QuillAudits

AUDIT REPORT

January 2026

For

B2Vault

Table of Content

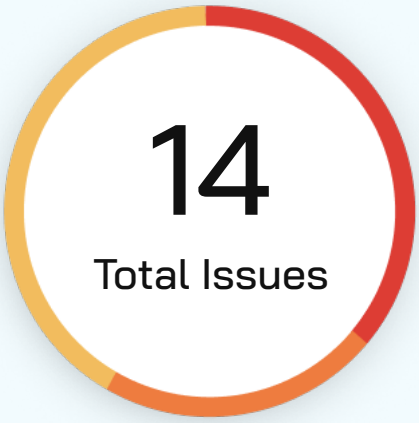
Executive Summary

Project Name	B2VAULT
Protocol Type	Source Code Review
Project URL	https://github.com/mbagherzad1367/CustodyFront https://github.com/mbagherzad1367/CustodyBackend
Overview	B2Vault is a non-custodial cryptocurrency wallet designed for team usage. B2Vault offers institutional-grade, comprehensive digital asset self-custody services and MPC privatization solutions, enabling enterprises to securely and efficiently manage their digital assets.
Review 1	29th Dec 2025 - 5th Jan 2026
Updated Code Received	14th Jan 2026
Review 2	15th Jan 2026
Fixed In	42c7216fd86871602bcf191adf5d80244b775b14

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0(0.0%)
High	5 (35.8%)
Medium	3 (21.4%)
Low	6 (42.8%)
Informational	0(0.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	**5**	**3**	**6**	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	SQL Injection Exploited - Data Extraction	High	Resolved
2	Hardcoded Database Credentials	High	Resolved
3	Hardcoded Encryption Key for Private Key Storage	High	Resolved
4	Endpoint Exposes Private Keys	High	Resolved
5	Private Keys & Mnemonics Exposed in API Response	High	Resolved
6	CORS Allows Any Origin with Credentials	Medium	Resolved
7	Base64 Encoding Used as “Encryption”	Medium	Resolved
8	No Rate Limiting on Authentication Endpoints	Medium	Resolved
9	Unsafe Content Security Policy	Low	Resolved
10	Mock/Fallback Encryption Provides No Security	Low	Resolved
11	Private Keys Stored in Database Model	Low	Resolved

Issue No.	Issue Title	Severity	Status
12	Insecure OTP Generation Using Math.random()	Low	Resolved
13	Shamir Secret Sharing With No Redundancy	Low	Resolved
14	Email Transport Without TLS	Low	Resolved

Checked Vulnerabilities

- Improper Authentication
- Improper Resource Usage
- Improper Authorization
- Insecure File Uploads
- Insecure Direct Object References
- Client-Side Validation Issues
- Rate Limit
- Input Validation
- Injection Attacks
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery
- Security Misconfiguration
- Broken Access Controls
- Insecure Cryptographic Storage
- Insufficient Cryptography
- Insufficient Session Expiration
- Insufficient Transport Layer Protection
- Unvalidated Redirects and Forwards
- Information Leakage
- Broken Authentication and Session Management
- Denial of Service (DoS) Attacks
- Malware
- Third-Party Components

And More..

Techniques and Methods

Throughout the pentest of application, care was taken to ensure:

- Information gathering – Using OSINT tools information concerning the web architecture, information leakage, web service integration, and gathering other associated information related to web server & web services.
- Using Automated tools approach for Pentest like Nessus, Acunetix etc.
- Platform testing and configuration
- Error handling and data validation testing
- Encryption-related protection testing
- Client-side and business logic testing

Tools and Platforms used for Pentest:

Burp Suite	Acunetix	Nmap
DNSenum	Neucli	Metasploit
Dirbuster	Nabbu	Horusec
SQLMap	Turbo Intruder	Postman
Netcat	Nessus	And Many more..

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

SQL Injection Exploited - Data Extraction

Resolved

Description

The SQL injection vulnerability was identified and was successfully exploited against the live production API at backend.paidgate.com. Despite API responses being encrypted with CryptoJS AES-256-CBC, the encryption key was provided by client, allowing full decryption of all responses. A custom decryption proxy was developed to automate the exploitation and data extraction process.

Target Endpoint

GET https://backend.paidgate.com/wallet/report

Vulnerable Parameters Confirmed

- field (ORDER BY injection)
- sort (ORDER BY injection)
- currency (ILIKE/WHERE clause injection)

Data Extracted

```
Users Extracted (3):
    - Shivang Tester: 2 transactions, 1.0000 total
    - Mo bagherzad: 29 transactions, 20.6000 total
    - Mark Smith: 37 transactions, 21.1500 total

Wallet Addresses Extracted (33):
    - 0x759be5f4bc74634a0924f06c6e5e6a722f15daf9
    - pPoowbSbYRdXNedj6M89cZZdibCscTfXPKzouQh2fEq
    - 0xc9610DaAC2a9C0336af6615736FB13Af141Dd3D7
    - 0x7e28dc2cf8e05988896e95da6a47ef831e800ac7
    - 0x7aa49d75e65b5a08ee8f06525ba25a45eb206f8b
    - TBgQoBwajC7i8FegZNFXBXUyimC3fJvD1Y
    - 0xad1d01d88b9425636096509cdbe3415471349dc5
    - TN8jrXoTiyj5moZgxXkyaEw9JdruumiMcA
    - 0x6be76faef0f7cc0ced8243a6ce16ff00d905f919
    - 0x34aa573cc6dc3d18f1a54caad2047a2582ea63e1
    [… and 23 more addresses]

Public Keys Extracted (15):
    - 0x03c2240d6a2517250522e8aa36846174417a70…
    - 0x0463a62ea78f14e0f21bb3b4cc2695b9608cd0…
    - 0x0202233c3d11ff3003afc3a9306923d50920aa…
    - 0x046c684ced4a6d62eda69397d09e1ba6834bb8…
    - 0x029b027350d10ab3bd631f5e8e55ada1d9d7cc…
    [… and 10 more public keys]
```

```
Vault IDs Extracted (11):
    - M7IJN64MLS
    - 407AQYAN67
    - BH7UQCWUBC
    - CJ53S4SG2S
    - SOHPE1XWMT
    - 0QCLYAPTHM
    - 0QUVKAGB2K
    - WTBP2S36AU
    - TK2ZH3CBC1
    - MUBNU9IM6M
    - 4MNH4ZH83X

Named Wallets with Balances:
    - Lima Fintech (USDC_ERC20): 0x759be5f4bc74634a0924f06c6e5e6a722f15daf9
    - Vuka Pay (USDT_ERC20): 0x34aa573cc6dc3d18f1a54caad2047a2582ea63e1
    - VukaCasino Ltd (USDT_ERC20): Balance 0.2
    - Sigma Casino (USDT_ERC20): Balance 0.1, (USDT_TRC20): Balance 2.5
    - Final LL Test (USDT_ERC20): Balance 0.3
    - Rome Casino A (USDT_ERC20): Balance 0.2
    - Test LL (USDT_ERC20): Balance 1.0

Total Transactions Extracted: 77
```

Impact

• Full database enumeration possible via SQL injection
• All user data, wallet addresses, and transaction history exposed

Recommendation

- IMMEDIATE: Take API offline until SQL injection is patched
- Use parameterized queries for ALL database operations
- Rotate ALL encryption keys immediately
- Implement proper input validation
- Deploy WAF with SQL injection detection rules

Hardcoded Database Credentials

Resolved

Description

Database credentials are hardcoded in plaintext directly in the source code configuration file. The same password "Oa6DlSbavLa0NNp" is reused across all three database configurations (development, db2, db3). These credentials are committed to the repository, meaning anyone with repository access can obtain full database access. This is a fundamental security violation as secrets should never be stored in source code.

Vulnerable Endpoint

CustodyBackend-master/config/config.js

```
Vulnerable Code (Lines 5-6, 27-28, 48-49):
    module.exports = {
      development: {
        username: "cryptoprocessingdb",
        password: "Oa6DlSbavLa0NNp",  // HARDCODED PASSWORD
        database: "postgres",
        host: "db",
        port: 5432,
        dialect: "postgres",
      },
      db2: {
        username: "cryptoprocessingdb",
        password: "Oa6DlSbavLa0NNp",  // SAME PASSWORD REUSED
        database: "postgresprime",
      },
      db3: {
        username: "cryptoprocessingdb",
        password: "Oa6DlSbavLa0NNp",  // SAME PASSWORD REUSED
        database: "postgressecond",
      }
    };

Also Found In:
/CustodyBackend-master/restore_db.sh (Line 14):
    PGPASSWORD="Oa6DlSbavLa0NNp"
```

Impact

- Complete database access if repository is leaked or accessed
- Access to all user data, wallet information, and encrypted private keys
- Password reuse across 3 databases means single compromise affects entire system
- Attacker can dump all tables including users, wallets, transactions, keys
- Combined with Finding 2, private keys can be fully decrypted

Recommendation

- Remove all hardcoded credentials immediately
- Use environment variables exclusively: process.env.DB_PASSWORD
- Implement secrets management (HashiCorp Vault, AWS Secrets Manager)
- Rotate the exposed password immediately on all systems
- Add pre-commit hooks to prevent credential commits
- Audit repository history for other leaked secrets

Hardcoded Encryption Key for Private Key Storage

Resolved

Description

A 64-character encryption key is hardcoded as a fallback value in the KMS (Key Management Service) module. This key is used for AES-256-GCM encryption of Shamir secret shares that protect cryptocurrency private keys. If the environment variable ENCRYPTION_SECRET_KEY is not set, this hardcoded key is used to encrypt/decrypt all private key material. This completely defeats the purpose of encryption since the key is visible in source code.

Vulnerable File

/CustodyBackend-master/files/kms.js

Vulnerable Code (Line 28):

```
    if (WHICH_CLOUD === "VPS") {
      const SECRET_KEY = process.env.ENCRYPTION_SECRET_KEY ||
        '5fTgYHMbQeThWmZq4t7w9u$Xn2r5u8x!A%D*G-KaPdSgVkYp3s6v9y$B&E)H@McQ';

      // This key is used for AES-256-GCM encryption of Shamir secret shares
      const getKey = () => {
        return crypto.createHash('sha256').update(SECRET_KEY).digest();
      };
    }
```

Impact

- All Shamir secret shares for cryptocurrency private keys can be decrypted
- Attacker with database access can reconstruct wallet private keys
- Complete compromise of all custodied cryptocurrency assets
- Potential theft of millions in cryptocurrency depending on custody balance
- No defense-in-depth if database is breached

Recommendation

- Remove hardcoded fallback key entirely
- Require environment variable with no fallback (fail-safe approach)
- Use Hardware Security Module (HSM) for key management
- Implement key rotation mechanism

Endpoint Exposes Private Keys

Resolved

Description

The /snapshot/take endpoint is defined BEFORE the authentication middleware is applied in the route file, meaning it executes without any authentication. This endpoint queries all wallets from the database including private keys, addresses, asset IDs, user IDs, vault IDs, and wallet names. Any attacker can trigger this endpoint without credentials and cause all wallet private keys to be processed and stored in the snapshot table.

Vulnerable File

/CustodyBackend-master/routes/snapshotRoute.js

```
Vulnerable Code (Line 11):
    // Line 11 - NO AUTHENTICATION MIDDLEWARE
    router.route("/take").get(catchAsync(snapshotApi));

    // Authentication middleware applied AFTER on lines 12-13
    router.use(authentication);
    router.use(restrictTo("all"));

Controller Code (/controller/snapController.js, Lines 44-74):
    const snapshotApi = async (req, res, next) => {
      const wallets = await wallet.findAll({
        attributes: [
          "address",
          "privateKey",      // PRIVATE KEYS INCLUDED IN QUERY
          "assetId",
          "userId",
          "vaultId",
          "walletName",
        ],
      });
      // Creates snapshots containing private keys
      await snapshot.bulkCreate(allSnapshots);
    };
```

Impact

- Complete exposure of all cryptocurrency private keys
- No authentication required to trigger snapshot creation
- Attacker can steal all custodied cryptocurrency assets

Recommendation

Move route AFTER authentication middleware immediately:
router.use(authentication);
router.use(restrictTo("admin"));
router.route("/take").get(catchAsync(snapshotApi));
Remove privateKey from snapshot attributes entirely

Private Keys & Mnemonics Exposed in API Response

Resolved

Description

Private keys and mnemonic phrases are returned DIRECTLY in normal API responses from the /wallet/reports endpoint. ANY authenticated user can extract the full private keys and BIP39 mnemonic phrases for wallets they have visibility into. This is a Critical vulnerability - no SQL injection or exploitation needed.
The sensitive cryptographic material is simply returned as part of the normal transaction data in the sourceAsset and targetAsset fields.This finding demonstrates COMPLETE FAILURE of sensitive data protection and
represents the highest possible security impact - immediate theft of all cryptocurrency assets is possible by any authenticated user.

Vulnerable Endpoint

```
Request:
    GET /wallet/reports?pageSize=100&pageNumber=1&field=createdAt&sort=DESC
    Host: backend.paidgate.com
    Authorization: Bearer [ANY_VALID_TOKEN]

Response (after decryption):
    {
      "body": {
        "data": [{
          "targetAsset": {
            "address": "TE94hsV8gbXNQUQivpVEptVqjURWP2PCnt",
            "privateKey": "xxxxxxxx",
            "mnemonic": "xxxxxxxxxxxx august",
            "walletName": "Charlies",
            "assetId": "USDT_TRC20"
          }
        }]
      }
    }
```

Steps to Reproduce

1. Wallet created with 3 shares, threshold 3
2. One KMS key becomes unavailable (rotation, deletion, permission change)
3. Cannot reconstruct private key with only 2 shares
4. Funds in wallet are permanently inaccessible
5. No recovery possible without all 3 shares

Impact

- COMPLETE THEFT of all custodied cryptocurrency (100% asset loss)
- No exploitation required - keys returned in normal API responses
- Any authenticated user becomes an insider threat
- BIP39 mnemonic provides PERMANENT wallet access (cannot be rotated)
- Single-point-of-failure for entire custody platform

Recommendation

Remove privateKey and mnemonic from ALL Sequelize includes:
attributes: { exclude: ['privateKey', 'mnemonic'] }
Deploy hotfix to production
RECHECK all API endpoints for similar exposure

Medium Severity Issues

CORS Allows Any Origin with Credentials

Resolved

Description

The CORS (Cross-Origin Resource Sharing) configuration is set to accept requests from ANY origin by returning the requesting origin or a wildcard “*”. Combined with credentials: true, this allows any malicious website to make authenticated API requests on behalf of logged-in users. This completely defeats the browser’s same-origin security policy and enables Cross-Site Request Forgery (CSRF) attacks from any domain.

Vulnerable File

/CustodyBackend-master/server.js

Vulnerable Code (Lines 22-27):

```
    const corsOptions = {
      origin: (origin, callback) => {
        callback(null, origin || "*");  // ALLOWS ANY ORIGIN
      },
      credentials: true,  // SENDS CREDENTIALS TO ANY ORIGIN
    };

    app.use(cors(corsOptions));
```

Impact

- Any malicious website can make authenticated API requests
- Cross-site request forgery (CSRF) attacks enabled
- User wallet data can be exfiltrated from any website
- Session hijacking possible

Recommendation

- Whitelist specific trusted origins only:
  const ALLOWED_ORIGINS = ['https://custody.b2vault.com'];

Base64 Encoding Used as "Encryption"

Resolved

Description

The application uses Base64 encoding (btoa/atob functions) throughout the frontend code as if it were encryption. Base64 is merely an encoding scheme that converts binary data to ASCII text - it provides absolutely zero security. Any data "protected" with Base64 can be decoded instantly by anyone. Function names like "encodeAuthBody" and "setSecurityBody" suggest developers believe this provides security.

Vulnerable File

```
 /CustodyFront-master/src/service/LocalstorageService.tsx
Vulnerable Code (Lines 25-26, 29-30, 44-46):
    encodeAuthBody(data: any) {
      return localStorage.setItem('authBody', btoa(JSON.stringify(data)));
    }

    setSecurityBody(data: any) {
      return localStorage.setItem('secureBody', btoa(JSON.stringify(data)));
    }

    encodeSwitchAccounts(data: any) {
      return localStorage.setItem('switchAccounts',
btoa(JSON.stringify(data)));
    }

Also in /CustodyFront-master/src/utils/EncryptedStorage.ts (Lines 11, 16):
    getItem(key: string) {
      const stringfied = decodeURIComponent(atob(data));
    }
    setItem(key: string, data: unknown) {
      return this.storage.setItem(key,
btoa(encodeURIComponent(JSON.stringify(data))));
    }
```

Steps to Reproduce

1. Log into the application
2. Open browser DevTools > Application > Local Storage
3. Copy the authBody value
4. Decode in console: atob('copied_value')
5. All authentication data is revealed in plaintext

Impact

- False sense of security - developers believe data is encrypted
- All "encrypted" user data, roles, and permissions are trivially readable
- Authentication bypass potential if data is modified and re-encoded
- No actual protection of sensitive client-side data

Recommendation

- Remove base64 encoding (it provides zero security)
- Don't store sensitive data client-side at all
- For necessary client-side data, use HTTP-only cookies instead
- If encryption is truly needed, use proper cryptographic libraries with server-controlled key

No Rate Limiting on Authentication Endpoints

Resolved

Description

All authentication endpoints (login, OTP verification, 2FA verification, password reset) lack rate limiting middleware. Attackers can make unlimited requests to brute-force passwords, guess OTPs, or enumerate valid accounts. With only 1,000,000 possible 6-digit OTPs and 1-minute expiry, an attacker with fast network connection could potentially guess valid OTPs.

Vulnerable File

```
/CustodyBackend-master/routes/authRoute.js
/CustodyBackend-master/controller/authController.js

Evidence - No rate limiting middleware on:
    router.route("/signup").post(catchAsync(signup));
    router.route("/login").post(catchAsync(login));
    router.route("/verifyEmail").post(verifyEmail);
    router.route("/verifyOTP").post(verifyOTP);
    router.route("/verify-two-factor-otp").post(verifyTwoFactorOTP);
    router.route("/reset_password").post(resetPassword);
```

Impact

- Brute-force attacks on passwords
- OTP guessing (1M combinations for 6-digit OTP)
- Account enumeration via different error messages
- Credential stuffing attacks

Recommendation

Implement rate limiting using express-rate-limit:

```
const rateLimit = require('express-rate-limit');
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // 5 attempts
  message: 'Too many login attempts'
});
router.post('/login', loginLimiter, login);
```

Add account lockout after N failed attempts
Implement CAPTCHA after failed attempts
Use progressive delays between attempts

Low Severity Issues

Unsafe Content Security Policy

Resolved

Description

The Content Security Policy (CSP) header is configured with 'unsafe-inline' and 'unsafe-eval' directives for script-src. These directives completely undermine CSP's protection against Cross-Site Scripting (XSS) attacks. 'unsafe-inline' allows execution of inline scripts (the primary XSS vector), and 'unsafe-eval' allows use of eval() and similar dangerous functions. This CSP provides essentially no XSS protection.

Vulnerable File

/CustodyFront-master/vite.config.ts

Vulnerable Code (Lines 34-45):

```
    res.setHeader(
      'Content-Security-Policy',
      "default-src 'self'; " +
        "script-src 'self' 'unsafe-inline' 'unsafe-eval'; " +  // DANGEROUS
        "style-src 'self' 'unsafe-inline'; " +                 // DANGEROUS
        "img-src 'self' data: https:; " +
        "font-src 'self' data:; " +
        "connect-src 'self'; " +
        "frame-ancestors 'none'; " +
        "base-uri 'self'; " +
        "form-action 'self'"
    );
```

Impact

- eval() and new Function() can execute attacker code
- Inline scripts can be injected and executed
- CSP header provides false sense of security
- Combined with Finding 9, XSS leads to token theft

Recommendation

- Remove 'unsafe-inline' and 'unsafe-eval' directives
- Use nonces for necessary inline scripts:
  script-src 'self' 'nonce-{random}';
- Refactor code to eliminate need for eval()
- Move inline scripts to external files

Mock/Fallback Encryption Provides No Security

Resolved

Description

The KMS module contains fallback encryption that activates when the primary KMS service fails. This fallback simply prepends “ENCRYPTED:” to the plaintext and base64 encodes it - providing zero actual encryption. Private keys “encrypted” with this fallback are stored in effectively plaintext.
The code logs “Falling back to mock format” but continues processing, silently compromising security.

Vulnerable File

CustodyBackend-master/files/kms.js

```
Vulnerable Code (Lines 59-61, 92-105):
    // Lines 59-61 - Fallback "encryption" is just base64
    } catch (error) {
      console.error("Encryption error:", error);
      // Fallback to original mock behavior on error
      return Buffer.from(`ENCRYPTED:${plaintext}`).toString('base64');
    }

    // Lines 92-105 - Mock decryption
    const decryptMockFormat = (ciphertextBase64) => {
      console.log("Falling back to mock format decryption");
      const text = Buffer.from(ciphertextBase64, 'base64').toString('utf-8');
      if (text.startsWith("ENCRYPTED:")) {
        return text.replace("ENCRYPTED:", "");
      }
      return ciphertextBase64;
    }
```

Impact

- Silent security failure - no alerts when fallback activates
- Anyone with database access can decode all private keys

Recommendation

- Remove all mock/fallback encryption code
- Fail hard if KMS is unavailable (fail-secure approach)
- Implement health checks for KMS availability before processing
- Alert immediately on any KMS failures

Private Keys Stored in Database Model

Resolved

Description

The database models define privateKey and mnemonic as plain STRING columns with no encryption at the database level. While the application may encrypt these values before storage, the schema allows storing them in plaintext.
Combined with the mock encryption fallback (Finding 13), private keys may
actually be stored unencrypted. The database provides no defense-in-depth.

Vulnerable File

/CustodyBackend-master/db/models/wallet.js

```
Vulnerable Code (Lines 33-35):
    mnemonic: { type: DataTypes.STRING },
    privateKey: { type: DataTypes.STRING },  // Stored as plain string
    publicKey: { type: DataTypes.STRING },

Also in /CustodyBackend-master/db/models/adminWallet.js (Lines 32-34):
    privateKey: { type: DataTypes.STRING },
    mnemonic: { type: DataTypes.STRING },
```

Impact

- Database breach = complete loss of all cryptocurrency
- No defense in depth for key material
- Backup phrases allow permanent access to wallets
- DB admins have access to key material

Recommendation

- Never store raw private keys in database
- Use Shamir Secret Sharing with proper threshold (2-of-3, not 3-of-3)
- Store only encrypted key shares in separate tables
- Consider HSM-based key storage

Insecure OTP Generation Using Math.random()

Resolved

Description

One-Time Password (OTP) codes for email verification and password reset are generated using Math.random(), which is a pseudo-random number generator (PRNG) that is NOT cryptographically secure. The PRNG state can be predicted if enough outputs are observed, allowing attackers to predict future OTP values. Additionally, only 6 digits (1,000,000 combinations) provides insufficient entropy for security-critical operations.

Vulnerable File

/CustodyBackend-master/controller/authController.js

Vulnerable Code (Line 283):

```
const otp = Math.floor(100000 + Math.random() * 900000).toString();
```

Impact

- Password reset OTPs can be predicted
- Account takeover via predicted OTP
- 2FA bypass potential
- Only 1M combinations allows brute-force in under 1 minute without rate limiting

Recommendation

- Use crypto.randomInt() for cryptographically secure random numbers:
  const crypto = require('crypto');
  const otp = crypto.randomInt(100000, 999999).toString();
- Consider 8-digit OTPs for additional entropy
- Implement rate limiting (Finding 17) to prevent brute-force
- Add account lockout after failed OTP attempts

Shamir Secret Sharing With No Redundancy

Resolved

Description

Shamir's Secret Sharing is configured with threshold=3 requiring ALL 3 shares to reconstruct the secret. This eliminates the entire benefit of secret sharing, which is to provide redundancy and fault tolerance. If any single share is lost, corrupted, or becomes inaccessible (e.g., KMS key rotation issue), funds are permanently and irrecoverably lost.

Vulnerable File

/CustodyBackend-master/controller/walletController.js

Vulnerable Code (Lines 124, 222):

```
    const shares = split(secret, { shares: 3, threshold: 3 });
```

Steps to Reproduce

1. Wallet created with 3 shares, threshold 3
2. One KMS key becomes unavailable (rotation, deletion, permission change)
3. Cannot reconstruct private key with only 2 shares
4. Funds in wallet are permanently inaccessible
5. No recovery possible without all 3 shares

Impact

- Threshold 3/3 means ALL shares required
- Loss of ANY single key share = permanent loss of funds
- No fault tolerance in key recovery
- KMS failure = wallet inaccessible
- Defeats purpose of secret sharing

Recommendation

- Use threshold scheme like 2-of-3:
  const shares = split(secret, { shares: 3, threshold: 2 });
- This allows recovery if one share is lost
- Consider 3-of-5 for higher security with redundancy
- Implement share backup and recovery procedures
- Regular validation that all shares can be accessed

Email Transport Without TLS

Resolved

Description

The email transport is configured with secure: false, meaning email communication with the SMTP server is not encrypted with TLS. This exposes email credentials during SMTP authentication and allows interception of email contents including OTP codes, password reset links, and other sensitive information. Any network observer can capture this traffic.

Vulnerable File

CustodyBackend-master/files/sendEmail.js

Vulnerable Code (Lines 4-10):

```
    const transport = nodemailer.createTransport({
      host: process.env.EMAIL_HOST,
      port: process.env.EMAIL_PORT,
      secure: false,  // NO TLS ENCRYPTION
      auth: {
        user: process.env.EMAIL_USERNAME,
        pass: process.env.EMAIL_PASSWORD,
      },
    });
```

Impact

- Email credentials transmitted in cleartext to SMTP server
- OTP codes can be intercepted in transit
- Password reset emails exposed to MITM attacks
- Account takeover via intercepted reset links
- Credential theft if email password reused

Recommendation

```
const transport = nodemailer.createTransport({
    host: process.env.EMAIL_HOST,
    port: 465,  // SSL port
    secure: true,  // Use TLS
    auth: {
      user: process.env.EMAIL_USERNAME,
      pass: process.env.EMAIL_PASSWORD,
    },
    tls: {
      rejectUnauthorized: true  // Verify certificates
    }
  });
```

Closing Summary

In this report, we have considered the security of the B2 VAULT. We performed our audit according to the procedure described above.

Some issues of high, medium and low severity were found. B2 Vault team resolved all the issues mentioned.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

January 2026

For

B2Vault

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com