak. Also mnemonic words were hashed by this method and sent to the backend (see finding "Exposed Mnemonic Hashes").

The following snippets shows different places where bcrypt hashes were created with 10 iterations:

/front-end/src/main/modules/ipcHandlers/utils.ts

```
27  ipcMain.handle(createChannelName('hash'), (_e, data: string): string => {
28      return bcrypt.hashSync(data, 10);
29    });
```

/front-end/src/main/services/localUser/auth.ts

```
7   export const register = async (email: string, password: string) => {
8     const prisma = getPrismaClient();
9
10    return await prisma.user.create({
11      data: {
12        id: randomUUID(),
13        email: email,
14        password: bcrypt.hashSync(password, 10),
15      },
16    });
17  };
```

```
79  export const changePassword = async (userId: string, oldPassword: string, newPassword:
    ↳ string) => {
80    const prisma = getPrismaClient();
81
82    const isOldCorrect = await comparePasswords(userId, oldPassword);
83
84    if (isOldCorrect) {
85      await prisma.user.update({
86        where: {
87          id: userId,
88        },
89        data: {
90          password: bcrypt.hashSync(newPassword, 10),
91        },
92      });
93    } else {
94      throw new Error('Incorrect current password');
```

```
95    }
96
97    await changeDecryptionPassword(userId, oldPassword, newPassword);
98  };
```

In comparison, `bcrypt` with 10 iterations is harder to brute-force than `PBKDF2` with 2,560 iterations, yet both implementations lack sufficient computational cost to deter attackers equipped with modern hardware.

Recommendation

Consider migrating from `PBKDF2` and `bcrypt` to stronger key derivation and password hashing functions, such as `Argon2id`, which is resistant to GPU-based and ASIC-based attacks due to its memory-hard properties and resistance to parallel processing. It is recommended to perform tests on the time required to compute the hash or encrypt/decrypt the data using different values to find an acceptable compromise between security and usability.

If migration is not possible, it is recommended to increase the iterations in `PBKDF2` to at least 1,000,000, and increase the iterations in `bcrypt` to at least 12. Larger numbers can be used.

Moreover, consider whether it is needed to store a hash of certain information such as the Mnemonic hash, which, while impractical if the mnemonic was securely generated, open the possibility to brute-force the hash.

For further guidance, refer to the latest recommendations for password hashing algorithms, such as the OWASP Password Storage Cheat Sheet.

Location

- /automation/utils/crypto.js
- /front-end/src/main/services/localUser/auth.ts
- /front-end/src/main/modules/ipcHandlers/utils.ts

Info

Reusable Password Reset Tokens

Overall Risk	Informational	**Finding ID**	NCC-E017270-DKK
Impact	Low	**Component**	Code Review
Exploitability	Low	**Category**	Access Controls
		Status	New

Impact

An attacker who steals a password reset token would be able to use the 20 minute time window to reset the password again to gain control of the user account.

Description

The application's password reset mechanism used JWT tokens with a 20-minute expiration period to to allow user password changes. However, the token did not enforce one-time use, allowing it to be reused multiple times within the validity window. This behaviour enabled repeated password resets with the same token.

The password was reset in the following function:

```
154 /* Set the password for the user if the email has been verified */
155 @ApiOperation({
156   summary: 'Set the password',
157   description: 'Set the password for the verified email.',
158 })
159 @ApiResponse({
160   status: 200,
161   description: 'Password successfully set.',
162 })
163 @UseGuards(OtpVerifiedAuthGuard)
164 @Patch('/set-password')
165 async setPassword(
166   @GetUser() user: User,
167   @Body() dto: NewPasswordDto,
168   @Res({ passthrough: true }) response: Response,
169 ): Promise<void> {
170   await this.authService.setPassword(user, dto.password);
171   this.authService.clearOtpCookie(response);
172 }
173
174   @SkipThrottle()
175   @MessagePattern('authenticate-websocket-token')
176   @Serialize(AuthDto)
177   async authenticateWebsocketToken(@Payload() payload: AuthenticateWebsocketTokenDto) {
178     return this.authService.authenticateWebsocketToken(payload.jwt);
179   }
180 }
```

Figure 16: /back-end/apps/api/src/auth/auth.controller.ts

The OTP expiration time was set in the .env file, as shown below:

```
# OTP expiration in minutes
OTP_EXPIRATION=20
```


Best practices in password reset implementations typically generate a unique short-lived token per reset request, and invalidate it immediately upon use to minimize the risk of unauthorized access.

Recommendation

It is recommended that the application enforces a one-time use policy for tokens generated in the password reset functionality.

Location

- /back-end/apps/api/src/auth/auth.controller.ts, `setPassword` function
- /back-end/apps/api/src/auth/auth.service.ts

6 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

7 Contact Info

The team from NCC Group has the following primary members:

- Ángel Román – Security Consultant
- Víctor Lasa – Security Consultant

The team from Hedera Hashgraph has the following primary members:

- Joe Blanchard
- Simi Hunjan

8 Supplemental Data - Decrypt Script

The following script was used as a proof of concept to decrypt the encrypted private keys stored in the application's local database by using the user's password:

```
const crypto = require('crypto');

function deriveKey(password, salt) {
  const iterations = 2560;
  const keyLength = 32;
  return crypto.pbkdf2Sync(password, salt, iterations, keyLength, 'sha512');
}

function decrypt(encryptedBase64Data, password) {
  const data = Buffer.from(encryptedBase64Data, 'base64');

  // Extract salt, IV, authentication tag, and encrypted data
  const salt = data.slice(0, 64);
  const iv = data.slice(64, 80);      // 16 bytes IV
  const tag = data.slice(80, 96);     // 16 bytes auth tag
  const encrypted = data.slice(96);   // Remaining bytes

  // Derive the key using the same parameters as in encryption
  const key = deriveKey(password, salt);

  // Create decipher instance
  const decipher = crypto.createDecipheriv('aes-256-gcm', key, iv);

  // Set the authentication tag
  decipher.setAuthTag(tag);

  // Decrypt the data
  let decrypted = decipher.update(encrypted);
  decrypted = Buffer.concat([decrypted, decipher.final()]);

  return decrypted.toString('utf8');
}

const args = process.argv.slice(2);

if (args.length < 2) {
  console.error('Usage: node decrypt.js <base64_encrypted_data> <password>');
  process.exit(1);
}

const encryptedData = args[0];
const password = args[1];

try {
  const decryptedData = decrypt(encryptedData, password);
  console.log('Decrypted Data:\n', decryptedData);
} catch (err) {
  console.error('Error decrypting data:', err.message);
}
```

Figure 17: decrypt.js