Cyberscope

A TAC Security Company

Audit Report

Celestia

November 2025

Repository https://github.com/HEIN-TECH/celestia.finance

Commit 0fcdd9c6293b533a5c811b549a3876e7204a5e76

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/HEIN-TECH/celestia.finance
Commit	0fcdd9c6293b533a5c811b549a3876e7204a5e76

Audit Updates

Initial Audit	03 Nov 2025

Findings Breakdown

Critical	1
Medium	3
Minor / Informative	5

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	3	0	0	0
Minor / Informative	5	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	ISTS	Insecure Session Token Storage	Unresolved
●	CSAC	Client Side Authentication Checks	Unresolved
●	LA	Layout Authorization	Unresolved
●	MRL	Missing Rate Limiting	Unresolved
●	ICC	Insecure CORS Configuration	Unresolved
●	MNTC	Magic Number Time Conversion	Unresolved
●	MIV	Missing Input Validation	Unresolved
●	PIAE	Potential IP Address Exposure	Unresolved
●	SESE	Sensitive Error Stack Exposure	Unresolved

ISTS - Insecure Session Token Storage

Criticality	Critical
Location	src/frontend/src/lib/sessionManager.ts#L44
Status	Unresolved

Description

The SessionManager class stores the user's session token in localStorage via the `setSession()` method. Local storage is accessible to all JavaScript running in the same origin, including potentially malicious scripts injected via Cross-Site Scripting (XSS) vulnerabilities. This means that if an attacker successfully injects JavaScript into the application, they can easily retrieve the session token and hijack the user's session. Furthermore, tokens stored in local storage are sent manually via client-side logic, bypassing built-in browser protections such as the HttpOnly flag available in cookies.

```
public setSession(sessionToken: string, userId: string): void {
  this.sessionToken = sessionToken;
  this.userId = userId;

  if (typeof window !== 'undefined') {
    localStorage.setItem(SESSION_STORAGE_KEYS.SESSION_TOKEN, sessionToken);
    localStorage.setItem(SESSION_STORAGE_KEYS.USER_ID, userId);

    // Set expiry time (30 days from now)
    const expiryTime = new Date();
    expiryTime.setDate(expiryTime.getDate() + 30);
    localStorage.setItem(SESSION_STORAGE_KEYS.SESSION_EXPIRY,
expiryTime.toISOString());
  }
}
```

Recommendation

Store session tokens securely using HTTP-only cookies instead of local storage. HTTP-only cookies are not accessible via JavaScript, significantly reducing the risk of token theft through XSS.

- Use secure cookies with the following attributes:
- httpOnly: true → prevents JavaScript access
- secure: true → ensures cookies are sent only over HTTPS
- sameSite: 'strict' or 'lax' → mitigates CSRF risks
- Handle authentication and session validation on the server side, issuing and verifying cookies automatically with each request.
- Remove all session management logic that depends on local storage (localStorage.setItem / getItem / removeItem).

By moving session storage to secure cookies, the application protects session data from client-side attacks and ensures compliance with best practices for authentication management.

CSAC - Client Side Authentication Checks

Criticality	Medium
Location	src/frontend/src/app/admin/dashboard/layout.tsx#L21
Status	Unresolved

Description

The authentication and authorization logic is executed entirely on the client using `fetch('/api/admin/auth/verify')` and local state (useState, useEffect). This approach exposes authentication flow details to end users and allows potential attackers to bypass access control by directly calling protected APIs or manipulating client-side state. Since client-side code runs in an untrusted environment, relying solely on it for authentication verification compromises the security of protected routes and sensitive data.

```
useEffect(() => {
  const checkAuth = async () => {
    try {
      const response = await fetch('/api/admin/auth/verify');
      const data = await response.json();

      if (data.authenticated) {
        setIsAuthenticated(true);

        // Update last login time in localStorage for 24-hour tracking
        const now = Date.now();
        localStorage.setItem('admin-last-login', now.toString());
      } else {
        // Check if within 24 hours from last login
        const lastLogin = localStorage.getItem('admin-last-login');
        const now = Date.now();
        const twentyFourHours = 24 * 60 * 60 * 1000;

        if (lastLogin && (now - parseInt(lastLogin)) < twentyFourHours) {
          // Still within 24 hours, but token expired - redirect to login
          console.log('Session expired but within 24 hours, redirecting to
login...');
        } else {
          // More than 24 hours, clear login time
          localStorage.removeItem('admin-last-login');
        }

        router.push('/admin/login');
      }
    } catch (error) {
      console.error('Auth check failed:', error);
      router.push('/admin/login');
    }
  };

  checkAuth();
}, [router]);
```

Recommendation

Move all authentication and authorization checks to the server side to ensure secure and consistent enforcement. The server should validate user sessions or tokens before rendering protected pages or returning data.

- Use Next.js Server Components, Middleware, or API route guards to perform authentication checks before the response is sent to the client.
- Only send minimal, non-sensitive user data to the client after authentication succeeds.
- The client should consume the authenticated state provided by the server, rather than determining it locally.

This ensures that authentication is securely validated on the server before rendering, preventing unauthorized access through client-side manipulation.

LA - Layout Authorization

Criticality	Medium
Location	src/frontend/src/app/admin/dashboard/layout.tsx#L21
Status	Unresolved

Description

The dashboard layout component performs authentication and admin checks directly within the layout using client-side logic. According to the Next.js documentation, layouts are designed to render only once and persist across route transitions. As a result, authentication checks implemented here will not re-run when navigating between pages within the dashboard, potentially leading to stale or invalid authentication states. This approach also exposes client-side validation that can be bypassed, reducing the overall security of the admin area.

```
useEffect(() => {
  const checkAuth = async () => {
    try {
      const response = await fetch('/api/admin/auth/verify');
      const data = await response.json();

      if (data.authenticated) {
        setIsAuthenticated(true);

        // Update last login time in localStorage for 24-hour tracking
        const now = Date.now();
        localStorage.setItem('admin-last-login', now.toString());
      } else {
        // Check if within 24 hours from last login
        const lastLogin = localStorage.getItem('admin-last-login');
        const now = Date.now();
        const twentyFourHours = 24 * 60 * 60 * 1000;

        if (lastLogin && (now - parseInt(lastLogin)) < twentyFourHours) {
          // Still within 24 hours, but token expired - redirect to login
          console.log('Session expired but within 24 hours, redirecting to
login...');
        } else {
          // More than 24 hours, clear login time
          localStorage.removeItem('admin-last-login');
        }

        router.push('/admin/login');
      }
    } catch (error) {
      console.error('Auth check failed:', error);
      router.push('/admin/login');
    }
  };

  checkAuth();
}, [router]);
```

Recommendation

The team is advised to move the authentication and authorization logic from the layout into the individual page components that require protection. This ensures that authentication is validated every time a page is loaded or rendered, maintaining consistent and secure access control.

- Perform the authentication and role verification inside server components within each protected page.
- Keep the layout focused solely on UI structure and navigation, without embedding access-control logic.

This approach ensures that each admin page independently enforces authentication and authorization on every render.

MRL - Missing Rate Limiting

Criticality	Medium
Location	src/frontend/src/app/api/admin/auth/login/route.ts#L5
Status	Unresolved

Description

The login API route (POST /api/admin/auth/login) does not implement any rate limiting or request throttling. Without such controls, the endpoint is vulnerable to brute-force and credential-stuffing attacks, where an attacker can repeatedly attempt different username–password combinations to gain unauthorized access. This can lead to account compromise, service degradation, or denial-of-service (DoS) conditions if abused at scale.

```
export async function POST(request: NextRequest) {
  try {
    const body = await request.json();
    const { username, password } = body;

    // Validate input
    if (!username || !password) {
      return NextResponse.json(
        { success: false, message: 'Username and password are required' },
        { status: 400 }
      );
    }

    // Verify credentials
    const isValid = verifyCredentials(username, password);
    ...
  } catch (error: any) {
    console.error('Login error:', error);
    return NextResponse.json(
      { success: false, message: 'Internal server error', details: error.message },
      { status: 500 }
    );
  }
}
```

Recommendation

Implement rate limiting on the login route to restrict the number of authentication attempts per IP address or user within a given time window. This effectively mitigates brute-force and automated login attacks.

- Use a rate-limiting solution such as Upstash Redis, @vercel/ratelimit, or express-rate-limit (if using Express).
- Apply a strict policy, e.g., allowing 5 login attempts per minute per IP.
- Return a standardized error response (e.g., HTTP 429) when the limit is exceeded.
- Optionally, introduce exponential backoff or temporary IP blacklisting for repeated violations.

Implementing rate limiting ensures fair usage, improves system resilience, and strengthens authentication security.

ICC - Insecure CORS Configuration

Criticality	Minor / Informative
Location	src/backend/main.py#L41
Status	Unresolved

Description

The CORS middleware is configured to allow all origins (*) via the allow_origins parameter. This configuration effectively permits any domain to send cross-origin requests to the API, which can expose sensitive data to unauthorized third-party websites. When combined with `allow_credentials=True`, it creates a severe security risk — attackers could execute Cross-Site Request Forgery (CSRF) or data exfiltration attacks from untrusted origins, as cookies and authentication tokens might be automatically sent with the requests.

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=ALLOWED_ORIGINS,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
    expose_headers=["*"],  # 暴露所有响应头
    max_age=3600,  # 预检请求缓存时间
)
```

Recommendation

Restrict CORS access to trusted domains only. Avoid using wildcard (*) origins, especially when allow_credentials=True is enabled.

- Replace `allow_origins=["*"]` with a list of known and trusted frontend domains
- Keep allow_credentials=True only if absolutely necessary for authenticated cross-origin requests.
- If the API is public and must support multiple origins, implement dynamic origin validation (check the Origin header against an approved whitelist at runtime).

Properly restricting CORS origins ensures that only authorized web applications can interact with the API, reducing the risk of cross-site attacks.

MNTC - Magic Number Time Conversion

Criticality	Minor / Informative
Location	src/frontend/src/lib/utils.ts#L10
Status	Unresolved

Description

The function uses the hardcoded value `1e12` to determine whether a timestamp is in seconds rather than milliseconds. This magic number lacks context, making the code less readable and harder to maintain. Without a clear explanation, future developers may not understand its purpose or origin.

```
export const getElapsedTimeLabel = (timestamp: number) => {
  if (timestamp < 1e12) {
    timestamp *= 1000;
  }

  const now = Date.now();
  const diffMs = now - timestamp;
  const diffMin = Math.floor(diffMs / (1000 * 60));
  const diffH = Math.floor(diffMin / 60);
  const diffD = Math.floor(diffH / 24);
  const diffW = Math.floor(diffD / 7);

  if (diffW >= 1) {
    return `${diffW}w`;
  } else if (diffD >= 1) {
    return `${diffD}d`;
  } else if (diffH >= 1) {
    return `${diffH}h`;
  } else {
    return `${diffMin}min`;
  }
};
```

Recommendation

Define named constants such as `SECONDS_TO_MILLISECONDS_THRESHOLD` and include comments that clarify the purpose and logic behind the conversion. This improves code readability and helps future developers understand the rationale behind the threshold values.

MIV - Missing Input Validation

Criticality	Minor / Informative
Location	src/frontend/src/app/api/invite/verify/route.ts#L71 src/backend/email_router.py#L10,91,109,127
Status	Unresolved

Description

The request body is parsed directly without any form of validation, which can lead to unexpected behavior or security vulnerabilities if the input is malformed or malicious.

```
const body = await request.json();
const { code, userEmail } = body;
```

```
customer_task = asyncio.create_task(
    email_service.send_customer_email(email_data)
)
admin_task = asyncio.create_task(
    email_service.send_admin_notification(email_data)
)
customer_result, admin_result = await asyncio.gather(
    customer_task, admin_task, return_exceptions=True
)
```

Recommendation

By implementing request validation, all input parameters can be rigorously checked for type correctness and required structure. This ensures that only well-formed data is processed by the application, reducing the risk of runtime errors and unexpected behavior.

Additionally, sanitizing user input helps prevent injection attacks and other forms of malicious input, enhancing the overall security and reliability of the system.

PIAE - Potential IP Address Exposure

Criticality	Minor / Informative
Location	src/frontend/next.config.ts#L15
Status	Unresolved

Description

The `allowedDevOrigins` array includes production IP addresses, which may inadvertently expose development-only endpoints in a live environment. This configuration increases the risk of unauthorized access to internal tools or untested features, potentially compromising application security.

```
const nextConfig: NextConfig = {
  distDir: 'hope-fund-frontend',
  reactStrictMode: true,

  transpilePackages: [
    '@ant-design',
    'antd',
    'rc-util',
    'rc-pagination',
    'rc-picker',
  ],

  allowedDevOrigins: [
    'https://45.77.170.234:3000',
    'https://45.77.170.234',
    'http://localhost:3000',
    'http://127.0.0.1:3000',
    'http://192.168.50.14:3000',
  ],
  ...
};
```

Recommendation

The team is advised to remove production IP addresses from development origins. This helps prevent unintended access to production resources during development.

Additionally, using environment-based configuration ensures that settings are appropriately tailored for development, staging, or production environments, reducing the risk of misconfiguration.

Finally, dev-only features should be properly gated to prevent exposure in production, enhancing security and maintaining a clean user experience.

SESE - Sensitive Error Stack Exposure

Criticality	Minor / Informative
Location	src/frontend/src/app/api/basic-info/route.ts#L204
Status	Unresolved

Description

The API route returns the full `error.stack` trace in its JSON response when an exception occurs. This exposes internal implementation details — such as file paths, function names, and potentially sensitive data — to clients. Attackers can use this information to map the server structure, identify frameworks or libraries in use, and craft targeted exploits. Exposing stack traces is a common security misconfiguration that violates secure error-handling best practices.

```
console.error('❌ API Error:', error);
return NextResponse.json(
  {
    error: 'Failed to load vault data',
    details: error.message,
    stack: error.stack
  },
  { status: 500 }
);
```

Recommendation

The team is advised to remove the `stack` and, ideally, the `details` fields from production error responses. Full error information should be logged securely on the server side using a reliable logging system such as Winston, Pino, or Sentry. Only a generic, non-sensitive error message should be returned to the client to prevent exposure of internal implementation details.

Summary

Celestia implements a backend and frontend mechanism. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a TAC blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

A **TAC Security** Company

The Cyberscope team

cyberscope.io