Cyberscope

A TAC Security Company

Audit Report

Plutus

June 2025

Repository https://github.com/PlutusDao/berancia

Commit f92d08986e903c462dc545007efa0f979a7c68f6

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/PlutusDao/berancia
Commit	f92d08986e903c462dc545007efa0f979a7c68f6

Audit Updates

Initial Audit	19 Jun 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.

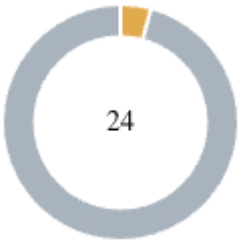
To manage complexity and optimize performance, Plutus loads large ABI files dynamically as separate JSON artifacts. This not only reduces memory usage but also supports automated code generation for type-safe contract bindings. It provides reusable schema modules for common DeFi operations such as swaps, 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	1
Minor / Informative	23

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	23	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	UJP	Unchecked JSON Parsing	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	DLS	Duplicate Logging Systems	Unresolved
Minor / Informative	EIP	Excessive IAM Permissions	Unresolved
Minor / Informative	EUOAT	Excessive Use of ANY Type	Unresolved
Minor / Informative	FGLWC	Fallback Gas Limit Without Context	Unresolved
Minor / Informative	HRA	Hard-Coded Resource ARNs	Unresolved
Minor / Informative	ICL	Inconsistent Crypto Libraries	Unresolved
Minor / Informative	ITD	Incorrect Type Definition	Unresolved
Minor / Informative	IEVV	Insufficient Environment Variables Validation	Unresolved
Minor / Informative	MCLS	Missing Centralized Logging Service	Unresolved
Minor / Informative	MEC	Missing ESLint Configuration	Unresolved
Minor / Informative	MIV	Missing Input Validation	Unresolved
Minor / Informative	MLFD	Multiple Lock Files Detected	Unresolved

●	PSDE	Potential Sensitive Data Exposure	Unresolved
●	PFEV	Private Field Encapsulation Vulnerability	Unresolved
●	RVD	Redundant Variable Declaration	Unresolved
●	SIE	Server Information Exposure	Unresolved
●	SVDC	State Variables could be Declared Constant	Unresolved
●	SSR	Suboptimal Signature Recovery	Unresolved
●	TID	Type Import Distinction	Unresolved
●	UPKC	Unbounded Public Key Cache	Unresolved
●	UEE	Unclassified External Errors	Unresolved
●	UCC	Unsafe CORS Configuration	Unresolved

UJP - Unchecked JSON Parsing

Criticality	Medium
Location	orange/src/services/kodiak/api.service.ts#L25 orange/src/services/oogabooga/api.service.ts#L30,43
Status	Unresolved

Description

The response from `fetch` is parsed as JSON without verifying that the request was successful. If the upstream API returns a non-2xx status, the `.json()` call may either fail or return an unexpected structure, potentially leading to misleading behavior or unhandled exceptions.

```
const res = await fetch(publicApiUrl);
return await res.json(); // No status check
```

Recommendation

The team is recommended to check the response status before attempting to parse the body. Only parse the response as JSON when the status code indicates success (e.g., 2xx range). This ensures that errors are not silently ignored and enables proper handling of failed or malformed upstream responses.

CR - Code Repetition

Criticality	Minor / Informative
Location	orange/src/controllers/strategies.controller.ts#L57,66,75 plutus/packages/functions/src/aws/signer.ts#L117,143,173 orange/src/services/aws/kms/signer.ts#L89,114,140
Status	Unresolved

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 iBGTBalance = await getBalance(stakingAddresses[poolAddress].infrared)
if (iBGTBalance > 0n) {
    const safeTx = await exitInfrared(stakingAddresses[poolAddress].infrared)
    safeBatchTxs.push(safeTx)
} else {
    console.log(`No iBGT balance found for vault
${stakingAddresses[poolAddress].infrared}`)
}
...
```

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.

DLS - Duplicate Logging Systems

Criticality	Minor / Informative
Location	orange/src/config/winston.ts orange/src/utils/logger.ts
Status	Unresolved

Description

The codebase defines two separate logging mechanisms: a custom JSON-based logger (`logger.ts`) and a Winston-based logger (`winston.ts`). This duplication results in inconsistent log formats, fragmented usage across files, and increased maintenance complexity. The custom logger uses direct console output with manual formatting, while Winston provides structured, extensible logging with built-in transport and formatting options.

Recommendation

The team is strongly advised to consolidate logging to a single implementation, preferably the more robust and configurable Winston logger. The team could ensure all application components use a unified logger to maintain consistency in log structure, support external log aggregation, and reduce cognitive overhead for debugging and maintenance.

EIP - Excessive IAM Permissions

Criticality	Minor / Informative
Location	plutus/stacks/MyStack.ts#L24
Status	Unresolved

Description

In `MyStack.ts`, the cron job binds a policy with `actions: ['kms:*']` against the entire KMS key. Granting all KMS actions may violate least-privilege principles and increase blast radius if the function is compromised.

```
new PolicyStatement({
    effect: Effect.ALLOW,
    actions: ['kms:*'],
    resources: [KMS_KEY_ARN],
})
```

Recommendation

The team is advised to restrict the IAM policy to only the specific KMS actions required (e.g. Sign, GetPublicKey), and scope the resource ARN narrowly.

EUOAT - Excessive Use of ANY Type

Criticality	Minor / Informative
Location	plutus/packages/functions/src/safe/utils.ts#L10 plutus/packages/functions/src/safe/confirmAndExecute.ts#L71,78 plutus/packages/functions/src/aws/signer.ts#L56,111,136 orange/src/services/safe/utils.ts#L10 orange/src/services/aws/kms/signer.ts#L53,108,133
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.

```
safeTx: any
confirmations: any
signature: any
unsignedTx: any
opts: any
message: any
data: 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.

FGLWC - Fallback Gas Limit Without Context

Criticality	Minor / Informative
Location	plutus/packages/functions/src/aws/signer.ts#L245
Status	Unresolved

Description

In the `sendTransaction` method of the Signer class, when gas estimation fails, a hard-coded fallback of `1_000_000n` is used. This may lead to over-paying or transaction failure on networks with different gas requirements.

```
let finalGas = gas
if (!finalGas) {
    try {
        finalGas = await client.estimateGas({
            ...txBase,
            account,
        })
    } catch (e) {
        console.warn('Gas estimation failed, falling back to default gas limit', e)
        finalGas = 1_000_000n // reasonable fallback
    }
}
```

Recommendation

To mitigate this issue, the team is recommended to use a configurable or dynamically calculated fallback limit based on chain parameters, with monitoring/alerts for repeated estimation failures.

HRA - Hard-Coded Resource ARNs

Criticality	Minor / Informative
Location	plutus/stacks/MyStack.ts#L5
Status	Unresolved

Description

The `KMS_KEY_ARN` is inlined in `MyStack.ts` instead of being injected or stored in a parameter store or secret. Hard-coding resource identifiers reduces flexibility across environments and risks drift.

```
const KMS_KEY_ARN = 'arn:aws:kms:eu-north-1:*********:key/******************'
```

Recommendation

It is recommended to externalize ARNs to SST `Config.Secret` or parameters so they can vary per-environment and avoid embedding sensitive identifiers in source.

ICL - Inconsistent Crypto Libraries

Criticality	Minor / Informative
Location	orange/src/services/aws/kms/signer.ts#L5,6 plutus/packages/functions/src/aws/signer.ts#L5,6
Status	Unresolved

Description

The codebase uses cryptographic functions from both `ethers` and `viem`, such as hashing, signing, and address recovery. These libraries may implement slightly different formats or assumptions (e.g., EIP-191 vs. raw hashes), leading to signature incompatibilities, verification failures, or subtle bugs—especially when cross-validating signatures or interacting with external systems.

```
import { ethers } from 'ethers';
import {
  hashMessage,
  hashTypedData,
  recoverAddress,
  // ... other viem utilities
} from 'viem';
```

Recommendation

The team is strongly advidsed tp use a single cryptographic library consistently for all signing, hashing, and recovery operations. This reduces complexity, ensures compatibility across components, and prevents issues stemming from mismatched data formats or differing implementation details.

ITD - Incorrect Type Definition

Criticality	Minor / Informative
Location	orange/src/services/oogabooga/api.service.ts#L16
Status	Unresolved

Description

The type `Number` is used instead of the primitive `number`. In TypeScript, `Number` refers to the wrapper object, not the primitive type. This can lead to subtle bugs, as wrapper objects behave differently—for example, `new Number(5)` is truthy even when wrapping a falsy value, and object references are compared by identity rather than value.

```
type SwapParams = {
    tokenIn: Address,
    tokenOut: Address,
    amount: bigint,
    to: Address,
    slippage: Number,
};
```

Recommendation

The team is advised to use primitive types (`number` , `string` , `boolean` , etc.) instead of their object counterparts (`Number` , `String` , `Boolean` , etc.) in type definitions. This ensures consistent value semantics, prevents unexpected behavior, and aligns with TypeScript best practices.

IEVV - Insufficient Environment Variables Validation

Criticality	Minor / Informative
Location	orange/src/config/dotenv.config.ts#L41 plutus/packages/functions/src/config/dotenv.config.ts#L18
Status	Unresolved

Description

The `Config` class verifies the presence of required environment variables but does not validate their format, type, or range. This creates a blind spot where improperly formatted values (e.g. invalid URLs, malformed API keys) can pass validation but lead to runtime errors or misconfigurations in downstream services.

```
const missingVars = requiredVars.filter((varName) => !process.env[varName])

if (missingVars.length > 0) {
    logger.error(
        'Environment variables validation failed',
        new Error(`Missing required environment variables: ${missingVars.join(',
')}`)
    )
    throw new Error(`Missing required environment variables: ${missingVars.join(',
')}`)
}
```

Recommendation

The team is strongly recommended to mitigate this issue by implementing stricter validation of environment variables using a schema validation library (e.g. zod, joi, or envsafe). By ensuring each variable is checked for type correctness, valid format (e.g., URL, numeric, enum), and appropriate range or pattern, the team can improve reliability, prevent silent misconfigurations, and catch errors early during application startup.

MCLS - Missing Centralized Logging Service

Criticality	Minor / Informative
Location	orange/app/api/v1/example/route.ts#L7,42,54,61 orange/app/api/v1/strategies/manage/route.ts#L5,38 orange/src/index.ts#L26 orange/src/controllers/strategies.controller.ts#L62,71,80 plutus/packages/functions/src/aws/signer.ts#L221,244,249 plutus/packages/functions/src/monitor.execute.transactions/index.ts#L6,12,17,26
Status	Unresolved

Description

The project currently relies on console statements for debugging and logging information and errors during development. While this approach may be sufficient for local testing, it is not suitable for production environments. Console logs are often left in the code unintentionally, leading to cluttered outputs and potential exposure of sensitive information. Additionally, `console.log` lacks the ability to manage log levels, such as separating error logs from informational logs, and does not provide flexibility in log outputs (e.g., to files, external logging services, or different environments).

The following code segments are a sample of all the occurences.

```
console.log('Viem KMS Orange Signer Address:', address)
console.log({ structSignature })
console.log('txRequest type:', txRequest.type)
console.log('Signed Transaction:', signedTx)
console.log('finalGasPrice', finalGasPrice)
console.warn('Gas estimation failed, falling back to default gas limit', e)
console.log('finalGas', finalGas)
...
```

Recommendation

The team is strongly advised to integrate a configurable logging library such as `Winston`, `Bunyan`, or `Pino`. These libraries support various log levels (e.g., debug, info, warn, error) and allow routing logs to different destinations (e.g., console, files, external services). The team should audit the codebase to remove or replace all the console statements with the chosen logging library, ensuring that logs are structured, appropriately leveled, and routed according to the environment (e.g., more verbose logging in development, minimal logging in production). By implementing a structured and configurable logging solution, the project will benefit from more maintainable, secure, and effective logging practices, aiding both development and production operations.

MEC - Missing ESLint Configuration

Criticality	Minor / Informative
Status	Unresolved

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#L15,34 orange/src/services/kodiak/api.service.ts#L25 orange/src/services/oogabooga/api.service.ts#L30,43,61
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.

```
const safeMultisigTx = await apiKit.getTransaction(hash)
const execTransactionEncoded = encodeFunctionData({
    abi: SAFE_ABI,
    functionName: 'execTransaction',
    args: [
        safeMultisigTx.to,
        safeMultisigTx.value,
        '0x',
        safeMultisigTx.operation,
        safeMultisigTx.safeTxGas,
        safeMultisigTx.baseGas,
        safeMultisigTx.gasPrice,
        safeMultisigTx.gasToken,
        safeMultisigTx.refundReceiver,
        encodedSignatures,
    ],
})
...
```

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.

MLFD - Multiple Lock Files Detected

Criticality	Minor / Informative
Status	Unresolved

Description

The project includes lock files from more than one package manager, which suggests that different tools (e.g., npm and Yarn) may have been used to install dependencies at different times. This can cause mismatches between the declared dependencies and the actual versions installed, making it difficult to ensure consistent builds, debug issues, or collaborate across environments.

Recommendation

The team is advised to choose a single package manager for the project and remove any lock files associated with other tools. This will ensure consistent dependency resolution, simplify project maintenance, and prevent subtle bugs caused by version conflicts or mismatched dependency trees.

PSDE - Potential Sensitive Data Exposure

Criticality	Minor / Informative
Location	plutus/packages/functions/src/safe/confirmAndExecute.ts#L51
Status	Unresolved

Description

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

```
console.log('Signature:', signature)
console.log('Encoded Signatures:', encodedSignatures)
console.log('execTransactionEncoded:', execTransactionEncoded)
console.log('Transaction executed successfully:', 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.

PFEV - Private Field Encapsulation Vulnerability

Criticality	Minor / Informative
Location	orange/src/config/dotenv.config.ts#L41 orange/src/utils/logger.ts#L2,3,43,68,77 plutus/packages/functions/src/config/dotenv.config.ts#L18
Status	Unresolved

Description

The codebase currently employs the private `modifier` to declare private properties within certain ES6 classes. It's crucial to note that the `private` modifier in TypeScript is a compile-time feature and doesn't enforce privacy at runtime. This could potentially lead to the exposure of sensitive information. As a result, these ES6 classes do not ensure true encapsulation or prevent unintended access.

```
private static validate(): void {}
private static instance: Logger;
private environment: string;
private log(level: string, message: string, metadata?: Record<string, any>): void {}
...
```

Recommendation

The team is strongly advised to replace the usage of the `private` modifier with private fields for sensitive properties within the class. Private fields provide runtime privacy, reducing the risk of unintentional access to sensitive information. To use private fields in JavaScript/TypeScript, the team must preced all private properties names with a '#'. The team is advised to check the Private properties documentation from the MDN Web Docs.

```
class MyClass {
  #privateProp: string;
  ...
}
```

RVD - Redundant Variable Declaration

Criticality	Minor / Informative
Location	plutus/packages/functions/src/aws/signer.ts#L8,216 plutus/packages/functions/src/safe/confirmAndExecute.ts#L14 plutus/sst.config.ts#L5 orange/__tests__/approve.test.tsx#L1 orange/__tests__/deposit.test.tsx#L4 orange/app/api/v1/example/route.ts#L4 orange/src/controllers/test.controller.ts#L6 orange/src/services/blockchain/claim.rewards.service.ts#L1,6
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.

```
Hex
block
protocolKit
_input
afterEach
stakeTokensBex
stakeTokensInfrared
request
...
```

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.

SIE - Server Information Exposure

Criticality	Minor / Informative
Location	orange/src/index.ts#L7
Status	Unresolved

Description

By default, Express sets the `X-Powered-By` HTTP response header, revealing that the application is built using the Express framework. Exposing framework details to potential attackers increases the risk of targeted attacks, as they can exploit known vulnerabilities specific to that framework version. Attackers often use this information to tailor their exploits, making it easier to identify weaknesses in the application.

```
const app = express();

// Middleware
app.use(cors({ origin: true, credentials: true }));
app.use(express.json());
app.use(cookieParser());
```

Recommendation

The `X-Powered-By` header should be disabled to prevent unnecessary information disclosure. This can be done using the built-in Express method `app.disable('x-powered-by')`. Additionally, using security middleware such as Helmet further enhances protection by setting HTTP headers that mitigate various attacks. Helmet is a middleware that sets various security-related HTTP headers, protecting the application against common vulnerabilities such as clickjacking, XSS, and MIME-sniffing attacks. Using Helmet along with disabling X-Powered-By significantly reduces the risk of exposing sensitive information about the server’s underlying technology.

SVDC - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	orange/src/controllers/strategies.controller.ts#L16,26
Status	Unresolved

Description

State variables can be declared as constant using the `const` keyword and initialized at the top of the file or a separate one. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables improve performance and memory consumption.

Furthermore, there are certain static variables declared as `const` inside function scopes. Hence, these variables will be created every time the functions are called.

The following segments is a sample of all the occurences.

```
const poolAddresses = [
    BERA_ADDRESSES.KODIAK_ISLAND_OHM_HONEY,
    BERA_ADDRESSES.KODIAK_ISLAND_WBERA_IBGT,
    BERA_ADDRESSES.KODIAK_ISLAND_WBERA_LBGT,
]

const stakingAddresses = {
    [BERA_ADDRESSES.KODIAK_ISLAND_OHM_HONEY]: {
        infrared: BERA_ADDRESSES.INFRARED_VAULT_OHM_HONEY_IBGT,
        bearn: BERA_ADDRESSES.BEARN_VAULT_OHM_HONEY_YBGT,
        bex: BERA_ADDRESSES.BEX_VAULT_OHM_HONEY_LBGT,
    },
    [BERA_ADDRESSES.KODIAK_ISLAND_WBERA_IBGT]: {
        infrared: BERA_ADDRESSES.INFRARED_VAULT_WBERA_IBGT_IBGT,
        bearn: BERA_ADDRESSES.BEARN_VAULT_WBERA_IBGT_YBGT,
        bex: BERA_ADDRESSES.BEX_VAULT_WBERA_IBGT_LBGT,
    },
    [BERA_ADDRESSES.KODIAK_ISLAND_WBERA_LBGT]: {
        infrared: BERA_ADDRESSES.INFRARED_VAULT_WBERA_LBGT_IBGT,
        bearn: null,
        bex: BERA_ADDRESSES.BEX_VAULT_WBERA_LBGT_LBGT,
    },
}
...
```

Recommendation

Constant state variables can be useful when the codebase wants to ensure that the value of a state variable cannot be changed by any function. This can be useful for storing values that are important to the app's behavior. The team is advised to add the `const` keyword to state variables that never change. Additionally, the team could move static constant variables which are declared inside function scopes to the most upper scope possible, so that they are declared only once.

SSR - Suboptimal Signature Recovery

Criticality	Minor / Informative
Location	plutus/packages/functions/src/aws/signer.ts#L117,143,173 orange/src/services/aws/kms/signer.ts#L89,114,140
Status	Unresolved

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 v of [0, 1]) {
    try {
        const signature = serializeSignature({
            r: ethers.hexlify(r) as `0x${string}`,
            s: ethers.hexlify(s) as `0x${string}`,
            yParity: v,
        })
        const recovered = await recoverAddress({
            hash: digestBytes,
            signature: signature,
        })
        if (recovered.toLowerCase() === address.toLowerCase()) {
            return signature
        }
    } 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/config/viem.client.config.ts#L1 orange/src/controllers/strategies.controller.ts#L3 orange/src/controllers/test.controller.ts#L1 orange/src/routes/index.ts#L2 orange/src/services/safe/signAndProposeTransaction.ts#L1 orange/src/services/kodiak/api.service.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.

```
import { createPublicClient, createWalletClient, http, PublicClient, TestClient,
WalletClient } from "viem"
import { MetaTransactionData } from '@safe-global/types-kit'
import { Request, Response } from 'express'
import { Address, formatUnits } from "viem";
```

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	Unresolved

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.

UEE - Unclassified External Errors

Criticality	Minor / Informative
Location	orange/src/services/aws/kms/utils.ts#L13,63 plutus/packages/functions/src/aws/utils.ts#L13,63
Status	Unresolved

Description

Errors from AWS KMS and external API calls are thrown using generic messages without classifying the nature of the failure (e.g., transient network issues vs. permanent misconfiguration). This lack of error granularity makes it difficult to implement intelligent retry mechanisms, logging, or alerts, and may obscure actionable root causes during incident analysis.

```
if (!response.PublicKey) throw new Error('Failed to get public key from KMS');
if (!signResponse.Signature) throw new Error('Failed to sign message with AWS KMS
key');
```

Recommendation

The team is recommended to classify and handle external errors explicitly by inspecting status codes, error types, or SDK-specific metadata. Distinguish between transient (retryable) and permanent (fatal) errors to enable robust retry/backoff strategies, clearer logging, and more resilient system behavior.

UCC - Unsafe CORS Configuration

Criticality	Minor / Informative
Location	orange/src/index.ts#L10
Status	Unresolved

Description

The CORS setup allows requests from any origin (`origin: true`) while also enabling credentials (`credentials: true`). This combination is insecure, as it permits cross-origin requests with cookies or authentication headers, potentially allowing malicious websites to perform unauthorized actions or exfiltrate sensitive data from authenticated sessions.

```
app.use(cors({ origin: true, credentials: true }));
```

Recommendation

To mitigate this issue, it is strongly advised to restrict the origin setting to a specific trusted domain (or a defined allowlist) when enabling credentials. This ensures that only approved origins can send authenticated requests, reducing the risk of cross-site request forgery and unauthorized data access.

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