QuillAudits

AUDIT REPORT

March, 2025

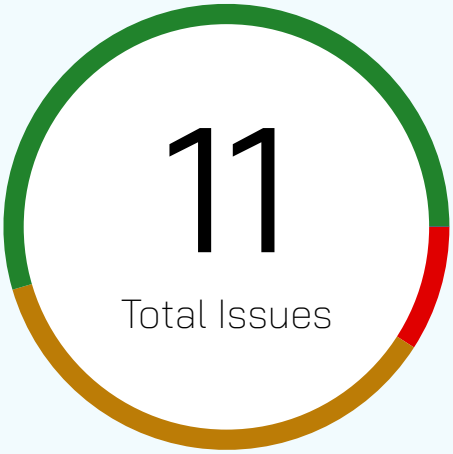
For

araa rypto

Executive Summary

Project name	Paraacrypto
Overview	Paraacrypto is a trading and investment platform designed to help you grow your wealth in the digital asset world.
Audit Scope	Scope of the Audit was limited to a Source code review of ParraCrypto Platform
In Scope:	frontend: https://github.com/paraa-crypto/paraacryp-to-frontend Backend: https://github.com/paraa-crypto/Backend-Node-Js
Timeline	27th February 2025 - 4th March 2025
Updated Code Received	7th March 2025
Review 2	9th march 2025 - 10th March 2025
Fixed in	https://github.com/paraa-crypto/Backend-NodeJs/com-mit/7b3a8d26abc4fb1359ee92361beaac6238292358 https://github.com/paraa-crypto/paraacrypto-fron-tend/tree/09eef76682bfcd6c343f41eb7667a6b02868ca1b

Number of Issues per Severity



High	1 (9.09%)
Medium	4 (36.36%)
Low	6 (54.55%)
Informational	0 (0.00%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**1**	**4**	**6**	0
Acknowledged	0	0	0	0
Partially Resolved	0	0	0	0

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 & Methods

Throughout the pentest of applications, 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	Nabbu	Dirbuster
DNSEnum	Turbo Intruder	SQLMap
Acunetix	Nmap	Horusec
Neucil	Metasploit	Postman
Netcat	Nessus	and many more...

Types of Severity

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

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

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

Resolved

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

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.

High Severity Issues

HardCoded Credentials

Resolved

Description

The source code contains hardcoded Redis credentials, including the password, host, and port, which exposes the database to unauthorized access. Attackers with access to the repository or deployed code can use these credentials to connect to the Redis instance, potentially leading to data exposure, privilege escalation, or full database compromise.

Vulnerable File :src/helpers/order.helper.ts

Recommendation

1. Remove Hardcoded Credentials and shift them to .env
2. Modify Redis configuration to bind only to localhost and Apply firewall rules to block external access to Redis

Impact

1. Unauthorized Access & Data Leakage: Attackers can connect to the Redis instance and retrieve cached sensitive data such as user sessions, API keys, and authentication tokens.
2. Privilege Escalation & Application Takeover: If Redis is used for session management or authentication, an attacker could modify user roles, impersonate privileged accounts, or gain admin access.
3. Remote Code Execution (RCE) via Redis Exploitation: If Redis is misconfigured, attackers could write malicious scripts, inject cron jobs, or execute arbitrary commands, potentially leading to full server takeover.

Medium Severity Issues

Missing Secure HTTP Server Configuration

Resolved

Description

The application initializes an HTTP server without enforcing secure communication. This exposes data in transit to man-in-the-middle (MITM) attacks, session hijacking, and credential theft. Without HTTPS and proper security configurations, sensitive information such as API tokens, authentication credentials, and user data can be intercepted.

Vulnerable File :src/index.ts

Recommendation

Enforce HTTPS & Redirect HTTP to HTTPS
Replace HTTP with an HTTPS server using TLS certificates

```
const https = require("https");
const fs = require("fs");
const options = {
 key: fs.readFileSync("/path/to/ssl/private.key"),
 cert: fs.readFileSync("/path/to/ssl/certificate.crt"),
};
const httpsServer = https.createServer(options, app);
httpsServer.listen(443, () => console.log("Secure server running on port 443"));
```

Use HSTS (HTTP Strict Transport Security): Add HSTS headers to enforce HTTPS and prevent protocol downgrades:

```
const helmet = require("helmet");
 app.use(helmet.hsts({ maxAge: 31536000, includeSubDomains: true, preload: true }))
```

Impact

1. Data Exposure via MITM Attacks: Without HTTPS, attackers on the same network can intercept API requests, steal credentials, and manipulate responses.
2. Session Hijacking & Token Theft: If authentication tokens (e.g., JWTs, cookies) are transmitted over HTTP, an attacker can capture and reuse them to impersonate users.
3. Insecure API Calls & Downgrade Attacks: APIs accessed via HTTP are vulnerable to downgrade attacks, where an attacker forces clients to use a guy n unencrypted connection

Leakage of Sensitive Data in Local Storage

Resolved

Description

The application stores authentication tokens in localStorage, which poses a serious security risk. Since localStorage is accessible via JavaScript, any XSS (Cross-Site Scripting) attack can extract and leak these tokens, leading to account takeover and session hijacking.

Vulnerable File :src/app/(dashboard)/register/page.tsx
Vulnerable Code (Line 97): localStorage.setItem("token", res?.data?.token);

Recommendation

1. Use Secure httpOnly Cookies Instead of Local Storage: Modify authentication to store tokens in secure, httpOnly cookies, preventing JavaScript access
2. Implement Secure Token Storage with Session Management: If session-based storage is needed, use sessionStorage instead of localStorage to ensure the token is cleared when the user closes the browser
3. Implement Token Expiry & Rotation: Set short-lived access tokens and refresh them securely via server-managed refresh tokens in httpOnly cookies.

Impact

1. Token Theft via XSS Attacks: If an attacker injects malicious JavaScript (via XSS), they can access localStorage and extract the token
2. Persistence Beyond Logout: Unlike httpOnly cookies, tokens stored in localStorage do not expire on session logout, allowing attackers to reuse stolen tokens indefinitely.
3. Any malicious browser extension can access localStorage, increasing the risk of data leakage.

API KEY Leakage

Resolved

Description

The source code contains hardcoded API keys for SendGrid (email service) and Google Firebase (cloud storage and authentication), leading to a critical security risk. Exposing these credentials allows unauthorized access to email services, user data, and cloud storage, potentially leading to data theft, spam abuse, or full account takeover.

Recommendation

1. Remove Hardcoded API Keys from Source Code and Rotate Compromised API Keys Immediately.
2. Restrict API Key Permissions
Limit SendGrid API access to allow only email sending, preventing abuse.
Set Firebase firestore and storage rules to restrict unauthorized access.

Impact

SendGrid API Key Exposure
Attackers can send phishing emails using your SendGrid account, damaging reputation.
Potential for spam abuse, leading to domain blacklisting.
If permission levels allow, an attacker could read and modify email logs.

Google Firebase API Key Exposure
If rules are misconfigured, attackers can access Firebase database contents, including user data, authentication logs, or cloud-stored files.
Firebase keys can be used for unauthorized authentication if authDomain settings allow public sign-ins.
Potential for privilege escalation if Firebase functions include sensitive API endpoints

Low Severity Issues

Insecure Random Number Generation Using Math.random

Resolved

Description

The application uses Math.random() for generating random values in multiple locations. Math.random() is not cryptographically secure, meaning its output can be predicted by an attacker. This vulnerability can lead to insecure token generation, weak authentication mechanisms, and predictable session identifiers, increasing the risk of exploitation.

Vulnerable File :Various locations within the codebase

Recommendation

Use a Cryptographically Secure Random Function:
Instead of Math.random(), use randomBytes or getRandomValues .
Using Math.random() in security-sensitive contexts exposes the application to predictability attacks.

Improper CORS Configuration

Resolved

Description

The application uses app.use(cors()) in src/index.ts without specifying a restrictive configuration. By default, this enables CORS for all origins (*), allowing any website to make requests to the server. This misconfiguration can lead to Cross-Origin Resource Sharing (CORS) abuse, where an attacker can make unauthorized API requests from a malicious website.

Vulnerable File :src/index.ts

Recommendation

Restrict Allowed Origins : Define specific trusted origins instead of allowing all (*)

```
app.use(cors({
 origin: ["https://yourdomain.com"], // Allow only specific domains
 methods: ["GET", "POST"], // Restrict allowed methods
 credentials: true, // Only if cookies/tokens need to be sent
}));
```

Unsanitized dynamic input in file path

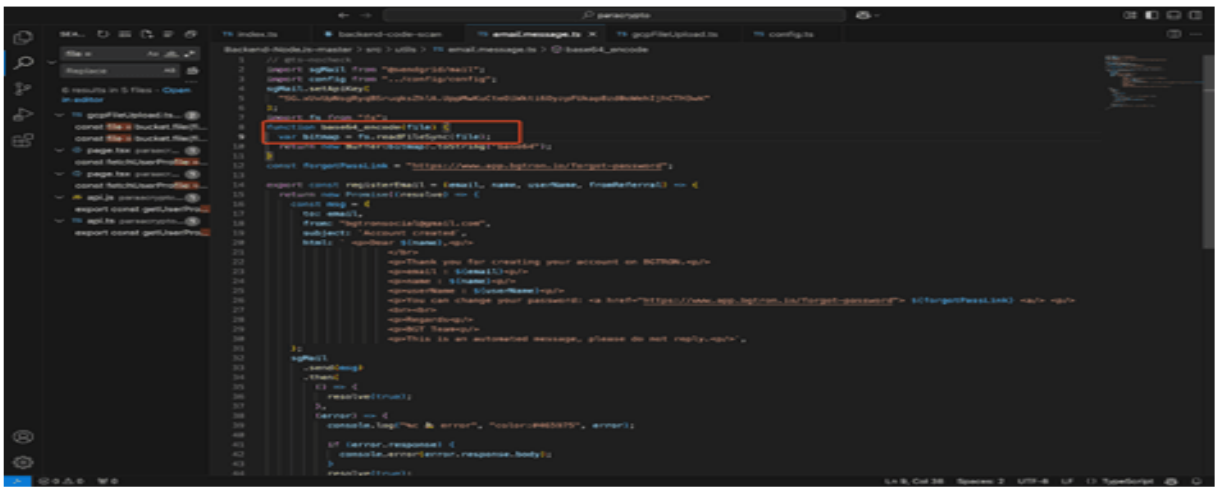
Resolved

Description

Allowing unsanitized dynamic input in file paths can lead to unauthorized file and folder access. This vulnerability arises when dynamic data is used within the file system operations, potentially allowing attackers to access unauthorized or hidden files and folders.
Vulnerable File :src/utils/email.message.ts

Recommendation

Do sanitize all dynamic data and function arguments before using them in file system operations. This step is crucial to prevent unauthorized access.
Do use a combination of hard-coded string literals and control logic, instead of directly passing dynamic data or function arguments to file system functions. This ensures safety.



Impact

Attacker can input malicious value as filename which could lead to path traversal vulnerability.

Missing Helmet Configuration on HTTP Headers

Resolved

Description

Helmet can help protect your app from some well-known web vulnerabilities by setting HTTP headers appropriately. Failing to configure Helmet for HTTP headers leaves your application vulnerable to several web attacks

Vulnerable File :src/index.ts

Recommendation

Do use Helmet middleware to secure your app by adding it to your application's middleware.

const helmet = require("helmet");
app.use(helmet())

Impact

There are certain web application attacks such as Clickjacking and XSS that can be exploited by header misconfigurations.

Leakage of Sensitive Data in JWT

Resolved

Description

Storing sensitive data in JWTs exposes it to potential security risks. JWTs are designed for transmitting data securely among parties but are not inherently secure storage for sensitive information.

Vulnerable File :src/admin/controller/auth.controller.ts
src/user/controller/auth.controller

Recommendation

Attackers can extract sensitive data such as user email, name, by base64 decoding stolen or breached JWT tokens leading to data breach.
Attackers can also create new JWT tokens by easily gaining access

Deprecated Package in use

Resolved

Description

The application includes one or more deprecated packages in package.json. Deprecated packages needs updates, making the application vulnerable to known exploits, compatibility issues, and potential supply chain attacks. Attackers could exploit unpatched vulnerabilities in outdated dependencies, leading to Remote Code Execution (RCE), Denial of Service (DoS), or data breaches.

Vulnerable File: package.json
Vulnerable Package:
Mongoose
braces
esbuild
micromatch
tough-cookie

Recommendation

Run the following command npm outdated and Update the packages.

Closing Summary

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

Some issues of High, medium, and low severity were found.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their Application 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 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 secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

March, 2025

For

Paraa Crypto

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com