Cyberscope

A TAC Security Company

Audit Report

Plutus

August 2025

Repository	https://github.com/PlutusDao/berancia
Commit	61045dcce9a1005e4c8ec527522555bb005e59f8
Audited by	© cyberscope

Table of Contents

Table of Contents **1**
Risk Classification **3**
Review **4**
Audit Updates 4
Overview **5**
Orange 5
Plutus 6
Code Duplication Comment 6
Findings Breakdown **7**
Diagnostics **8**
CR - Code Repetition 9
Description 9
Recommendation 10
EUOAT - Excessive Use of ANY Type 11
Description 11
Recommendation 11
MEC - Missing ESLint Configuration 12
Description 12
Recommendation 12
MIV - Missing Input Validation 13
Description 13
Recommendation 13
PSDE - Potential Sensitive Data Exposure 14
Description 14
Recommendation 14
RVD - Redundant Variable Declaration 15
Description 15
Recommendation 15
SSR - Suboptimal Signature Recovery 16
Description 16
Recommendation 17
TID - Type Import Distinction 18
Description 18
Recommendation 18
UPKC - Unbounded Public Key Cache 19
Description 19
Recommendation 19
Summary **20**
Disclaimer **21**

About Cyberscope **22**

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/PlutusDao/berancia
Commit	61045dcce9a1005e4c8ec527522555bb005e59f8

Audit Updates

Initial Audit	19 Jun 2025
Corrected Phase 2	20 Aug 2025

Overview

Cyberscope conducted an audit of the `berancia` repository within the Plutus ecosystem. The repository consists of two projects. Together, Orange and Plutus offer a robust infrastructure: Orange focuses on orchestrating real-time on-chain actions, while Plutus enables the secure, standards-compliant construction of transactions and signature workflows.

Orange

Orange serves as the middleware backbone for client applications interacting with on-chain protocols on the Berachain network. Built with Node.js and Express, it functions as a centralized gateway that abstracts the complexity of raw JSON-RPC communication, offering a streamlined interface for both data retrieval and state-changing operations. Whether it's querying balances and rewards or executing deposits, withdrawals, and emergency exits, Orange handles these actions on behalf of front-end dashboards, automation layers, and external integrations.

At its core, Orange manages communication with Berachain nodes through configurable RPC endpoints, incorporating reconnection strategies and fallback logic to ensure resilience. It interfaces directly with the Bearn Protocol's smart contracts, overseeing aspects such as gas estimation, nonce tracking, and dynamic fee calculations. Its RESTful API surface can be consumed by a variety of clients, ranging from CLI tools to web-based dashboards and scheduled job runners.

Operationally, Orange emphasizes environment separation and observability. Configuration is centralized through environment variables or CLI-driven config files, simplifying deployment across development, staging, and production. Middleware components take care of cross-origin resource sharing, request logging, rate limiting, and consistent error formatting—creating a standardized and predictable interface for client systems. The service is stateless by design, enabling horizontal scaling behind load balancers and facilitating coordination through shared Redis caches or message queues.

Plutus

Plutus, on the other hand, is a TypeScript-based monorepo tailored for constructing and managing EIP-712 typed-data transactions. It plays a critical role in multi-signature workflows, especially within protocols that require off-chain signature aggregation prior to on-chain execution, such as those using Gnosis Safe.

Plutus generates domain-specific payloads conforming to the EIP-712 standard, enabling transaction clarity for both users and wallets. These payloads are validated against strict TypeScript interfaces to catch discrepancies during development, ensuring a more secure and predictable deployment pipeline. The monorepo wraps the Safe Protocol Kit, streamlining the full lifecycle of a multi-sig transaction—from proposal creation to signature collection and execution. It supports multiple signer backends, accommodating various setups including Web3 wallets, hardware devices, and custodial signers.

To manage complexity and optimize performance, Plutus loads large ABI files dynamically as separate JSON artifacts. This not only reduces memory usage but also supports automated code generation for type-safe contract bindings. It provides reusable schema modules for common DeFi operations such as swaps, staking, and liquidity provision, ensuring transactional consistency across the ecosystem.

In terms of deployment, Plutus includes lightweight serverless handlers, deployable on platforms like AWS Lambda, that expose minimal HTTP endpoints suitable for integration with front-end applications or orchestration layers. It’s also distributed as a set of internal NPM packages, making its components modular and composable. Whether it's typed-data generation, signer abstraction, or ABI management, each module can be selectively integrated as needed.

Code Duplication Comment

The two projects contain duplicated or closely related code segments, indicating an opportunity for reuse through modularization. Extracting these shared components into a standalone NPM library or a shared internal package within a monorepo would improve maintainability, reduce redundancy, and ensure consistency across both codebases. This approach promotes code reuse, simplifies updates, and fosters a cleaner architecture by centralizing logic that is common to multiple services or applications.

Findings Breakdown

Critical	0
Medium	0
Minor / Informative	9

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	4	5	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	CR	Code Repetition	Acknowledged
●	EUOAT	Excessive Use of ANY Type	Unresolved
●	MEC	Missing ESLint Configuration	Acknowledged
●	MIV	Missing Input Validation	Unresolved
●	PSDE	Potential Sensitive Data Exposure	Acknowledged
●	RVD	Redundant Variable Declaration	Unresolved
●	SSR	Suboptimal Signature Recovery	Acknowledged
●	TID	Type Import Distinction	Unresolved
●	UPKC	Unbounded Public Key Cache	Acknowledged

CR - Code Repetition

Criticality	Minor / Informative
Location	orange/app/api/v1/strategies/manage/claimAll/route.ts#L12 orange/app/api/v1/strategies/manage/compoundToPoolToken/route.ts#L13 orange/app/api/v1/strategies/manage/exitAll/route.ts#L11 orange/app/api/v1/strategies/manage/fetchFurthermoreData/route.ts#L11 orange/app/api/v1/strategies/manage/returnToVault/route.ts#L13 orange/app/api/v1/strategies/manage/stakeAll/route.ts#L13
Status	Acknowledged

Description

The codebase contains repetitive code segments. There are potential issues that can arise when using code segments in Javascript. Some of them can lead to issues like efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
const authHeader = request.headers.get('authorization') ||
request.headers.get('Authorization')
const expectedKey = process.env.ORANGE_API_KEY

if (!authHeader || !authHeader.startsWith('Bearer ') || !expectedKey) {
    return new Response(JSON.stringify({ error: 'Unauthorized' }), {
        status: 401,
        headers: { 'Content-Type': 'application/json' },
    })
}

const token = authHeader.replace('Bearer ', '').trim()
if (token !== expectedKey) {
    return new Response(JSON.stringify({ error: 'Unauthorized' }), {
        status: 401,
        headers: { 'Content-Type': 'application/json' },
    })
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the codebase easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the codebase. For instance, the team could reuse the common code segments in a separate module that exports the common code segments in order to avoid repeating the same code in multiple places.

EUOAT - Excessive Use of ANY Type

Criticality	Minor / Informative
Location	orange/src/services/aws/kms/utils.ts#L7,23,83 orange/src/services/aws/kms/signer.ts#L56,111,135 orange/src/services/furthermore/api.service.ts#L8 orange/src/services/kodiak/api.service.ts#L19 plutus/packages/functions/src/aws/signer.ts#L56,112,136 plutus/packages/functions/src/aws/utils.ts#L7,24,84 plutus/packages/functions/src/safe/confirmAndExecute.ts#L38,151
Status	Unresolved

Description

The codebase includes several instances where the `any` type is used to declare variables or parameters, implicitly or explicitly. While TypeScript provides the flexibility to use the `any` type when the type is not precisely known or needs to be intentionally left open, excessive use of `any` can undermine the benefits of static typing. Overusing `any` diminishes the advantages of TypeScript, as it removes the benefits of type checking and type safety, making the code more error-prone and less maintainable.

```
unsignedTx: any
opts: any
message: any
data: any
error: any
```

Recommendation

The team is advised to replace `any`, whenever possible, with precise types to provide clarity about the expected data structure or format. By creating explicit type definitions for objects, functions, and parameters, the team could ensure robust type checking and improved code readability. Union types or generics could be used to handle scenarios where a variable can have multiple types, maintaining the advantages of type safety. By minimizing the use of the `any` type and leveraging TypeScript's strong typing features, the codebase can benefit from improved developer experience, better IDE support, and enhanced code maintainability.

MEC - Missing ESLint Configuration

Criticality	Minor / Informative
Status	Acknowledged

Description

The codebase lacks an ESLint configuration, which is a valuable tool for identifying and fixing issues in JavaScript code. ESLint not only helps catch errors but also enforces a consistent code style and promotes best practices. It can significantly enhance code quality and maintainability by providing a standardized approach to coding conventions and identifying potential problems.

Recommendation

The team is strongly advised to integrate ESLint into the project by creating an ESLint configuration file (e.g., `.eslintrc.js` or `.eslintrc.json`) and defining rules that align with the team's coding standards. Consider using popular ESLint configurations, such as Airbnb, Standard, or your own customized set of rules. By incorporating ESLint into the project, the team ensures consistent code quality, catches potential problems early in the development process, and establishes a foundation for collaborative and maintainable code.

MIV - Missing Input Validation

Criticality	Minor / Informative
Location	plutus/packages/functions/src/safe/confirmAndExecute.ts#L26 orange/src/services/kodiak/api.service.ts#L66 orange/src/services/oogabooga/api.service.ts#L34,46 orange/app/api/v1/strategies/manage/claimAll/route.ts#L32 orange/app/api/v1/strategies/manage/compoundToPoolToken/route.ts#L33 orange/app/api/v1/strategies/manage/exitAll/route.ts#L31 orange/app/api/v1/strategies/manage/fetchFurthermoreData/route.ts#L31
Status	Unresolved

Description

External inputs, such as transaction hashes and API responses, are used without schema or format validation. This exposes the application to potential runtime errors, data inconsistencies, or unexpected behavior—especially when interacting with on-chain logic or third-party services. In the example below, data from an external API is used directly without verifying structure or content.

```
let safeMultisigTx = await apiKit.getTransaction(hash)

const body = await request.json()
const { safeAddress, poolAddress, nonce }: IManagePoolRequest = body
...
```

Recommendation

It is always a good practice to validate all external inputs and API responses against expected schemas or type definitions before use. This includes checks for presence, type, and format (e.g., using regex for hashes or schema validation libraries). Proper validation mitigates risks related to malformed data, unexpected behavior, or exploitation via crafted inputs.

PSDE - Potential Sensitive Data Exposure

Criticality	Minor / Informative
Location	plutus/packages/functions/src/safe/confirmAndExecute.ts#L84,109
Status	Acknowledged

Description

The code logs raw signatures, transaction payloads, and AWS KMS interactions (e.g. `execTransactionEncoded, receipt`). Printing these to console may leak cryptographic material in CloudWatch or local logs.

```
logger.info(`Encoded transaction: ${execTransactionEncoded}`);
logger.info(`Transaction executed successfully. Receipt: ${safeStringify(receipt)}`);
```

Recommendation

The team is advised to remove or redact sensitive fields from logs and log only high-level statuses or non-secret identifiers. This could be easily implemented by introducing a centralized logging service as described in the MCLS section.

RVD - Redundant Variable Declaration

Criticality	Minor / Informative
Location	orange/src/services/blockchain/claim.rewards.service.ts#L1 orange/src/services/oogabooga/api.service.ts#L1,5
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The codebase declares certain variables that are not used in a meaningful way. As a result, these variables are redundant.

```
Address
formatUnits
USDC
```

Recommendation

The team is advised to remove any unnecessary variables to clean up the code. If they are meant for future usage, the team could prefix them with _ (e.g., _actions) to indicate intentional non-use. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

SSR - Suboptimal Signature Recovery

Criticality	Minor / Informative
Location	plutus/packages/functions/src/aws/signer.ts#L93,118,143,171 orange/src/services/aws/kms/signer.ts#L72,92,117,142,170
Status	Acknowledged

Description

The signature recovery logic relies on a brute-force trial of both possible `yParity` values (0 and 1), with no structured error handling or fallback mechanism if recovery fails. This approach introduces inefficiency and may silently fail in edge cases, especially if neither attempt succeeds or if an unrelated error occurs during processing.

```
for (const recovery of [0, 1]) {
    const chainId = unsignedTx.chainId
    const v =
        typeof chainId === 'bigint' && chainId > 0n
            ? 35n + chainId * 2n + BigInt(recovery)
            : 27n + BigInt(recovery)
    try {
        const serializedSig = serializeSignature({ r: toHex(r), s: toHex(s), v })

        const recovered = await recoverAddress({ hash: digest, signature:
serializedSig })

        if (recovered.toLowerCase() === address.toLowerCase()) {
            return serializer(unsignedTx, { r: toHex(r), s: toHex(s), v })
        }
    } catch {
        continue
    }
}
...
```

Recommendation

To mitigate this issue, the team could replace the trial-and-error logic with a deterministic recovery method when possible, or enhance error handling to explicitly log or propagate failures. If brute-force is necessary, the team could handle and report failures clearly to avoid silent degradation and improve debuggability.

TID - Type Import Distinction

Criticality	Minor / Informative
Location	orange/src/services/aws/kms/utils.ts#L1 orange/src/services/blockchain/claim.rewards.service.ts#L1,2 orange/src/services/blockchain/deposit.tokens.service.ts#L5 orange/src/services/blockchain/helpers.service.ts#L1,5 orange/src/services/blockchain/mint.tokens.service.ts#L1 orange/src/services/blockchain/stake.tokens.service.ts#L1,9 orange/src/services/blockchain/swap.tokens.service.ts#L4,5 orange/src/services/safe/sample/sample.ts#L1
Status	Unresolved

Description

The current codebase imports multiple types from the `types.ts` file without explicitly using the `type` keyword, as provided by the TypeScript compiler. This may introduce ambiguity in distinguishing between regular imports and type imports, impacting code clarity and maintainability consistency.

```
KMSClient
Address
Hex
MetaTransactionData
```

Recommendation

For improved code clarity and consistency, the team is advised to explicitly use the `type` keyword when importing types or interfaces. This practice enhances readability and ensures a clear distinction between regular imports and type imports, contributing to a more maintainable and comprehensible codebase.

UPKC - Unbounded Public Key Cache

Criticality	Minor / Informative
Location	orange/src/services/aws/kms/utils.ts#L5 plutus/packages/functions/src/aws/utils.ts#L5
Status	Acknowledged

Description

The `CACHE` map stores raw public keys retrieved from AWS KMS without any eviction policy or size constraint. In long-running services or high-throughput systems using many distinct KMS keys, this unbounded caching can lead to memory bloat and degrade application performance over time.

```
const CACHE = new Map<string, Uint8Array<ArrayBuffer>>()
```

Recommendation

To mitigate this issue, it is recommended to implement a bounded caching strategy, such as an LRU (Least Recently Used) cache or time-based expiration, to prevent unbounded memory growth. This ensures efficient memory usage while retaining the performance benefits of caching frequently accessed keys.

Summary

Plutus implements a frontend and backend 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