Cyberscope

# Audit Report

# Ithaca Protocol

September 2024

| | |
|---|---|
| Repository | https://github.com/ithaca-protocol/ithaca-bot-v2 |
| Commit | 1468bbfdfa7898e5b83f22a264a44156669855b6 |
| 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/ithaca-protocol/ithaca-bot-v2 |
| **Commit** | 1468bbfdfa7898e5b83f22a264a44156669855b6 |
| **Audit Scope** | Development branch |

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 10 Sep 2024 |

# Overview

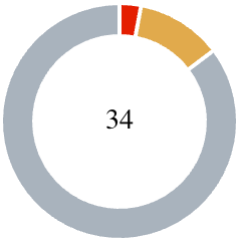
The Ithaca Protocol is a decentralized finance (DeFi) platform designed to facilitate options trading and liquidity provision on the Arbitrum blockchain. By consolidating fragmented markets into a single, optimized auction system, it provides comprehensive options markets, structured products, and trading strategies. This unified approach streamlines access to multiple trading instruments, enhancing liquidity and efficiency within the DeFi space. Ithaca's infrastructure is composed of several interconnected projects and smart contracts, enabling seamless interaction with the platform through a variety of interfaces. One notable interface is Ithaca's Telegram bot, which offers users an intuitive way to interact with the protocol.

The Ithaca Telegram bot serves as a multifunctional tool, enabling users to manage wallets, place trades, deposit and withdraw funds, and access account information. Its wallet management feature supports both self-custody and protocol-custody options, allowing users to create, import, or link wallets. The bot's trading functionality provides users with the ability to execute market orders and employ various option strategies by selecting parameters such as expiry dates, strike prices, and order sizes. Additionally, the bot facilitates cross-chain deposits from networks like Ethereum, Base, and Optimism, alongside handling withdrawals. Through its dashboard, users can monitor their current positions and transaction history, while the bot's pricing and analytics feature provides real-time pricing and visual representations of payoff structures. Backed by the Ithaca SDK, the bot integrates with multiple blockchain networks and external services for pricing calculations, ensuring a secure trading experience through measures like encryption and the use of a Hardware Security Module (HSM) for key operations.

## Hardware Security Module

A Hardware Security Module (HSM) is a dedicated hardware device designed to manage, process, and securely store cryptographic keys and perform encryption and decryption functions. Ithaca's Telegram Bot utilizes an AWS CloudHSM cluster to handle its cryptographic operations, ensuring the secure generation, storage, and use of encryption keys. By leveraging CloudHSM, the bot can securely encrypt sensitive data, such as private keys and transaction information, protecting user assets and communications. AWS CloudHSM offers the advantage of high availability and scalability, as well as compliance with stringent security standards, making it a robust solution for safeguarding cryptographic operations within decentralized finance (DeFi) environments like Ithaca.

# Findings Breakdown




| | |
|---|---|
| Critical | 1 |
| Medium | 4 |
| Minor / Informative | 29 |

| Severity | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|
| Critical | 1 | 0 | 0 | 0 |
| Medium | 4 | 0 | 0 | 0 |
| Minor / Informative | 29 | 0 | 0 | 0 |

# Diagnostics

Critical Medium Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| | IIG | Insecure IV Generation | Unresolved |
| | CSP | Chat State Persistence | Unresolved |
| | SSB | Switch Statement Break | Unresolved |
| | USV | Undeclared State Variables | Unresolved |
| | UNC | Unsafe Numeric Conversion | Unresolved |
| | AKL | API Keys Leakage | Unresolved |
| | AOO | Asynchronous Operations Optimization | Unresolved |
| | CR | Code Repetition | Unresolved |
| | CSU | CommonJS Syntax Usage | Unresolved |
| | CTOU | Complex Ternary Operator Usage | Unresolved |
| | DCE | Dead Code Elimination | Unresolved |
| | EBS | Empty Block Statements | Unresolved |
| | FDD | Function Declaration Duplication | Unresolved |
| | GOVA | Global Object Value Assignment | Unresolved |

| | | | |
|---|---|---|---|
| ● | IUEH | Improper Uncaught Exception Handling | Unresolved |
| ● | IDF | Inconsistent Date Formatting | Unresolved |
| ● | IDR | Inefficient Data Retrieval | Unresolved |
| ● | ISCC | Insecure Session Cookie Configuration | Unresolved |
| ● | IUIV | Insufficient User Input Validation | Unresolved |
| ● | ISIRI | Ithaca SDK Instance Reinitialization Inefficiency | Unresolved |
| ● | MCO | Membership Check Optimization | Unresolved |
| ● | MVN | Misleading Variables Naming | Unresolved |
| ● | MCLS | Missing Centralized Logging Service | Unresolved |
| ● | MEVD | Missing Environment Variables Documentation | Unresolved |
| ● | MEC | Missing ESLint Configuration | Unresolved |
| ● | MPVC | Missing package-lock.json From Version Control | Unresolved |
| ● | MPC | Missing Prettier Configuration | Unresolved |
| ● | MRLM | Missing Rate Limit Middleware | Unresolved |
| ● | PUMU | Potential Unnecessary Mutex Usage | Unresolved |
| ● | SVDC | State Variables could be Declared Constant | Unresolved |
| ● | UCBE | Unexpected Constant Binary Expressions | Unresolved |

| | | | |
|---|---|---|---|
| ● | ULD | Unexpected Lexical Declarations | Unresolved |
| ● | UJSD | Unnecessary JSON Serialization and Deserialization | Unresolved |
| ● | USV | Unused State Variable | Unresolved |

## IIG - Insecure IV Generation

| | |
|---|---|
| **Criticality** | Critical |
| **Location** | utils/hsm.js#L39 |
| **Status** | Unresolved |

### Description

The code generates an Initialization Vector (IV) for encryption by hashing a text input using SHA-256 and then truncating the result to 16 bytes. Using a hash function such as SHA-256 for IV generation is insecure because IVs need to be unique and unpredictable for every encryption operation. Hashing static or predictable data can lead to repeated IVs, weakening the security of the encryption by making it vulnerable to certain types of attacks, such as chosen plaintext or ciphertext attacks.

```
createIV(text) {
    const hash = crypto.createHash('sha256');
    hash.update(text);
    return hash.digest().subarray(0, 16);
}
```

## Recommendation

The team is strongly advised to take these segments into consideration and rewrite them to increase the application's security. The team could follow the recommendations below:

- IVs should be generated randomly for every encryption operation using a secure random number generator. The `crypto.randomBytes(16)` function in Node.js or the `pkcs11.C_GenerateRandom(new Buffer(16))` in PKCS11 is recommended for this purpose.
- When using symmetric encryption algorithms like AES, store the IV alongside the encrypted data (e.g., prepend it to the ciphertext) so that it can be used for decryption. The IV does not need to be secret, but it must be unique for every encryption operation.
- The size of the IV should match the block size of the cipher being used. For example, AES typically uses a 16-byte (128-bit) IV.

Following cryptographic best practices ensures the encryption is robust and secure against potential vulnerabilities.

## CSP - Chat State Persistence

| | |
|---|---|
| **Criticality** | Medium |
| **Status** | Unresolved |

### Description

The contract has been designed to store the chat information in the memory. As a result, in case of an unexpected interruption to the server, the chat state with all the users will be lost. This might produce unexpected results since the users will try to interact with the chat but the application will not be aware of the history.

```
var states = new Map();
// ...
module.exports = {
    clearChatStates,
    clearState,
    getState,

    setStateValue,
    getStateValue,
    clearStateValue,
    stateValueFuncs,

    setLastMessagesState,
    getLastMessagesState,
    appendLastMessagesState,
    clearLastMessagesState,

    setOnboarded,
    isOnboarded,

    setQueryStep,
    getQueryStep
}
```

### Recommendation

The team is advised to evaluate the chat state sync architecture and check for more stable and persist solutions. For instance, databases that are not in the same context as the application server will not lose the state in case of interruption.

## SSB - Switch Statement Break

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | handlers/deposit/arbitrumDepositHandler.js#L47<br>handlers/pricingHandler.js#L38<br>handlers/trading/customStrategyHandler.js#L37,43<br>handlers/trading/marketHandler.js#L32,38,44,50,63,69 |
| **Status** | Unresolved |

## Description

The codebase contains a switch statement with case blocks that lack `break` or `return` statements, potentially leading to fall-through behavior. These case blocks include if-else statements without an else block, which may cause issues if none of the conditions are met. Furthermore, the switch statement does not include a default case as a fallback, which is a best practice to handle unexpected values.

```
case 'Currency':
    setStateValue(chatId, 'currency', dataParts[1]);
case 'SelectAmount':
    handler = showArbitrumSelectAmount;
    break;

case 'Date':
    setStateValue(chatId, 'date', dataParts[1]);
case 'SelectStrike':
    handler = showStrikes;
    break;
...
```

## Recommendation

To prevent fall-through behavior and ensure the switch statement handles all cases appropriately, the team could add `break` or `return` statements to each case block and include a default case. Adding the break statements ensures each case block is exited properly, preventing unintended fall-through. Additionally, including a `default` case handles unexpected transaction types gracefully, improving the robustness of the code.

## USV - Undeclared State Variables

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | handlers/dashboard/liveOrdersHandler.js#L37<br>handlers/dashboard/positionsHandler.js#L38<br>handlers/dashboard/settlementHandler.js##L34<br>handlers/dashboard/tradeHandler.js#L57,157<br>handlers/dashboard/transactionHandler.js#L31<br>handlers/deposit/arbitrumDepositHandler.js#L56,60<br>handlers/deposit/crossDepositHandler.js#L283,284,289 |
| **Status** | Unresolved |

### Description

The codebase contains references to undeclared variables, which could result from misspellings, incorrect variable names, unimported files or accidental creation of implicit global variables. This issue can lead to runtime errors, unexpected behavior, and difficulties in debugging, as the JavaScript engine will either throw a ReferenceError or silently create a global variable if the code is executed in non-strict mode. Such errors can be particularly hard to trace and fix, especially in larger codebases.

```
appendLastMessagesState(chatId, [message])
formatNumberByFixedPlaces(trade.modelPrice, 3)
newMessage
...
```

### Recommendation

The team is strongly advised to use JavaScript's strict mode at the beginning of the scripts or functions. This will prevent the creation of implicit global variables and help catch undeclared variables early in the development process. Additionally, the team should ensure that all variables are declared using let, const, or var before they are used. Lastly, the team should make sure that all required modules are imported correctly where needed. By addressing undeclared variables, the codebase will become more stable, predictable, and easier to work with, leading to a better development experience and a more reliable application.

## UNC - Unsafe Numeric Conversion

| | |
|---|---|
| **Criticality** | Medium |
| **Status** | Unresolved |

### Description

The contract converts the `BigInt` and `String` types directly to the `Number` data type. This convertion might produce invalide results since the `Number` data type support numbers that are less than `253 - 1`. In other cases it might loss the precision. The loss of precision might produce unexpected results to the runtime.

```
const platformFee = Number(orderFee.numeraireAmount);
const collateralRequirementUSDC = parseFloat(orderLock.numeraireAmount || 0) -
(usdcCollateral?.orderValue || 0);
const collateralRequirementWETH = parseFloat(orderLock.underlierAmount || 0) -
(wethCollateral?.orderValue || 0);
const limit = Math.abs(Number(order.totalNetPrice));
const totalPremium = Math.abs(Number(order.totalNetPrice) + platformFee);
```

### Recommendation

The team is advised to check the number size before proceeding to the conversion. If the numbers might be greater than the Javascript Maximum Safe Integer, then it should adopt different data types that support this functionality.

## AKL - API Keys Leakage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/productPlots.js#L127 |
| **Status** | Unresolved |

### Description

Hardcoding API keys in the source code is a security risk and violates best practices. These keys should be stored in environment variables to keep them secure and allow for easy configuration changes without modifying the code.

```
const params = {
    "expiration": "600",
    "key": "7d871677d0400e14580aea35efe617c3",
};
```

### Recommendation

The team is strongly advised to move the API keys to environment variables and access them using `process.env`. By storing API keys in environment variables, the team can enhance the security of the application and make it easier to manage configurations across different environments. Additionally, the team should ensure that the `.env` file is added to `.gitignore` to prevent it from being committed to version control.

## AOO - Asynchronous Operations Optimization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/dashboard/collateral.js#L8,9,10<br>utils/ithaca.js#L234,235,353,355,417,419,420,421,423,452,453,749,751 |
| **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 includes code segments that execute independent asynchronous operations sequentially. This approach introduces a “waterfall effect”, where each operation waits for the previous one to complete before starting. Since these operations are independent and do not rely on each other's results, this leads to unnecessary delays and suboptimal performance.

```
const lockedCollateralData =  await ithacaUtils.getLockedCollateral(chatId)
const fundLockStateData = await ithacaUtils.getFundLockState(chatId)
const fundlockHistoryData =  await ithacaUtils.getFundlockHistory(chatId)

const ithacaSDK = await this.#getIthacaSDK(chatId);
const wallet = await getUserPublicKey(chatId, true);
...
```

## Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. Specifically, the team could refactor the code to execute the asynchronous operations concurrently using `Promise.all()`. This allows the independent functions to run in parallel, reducing the total execution time without altering the logic or behavior of the code.

## CR - Code Repetition

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | handlers/dashboard/liveOrdersHandler.js#L73<br>handlers/dashboard/positionsHandler.js#L74<br>handlers/dashboard/settlementHandler.js#L70<br>handlers/dashboard/transactionHandler.js#L49<br>utils/marketMessages.js#L38,55<br>utils/numbers.js#L2,10 |
| **Status** | Unresolved |

## Description

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

For instance, the pagination business logic in the following functions is identical:

- showLiveOrders
- showPositions
- showSettlements
- showTransactions

```
let orders;
if (!currentPage) {
    orders = await getLiveOrders(chatId)
    setStateValue(chatId, 'orders', orders)
} else {
    orders = getStateValue(chatId, 'orders')
}

const totalPages = Math.ceil(orders.length / ordersPerPage)

const paginatedOrders = orders.slice(currentPage * ordersPerPage, (currentPage + 1) *
ordersPerPage)
```

## 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 codebase could reuse the common code segments in an internal method in order to avoid repeating the same code in multiple places.

## CSU - CommonJS Syntax Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Status** | Unresolved |

### Description

The project currently uses CommonJS syntax for module management, as evidenced by the use of `require()` statements and `module.exports`. While CommonJS has been the standard for Node.js for many years, ES Modules (ESM) have become the preferred module system in modern JavaScript development. ES Modules offer several advantages over CommonJS, including better support for static analysis, improved performance, and native module loading in both browser and server environments.

Sticking with CommonJS might limit the project's ability to take full advantage of modern JavaScript features and tools. As the JavaScript ecosystem continues to evolve, ES Modules are increasingly becoming the standard, with many new tools and libraries designed with ESM in mind.

```
require('dotenv').config();
const mongoose = require('mongoose');
const telegramController = require('./controllers/telegramController');
const ithacaUtils = require('./utils/ithaca');
```

## Recommendation

The team is advised to transition the project from CommonJS to ES Modules by making the following adjustments:

1. Replacing `require()` statements with import and `module.exports` with export. This change will align the project with modern JavaScript practices.
2. Ensuring that the package.json file includes `"type": "module"` to indicate that the project is using ES Modules.
3. Refactoring existing code to adopt ESM syntax, such as replacing `require('dotenv').config()` with `import 'dotenv/config';` .
4. After migration, thoroughly test the application to ensure that all modules are correctly imported and that the application functions as expected in different environments.

By embracing ES Modules, the project will benefit from improved performance, better tooling, and alignment with the latest JavaScript standards, ensuring long-term maintainability and compatibility.

## CTOU - Complex Ternary Operator Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/calcServer.js#L14,28 |
| **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 uses ternary operators for assigning values to variables based on different input conditions. However, the current implementation introduces unnecessary complexity due to deeply nested ternary expressions. This makes the code hard to understand, maintain, and prone to errors as the number of possible values increases.

```
const requestPayoff = payoff === "digital call" ? "binarycall" : payoff === "digital put" ? "binaryput" : payoff;
const productParam = product === "digital call" ? "BinaryCall" : product === "digital put" ? "BinaryPut" : product === "call" ? "Call" : product === "put" ? "Put" : product === "forward" ? "Forward" : product;
```

## Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. The team could refactor the ternary operations into more readable and maintainable structures, such as a switch statement or a mapping object. Both approaches improve readability and make future modifications easier to implement. The mapping object approach is especially useful if the number of conditions grows further.

```
const productMap = {
    "digital-call": "BinaryCall",
    "digital-put": "BinaryPut",
    "call": "Call",
    "put": "Put",
    "forward": "Forward"
};
```

## DCE - Dead Code Elimination

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | handlers/start/termsHandler.js#L47<br>handlers/dashboard/liveOrdersHandler.js#L34<br>handlers/dashboard/positionsHandler.js#L35<br>handlers/deposit/arbitrumDepositHandler.js#L189 |
| **Status** | Unresolved |

### Description

In JavaScript, dead code is code that is written in the codebase, but is never executed or reached during normal execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a codebase more difficult to understand and maintain, and can also increase the size of the app and the cost of deploying and interacting with it.

```
let handler = null;

if (handler) {
    const message = await handler(bot, chatId)
    if (message) {
        appendLastMessagesState(chatId, [message])
    }
}
...
balances.ETH.numberBalance = 1

if (balances.ETH.numberBalance < 0.0001) {
    enoughBalance = false;
    text = 'ETH on Arbitrum One (for gas)';
}
```

## Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

## EBS - Empty Block Statements

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | handlers/dashboard/dashboardHandler.js#L56 |
| **Status** | Unresolved |

### Description

Empty block statements, such as an unimplemented switch statement, often appear in code due to incomplete refactoring or as placeholders for future logic. While these empty blocks do not cause runtime errors, they can create confusion for anyone reading the code, as they suggest that logic was intended to be included but was either forgotten or left unfinished. This can lead to misunderstandings about the purpose of the code and may cause issues during future maintenance or development.

```
switch (dataParts[0]) {
}
```

### Recommendation

To mitigate this issue, the team is advised to take these segments into consideration and rewrite them accordingly:

- If the block is not needed, the team could remove it entirely. This will keep the codebase clean and reduce unnecessary complexity.
- If the block is intended to be implemented later, the team could add a TODO comment explaining what needs to be added. This provides clear guidance for future development.
- If the block was left empty by mistake, the team should complete the necessary logic to handle the required cases.

Ensuring that all block statements are purposeful and complete helps maintain a clean, understandable, and functional codebase.

## FDD - Function Declaration Duplication

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/date.js#L9<br>utils/marketMessages.js#L25 |
| **Status** | Unresolved |

## Description

The codebase currently includes duplicate implementations of certain functions in more than one file. Code duplication can lead to several issues, including increased maintenance complexity, potential inconsistencies when updates are made, and confusion about which version to use.

For instance, the `getNextFriday` function is declared in two separate files, in which both versions of the function serve the same purpose, making it harder to maintain and prone to divergence over time.

```
const getNextFriday = () => {
    let currentFriday = moment().utc().startOf('isoWeek').add(4, 'days').add(8,
'hours');
    if (moment().utc().isAfter(currentFriday)) {
        currentFriday = currentFriday.add(1, 'week');
    }
    return currentFriday;
};
```

## Recommendation

The team is advised to choose a single location for each of these functions and eliminate the duplicate versions. If there were any subtle differences between the two implementations, consider making the function flexible enough to handle all use cases through parameters. By following these steps, the codebase will be more maintainable, and the risk of inconsistencies will be minimized. Centralizing utility functions is a best practice that leads to cleaner, more efficient code.

## GOVA - Global Object Value Assignment

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | app.js#L6 |
| **Status** | Unresolved |

### Description

The project assigns the WebSocket module to the global object using `Object.assign(global, { WebSocket: require("ws") })`. While this approach might offer convenience in development by making certain modules or variables globally accessible, it is strongly discouraged in production environments. Modifying the global object can lead to various issues, including namespace collisions, difficulty in debugging, and unintentional side effects that can make the codebase harder to maintain and reason about.

```
Object.assign(global, { WebSocket: require("ws") });
```

### Recommendation

The team is recommended to mitigate this issue by refactoring the code to avoid assigning values to the global object. Instead, the team could use local imports and pass dependencies explicitly where needed. Where necessary, the team could use dependency injection to pass modules or variables through function arguments, constructors, or higher-order functions. This approach makes the dependencies explicit and easier to manage. Additionally, ensuring that each module explicitly imports the dependencies it requires, the team promotes modularity and improves code maintainability. For example, instead of relying on `global.WebSocket`, the project could directly import WebSocket in each module that needs it. By avoiding modifications to the global object, the project will benefit from cleaner, more maintainable code with fewer risks of unexpected behaviors or conflicts, leading to a more robust and reliable application.

## IUEH - Improper Uncaught Exception Handling

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | app.js#L16,23 |
| **Status** | Unresolved |

### Description

The project currently handles `uncaughtException` and `unhandledRejection` events by logging the errors and restarting the bot if it is not already running. While this method can keep the bot operational in the face of unexpected errors, it is not considered best practice for production environments. These event handlers should primarily be used to log errors and gracefully shut down the process. Relying on them for bot restarts can lead to issues such as memory leaks, corrupted state, or other undefined behaviors if the underlying problem is not addressed. Instead, a process manager like PM2, Docker, or systemd should be utilized to handle process restarts.

```
process.on('uncaughtException', async (err) => {
  console.error('Uncaught Exception:', err);
  if (!ithacaUtils.bot) {
    telegramController.startBot();
  }
});
process.on('unhandledRejection', async (reason, promise) => {
  console.error('Unhandled Rejection:', reason);
  if (!ithacaUtils.bot) {
    telegramController.startBot();
  }
});
```

## Recommendation

The team is strongly advised to employ a process manager like PM2, Docker, or systemd to handle the restarting of the bot. These tools are designed to manage application processes, automatically restart them on failure, and provide additional benefits such as log management and load balancing. Additionally, the team could modify the uncaughtException and unhandledRejection handlers to log the error and gracefully shut down the process by ensuring that the bot and any associated resources (e.g., database connections, file handles) are properly shut down before exiting. By following these recommendations, the project will benefit from a more stable, maintainable, and resilient application, reducing the risk of unexpected failures and improving the overall reliability of the bot.

## IDF - Inconsistent Date Formatting

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/date.js#L1,7 |
| **Status** | Unresolved |

### Description

The codebase currently utilizes more than one date formatting library, specifically dayjs and moment. This inconsistency increases code complexity, potential bugs, and maintenance overhead. Having multiple libraries for similar functionalities can lead to confusion, version conflicts, and inconsistent date manipulations across the codebase.

```
const dayjs = require("dayjs")
const customParseFormat = require("dayjs/plugin/customParseFormat")
const UTC = require("dayjs/plugin/utc")
dayjs.extend(customParseFormat);
dayjs.extend(UTC);

const moment = require('moment');
```

### Recommendation

The team is advised to standardize the codebase to use only one date formatting library. This approach simplifies the codebase, reduces dependencies, and establishes a single source of truth for date formatting and manipulation.

## IDR - Inefficient Data Retrieval

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/db.js#L37,46 |
| **Status** | Unresolved |

### Description

The codebase contains queries that retrieve entire MongoDB documents even when only a specific property (e.g., public_key) is required. This practice leads to unnecessary data retrieval, increased memory usage, and inefficient database queries, especially when working with large datasets or documents containing many fields.

For example, in the `getUserPublicKey` function, the entire user document is fetched, even though only the `public_key` field is needed.

```
const getUserPublicKey = async (user_id, throwError) => {
  const user = await getUser(user_id);
  const publicKey = user?.public_key;
  if (!publicKey && throwError) {
    throw new WalletNotFoundError();
  }
  return publicKey;
}
```

### Recommendation

The team is recommended to optimize queries to fetch only the required fields by using MongoDB's projection feature. This reduces the amount of data retrieved from the database, improving performance and reducing resource consumption. By applying this approach, the database query becomes more efficient by fetching only the required fields, reducing unnecessary overhead and improving performance.

## ISCC - Insecure Session Cookie Configuration

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | middlewares/session.js#L7 |
| **Status** | Unresolved |

### Description

The session middleware in the codebase is configured with the cookie's secure attribute set to false, which means that the session cookies can be transmitted over an insecure (HTTP) connection. When the secure attribute is false, cookies are sent in plaintext, making the application susceptible to man-in-the-middle (MITM) attacks where attackers can intercept and potentially hijack user sessions.

```
const session = require('express-session');

module.exports = session({
  secret: 'xxxxx',
  resave: false,
  saveUninitialized: true,
  cookie: { secure: false }
});
```

### Recommendation

The team is strongly advised to set the `cookie.secure` attribute to true, ensuring that session cookies are only sent over HTTPS connections. This change enhances security by ensuring that cookies are only transmitted over encrypted channels. Additionally, the team could configure `secure: process.env.NODE_ENV === 'production'` to dynamically apply secure settings based on the environment. Adopting secure cookie configurations will significantly enhance the application's resilience against session-related attacks.

# IUIV - Insufficient User Input Validation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Status** | Unresolved |

## Description

In several files, user input is not adequately validated before being processed and used in the application. This can lead to a variety of security vulnerabilities, such as input tampering, injection attacks, and potential crashes due to invalid data types. For instance, in the code segment the app directly accepts user input from chat messages without thorough validation, making the application vulnerable to malicious input.

The following example is a sample of all the occurences.

```
const enteredText = msg.text;
...
if (step.includes(QueryPrefix + ':Amount:Custom')) {
    appendLastMessagesState(chatId, [msg]);
    const amountInWei = ethers.parseUnits(enteredText,
Networks[network].tokens[token].decimals);
    await handleCallbackQuery(bot, {message: {chat: {id: chatId}}, data: QueryPrefix
+ ':Amount:' + amountInWei});
}
...
```

## Recommendation

To mitigate this issue, it is recommended to always validate user input. The team could use the following recommendations:

- Implement robust validation mechanisms for all user inputs.
- Where applicable, use specialized libraries to validate sensitive data such as Ethereum private keys and cryptocurrency amounts.
- Add appropriate error handling for invalid inputs.
- Instead of basic checks (e.g., if a private key starts with 0x), verify the entire key for both length and content integrity.

By following the above recommendations, the app could prevent various attack vectors, such as injection and tampering, ensure that only valid inputs are processed, reducing crashes and unhandled errors, and provide users with immediate feedback for invalid inputs, improving the overall user experience and minimizing errors.

## ISIRI - Ithaca SDK Instance Reinitialization Inefficiency

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/ithaca.js#L90 |
| **Status** | Unresolved |

### Description

The `#getIthacaSDK` method is responsible for initializing the Ithaca SDK instance based on the provided `chatId`. Currently, when the chatId changes, the method logs out of the existing SDK instance and reinitializes it, including re-authentication. While this ensures the SDK is correctly associated with the active chat, it introduces inefficiencies by reinitializing and logging out the SDK even when the underlying state does not require modification. This reinitialization can be resource-intensive, especially if the SDK is frequently switched between chat IDs, impacting performance.

```
async #getIthacaSDK(chatId) {
    if (this.ithacaSDKInstance) {
        if (this.activeChatId === chatId) {
            return this.ithacaSDKInstance;
        }
        await this.ithacaSDKInstance.auth.logout();
        this.ithacaSDKInstance = null;
    }

    this.activeChatId = chatId;

    const userInfo = await getUserInfo(chatId, true);

    let walletClient = undefined;
    if (userInfo.private_key) {
        const account = privateKeyToAccount(userInfo.private_key);
        walletClient = createWalletClient({
            account,
            chain: process.env.BC_NET === 'arbitrum' ? arbitrum : arbitrumSepolia,
            transport: http(),
        });
    }

    this.ithacaSDKInstance = new IthacaSDK({
        ...
    });

    let result;
    if (userInfo.private_key) {
        result = await this.ithacaSDKInstance.auth.login();
    } else {
        result = await this.ithacaSDKInstance.auth.rsaLogin(userInfo.public_key);
    }
    console.log('chatId', chatId, 'ithaca auth login result =>', result);
    return this.ithacaSDKInstance;
}
```

## Recommendation

To improve efficiency, it is recommended to refactor the code to avoid unnecessary SDK reinitialization when switching between chat IDs. The team could implement a caching mechanism that stores instances of the SDK for different chat IDs, allowing reuse without redundant logouts and reinitializations. This approach reduces the overhead of re-authentication and SDK recreation. It could be improved even further with an automatic cleanup mechanism to remove instances for chat ids after a specified period. This refactor eliminates unnecessary logouts and SDK reinitializations, improving both performance and resource efficiency. The cached instances allow seamless switching between chat IDs without repetitive SDK re-creation, reducing the overhead on the system.

## MCO - Membership Check Optimization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | controllers/telegramController.js#L67 |
| **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 current implementation of user role verification in the Telegram bot checks if a user is authorized to interact with the bot by making an API request to retrieve the user's role in the group. The code then verifies if the user's status is one of the allowed roles ( `member` , `administrator` , `creator` ) using an `if` statement with an array of allowed statuses. While this approach works, it can be simplified to improve code readability and maintainability. Additionally, moving the static array outside the function scope could also improve performance.

```
const checkMembership = async (bot, chatId) => {
    try {
        ...
        if (['member', 'administrator', 'creator'].includes(status)) {
            return true;
        }
    } catch (error) {
        console.log('Error in checking group membership', 'chatId:', chatId,
'error:', error)
    }
    ...
    return false
}
```

## Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. A recommended approach would be to simplify the role verification logic by directly returning the result of the value of the status check.

```
const allowedRoles = new Set(['member', 'administrator', 'creator']);

const checkMembership = async (bot, chatId) => {
    try {
        ...
        const isAuthorized = allowedRoles.has(status);
        if (!isAuthorized) {
            // delete previous messages, etc...
        }

        return isAuthorized;
    } catch (error) {
        console.log('Error in checking group membership', 'chatId:', chatId,
'error:', error);
        return false;
    }
}
```

## MVN - Misleading Variables Naming

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/date.js#L10,18 |
| **Status** | Unresolved |

## Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The codebase uses certain variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

For instance, the `getNextFriday` function returns the date of the next friday. However, the variable holding that value is named `currentFriday` .

```
const getNextFriday = () => {
    let currentFriday = moment().utc().startOf('isoWeek').add(4, 'days').add(8,
'hours');
    if (moment().utc().isAfter(currentFriday)) {
        currentFriday = currentFriday.add(1, 'week');
    }
    return currentFriday;
};
let currentFriday = getNextFriday();
```

## Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

## MCLS - Missing Centralized Logging Service

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | app.js#L11,13,24<br>controllers/telegramController.js#L22,46,57,75<br>handlers/priceHandler.js#L286,294 |
| **Status** | Unresolved |

### Description

The project currently relies on console statements for debugging and logging information and erros 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("MongoDB connected");
console.error.bind(console, "MongoDB connection error:")
console.error('Unhandled Rejection:', reason);
console.log('Handling callback query', step, 'chatId:', chatId);
console.log('Handling command', command.pattern, 'chatId:', chatId, 'message:',
msg.text);
```

### 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.

# MEVD - Missing Environment Variables Documentation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/balance.js#L8<br>utils/ithaca.js#L770<br>utils/hsm.js#L5,6,7<br>constants/networks.js#L5<br>handlers/start/termsHandler.js#L72 |
| **Status** | Unresolved |

## Description

The project relies on several environment variables for its functionality. However, some of these variables are not listed in the `.env.example` file or documented in the `README.md` file. This omission can lead to confusion among developers, contributors, and users, making it difficult to set up the project correctly or understand what configuration is required. Missing documentation on these variables may also increase the risk of misconfigurations or deployment issues.

```
const provider = new ethers.InfuraProvider(chainId, process.env.ETH_INFRA_ID);
const HSM_USER = process.env.HSM_USER;
const HSM_PASSWORD = process.env.HSM_PASSWORD;
const HSM_KEY_LABEL = process.env.HSM_KEY_LABEL ?? 'canbotv2aes';
...
```

## Recommendation

To mitigate this issue, it is recommended to include all required environment variables in the `.env.example` file. Provide default values where applicable, and leave placeholders for sensitive information. This serves as a template for setting up the environment correctly. Additionally, adding a section in the README.md file that explains the purpose of each environment variable, how to obtain necessary values (e.g., API keys), and any default values or recommended settings helps users understand the importance of each variable and how to configure them properly. By ensuring all necessary environment variables are documented and included in the .env.example file, the project will be more accessible, reliable, and easier to maintain.

# 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.

## MPVC - Missing package-lock.json From Version Control

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | .gitignore#L11 |
| **Status** | Unresolved |

## Description

The project's `.gitignore` file currently excludes the `package-lock.json` file from version control. This can introduce potential vulnerabilities and inconsistencies in the project. When package-lock.json is not committed to the repository, each contributor might install different versions of dependencies, leading to discrepancies in the development and production environment and potentially introducing bugs or vulnerabilities that are hard to trace.

Without a consistent package-lock.json file, contributors could end up with different versions of the same packages, causing inconsistent development environments, hard to reproduce bugs and increased vulnerability risk.

```
 # Ignore package-lock.json if you want (optional)

package-lock.json
```

## Recommendation

The team is advised to ensure the package-lock.json file is included in the repository and shared among all contributors. This guarantees that everyone is using the same versions of the dependencies, creating a consistent and reproducible development environment. Additionally, the team could periodically review and update dependencies to their latest stable versions, and then commit the updated package-lock.json file to the repository. By following these recommendations, the project will have a more stable and secure dependency management process, reducing the risk of inconsistencies and vulnerabilities.

## MPC - Missing Prettier Configuration

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Status** | Unresolved |

### Description

The app lacks a Prettier configuration file, which results in inconsistent code formatting. The codebase exhibits inconsistencies in spacing and comments formatting in several places. Without a Prettier configuration, contributors may use different formatting styles, leading to a codebase that is harder to read and maintain.

### Recommendation

The team is advised to add a Prettier configuration file ( `.prettierrc` or `prettier.config.js` ) to the root of the project. This ensures that all contributors follow the same formatting guidelines. Additionally, the team could consider adding a prettier script to the `package.json` to facilitate formatting. By enforcing a consistent coding style, the codebase will become more readable and maintainable, improving overall code quality.

## MRLM - Missing Rate Limit Middleware

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | controllers/telegramController.js#L19 |
| **Status** | Unresolved |

### Description

The project does not implement any form of rate limiting, especially for the bot that processes user interactions via polling. This absence of rate limiting exposes the application to Denial of Service (DoS) attacks and other abusive behaviors, where a malicious user can flood the bot with excessive requests, potentially overwhelming the server, consuming resources, and making the service unavailable to legitimate users.

```
bot = new TelegramBot(telegramToken, { polling: true });
ithacaUtils.setBot(bot);
```

### Recommendation

The team is recommended to introduce rate limiting to control the number of requests allowed from a single user or IP address in a given period. Additionally, implementing security measures such as blocking or blacklisting IPs that make an excessive number of requests, especially after hitting rate limits, would be beneficial.

## PUMU - Potential Unnecessary Mutex Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/ithaca.js#L195,209,220,231,243,254,265,276,287 |
| **Status** | Unresolved |

### Description

The codebase applies mutex locks to multiple asynchronous getter functions interacting with the Ithaca SDK (e.g., `getLockedCollateral`, `getFundLockState`). Mutexes are typically used to prevent race conditions in cases where shared resources are being modified by concurrent operations. However, the getter functions in question are seamingly data-retrieval methods, which are generally not expected to modify shared state. The use of mutexes in these contexts introduces performance overhead due to unnecessary synchronization, especially if these methods are frequently called.

```
async getLockedCollateral(chatId) {
    const release = await mutex.acquire();
    try {
        const ithacaSDK = await this.#getIthacaSDK(chatId);
        const result = await ithacaSDK.client.getLockedCollateral();
        return result;
    } finally {
        release();
    }
}
async getFundLockState(chatId) {
    const release = await mutex.acquire();
    try {
        const ithacaSDK = await this.#getIthacaSDK(chatId);
        const result = await ithacaSDK.client.fundLockState();
        return result;
    } finally {
        release();
    }
}
...
```

## Recommendation

The team is advised to confirm whether these SDK methods being used modify shared state or require synchronization. If they do not modify shared resources, the mutex can be safely removed. If certain parts of the SDK interaction require synchronization but not all getter functions, apply mutexes only to those methods. Consider using a read-write lock for scenarios where multiple reads can happen concurrently, but writes need to be exclusive. By ensuring mutex usage aligns with actual concurrency requirements, performance can be improved while maintaining correct behavior.

## SVDC - State Variables could be Declared Constant

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | handlers/dashboard/dashboardHandler.js#L69<br>handlers/deposit/arbitrumDepositHandler.js#L93,115,141,168,234,330<br>handlers/deposit/crossDepositHandler.js#L127,161,166,188,304,342,371,406,472,523,554<br>handlers/deposit/depositHandler.js#L41,50<br>handlers/start/onboardHandler.js#L91,109,121,145,166,182,196,207<br>handlers/trading/customStrategyHandler.js#L106,135,149,163,177,191,209,241,256,266,287 |
| **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.

```
[
    [
        {
            text: 'Collateral',
            callback_data: CollateralQueryPrefix,
        },
        {
            text: 'Live Orders',
            callback_data: LiveOrdersQueryPrefix,
        },
    ],
    [
        {
            text: 'Positions',
            callback_data: PositionsQueryPrefix
        },
        {
            text: 'Trade History',
            callback_data: TradeHistoryQueryPrefix
        },
    ],
    [
        {
            text: 'Settlements',
            callback_data: SettlementsQueryPrefix
        },
        {
            text: 'Transaction History',
            callback_data: TransactionsQueryPrefix
        }
    ],
    [{text: 'Back', callback_data: 'Menu'}]
]
...
```

## 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.

## UCBE - Unexpected Constant Binary Expressions

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/pricing.js#L10 |
| **Status** | Unresolved |

### Description

The codebase contains binary expressions where comparisons are made using constants, such as `null === undefined`, which will always evaluate to either true or false. In this example, null === undefined will always evaluate to false, making the expression unnecessary and likely indicating a logic error. Additionally, this type of issue can occur in logical expressions (||, &&, ??`), where certain parts of the expression might always short-circuit or never short-circuit, leading to unintended behavior or bugs.

```
if (num === null || null === undefined || isNaN(num)) return '      -      ';
```

### Recommendation

The team is advised to remove the constant binary expressions and replace them with meaningful checks, ensuring that all parts of the conditional expression are necessary and relevant to the logic. Simplifying logical expressions and removing constant binary comparisons will make the codebase cleaner, more efficient, and less prone to bugs.

# ULD - Unexpected Lexical Declarations

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | handlers/priceHandler.js#L52,57,59<br>handlers/start/termsHandler.js#L29,33<br>handlers/trading/linearCombinationHandler.js#L34<br>handlers/withdrawHandler.js#L42 |
| **Status** | Unresolved |

## Description

The codebase uses lexical declarations such as let, const, and function inside `case` or `default` clauses within a switch statement. Lexical declarations are hoisted to the top of the switch block, meaning they are in scope for the entire switch but remain uninitialized until the case where they are declared is executed. If a different case is executed first, accessing such variables will throw a ReferenceError before the variable is initialized.

This can lead to unexpected behavior and difficult-to-debug issues since the variable appears to be in scope but isn't initialized unless its associated case is executed.

```
case 'Payoff':
    const new_payoff = dataParts[1].toLowerCase();
    setStateValue(chatId, 'payoff', new_payoff);
    handler = showSelectExpiryDates;
    break;
case 'Product':
    const productKey = dataParts[1].toLowerCase().replaceAll(' ', '-');
    setStateValue(chatId, 'product', productKey);
    const new_strategy = strategies.findStrategy(productKey);
...
```

## Recommendation

To mitigate this issue, the team is advised to wrap each case clause in a block {} to ensure that lexical declarations are scoped to their respective cases, ensuring that variables are only accessible and initialized within the correct case block. By properly scoping lexical declarations within switch statements, the code will become more stable and less prone to errors related to variable hoisting and initialization.

## UJSD - Unnecessary JSON Serialization and Deserialization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | utils/strategies.js#L818 |
| **Status** | Unresolved |

## Description

The `findStrategy` function performs an unnecessary JSON serialization and deserialization operation on the found strategy object. This process is redundant because the strategy object is already in the correct format when retrieved from the `STRUCTURED_STRATEGIES` or `LINEAR_STRATEGIES` arrays. The current implementation uses `JSON.parse(JSON.stringify(strategy))`, which creates a deep copy of the object but is an inefficient, redundant and potentially problematic approach.

```
const findStrategy = (key) => {
    const strategy = STRUCTURED_STRATEGIES.concat(LINEAR_STRATEGIES).find(strategy =>
strategy.key === key);
    return strategy ? JSON.parse(JSON.stringify(strategy)) : strategy;
};
```

## Recommendation

The team is advised to simplify the `findStrategy` function by removing the unnecessary JSON operations. Instead, the team could return the found strategy object directly. If a deep copy is necessary, the team could consider using a dedicated deep cloning function or library, like the Node.js built-in function `structuredClone`, which would be more efficient and safer than JSON serialization.

## USV - Unused State Variable

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | app.js#L23<br>handlers/dashboard/dashboardHandler.js#L2,12<br>handlers/dashboard/liveOrdersHandler.js#L2<br>handlers/dashboard/positionsHandler.js#L1<br>handlers/dashboard/settlementHandler.js#L1<br>handlers/dashboard/tradeHandler.js#L1,6<br>handlers/dashboard/transactionHandler.js#L1,36 |
| **Status** | Unresolved |

## Description

An unused state variable is a state variable that is declared in the codebase, but is never used in any of the codebase's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the codebase and make it more difficult to understand and maintain. They can also increase the size of the codebase and impact the performance of the app.

The following are a sample of all the occurences.

```
promise
alignButtons
setStateValue
getStateValue
defaultMessageOption
deleteMessage
...
```

## Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the app's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the codebase.

# Summary

The Ithaca Protocol implements both a backend infrastructure and a frontend interface, through its Telegram bot. 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 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.

**The Cyberscope team**

cyberscope.io