Blaize.Security

November 16th, 2023 / V. 2.0

PROMETHIUM

PROMETHIUM SECURITY AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **9.75**/10

The scope of the project includes Promethium Rebalancer protocol and backend service:

Smart contracts:
DataTypes.sol, RBAC.sol
Rebalancer.sol, Registry.sol, PriceRouter.sol

Repository: https://github.com/Spectrum-Dev-Team/promethium-smart-contract/, main branch

Initial commit: ■ e8ebd386707f6e8abb4bf87541fb046535aafd8e
Final commit: ■ e603a7e2e0d66fa43015a96a8a2a507332889a13

Backend service:
jest.config.ts, libs/domain/src/ *.ts
apps/rebalance-executor/jest.config.ts
apps/rebalance-executor/src/ *.ts
apps/rebalance-approver/jest.config.ts
apps/rebalance-approver/src/ *.ts
apps/api/jest.config.ts, apps/api/src/ *.ts
apps/api/src/app/ *.ts
apps/rebalance-proposer/jest.config.ts
apps/rebalance-proposer/ *.ts

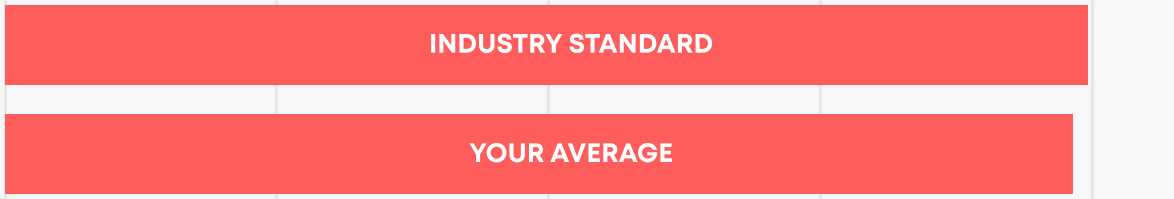
Repository: https://github.com/Promethium-Dev-Team/promethium-backend-ts, main branch

Initial commit: ■ 938e39b4111ed51822d2019dbfa4b7558fa11394
Final commit: ■ 57b94fcabcff3ca4453842cafe6a12fbab23fa59

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts and backend service for the Promethium protocol. Our task was to find and describe any security issues in the smart contracts of the platform and rebalancing backend service. This report presents the findings of the security audit of the **Promethium** codebase conducted during 2 audit iterations (and including several additional reviews of deployed code and code upgrades) between **September 19th, 2023** and **November 16th, 2023.**

Testable code



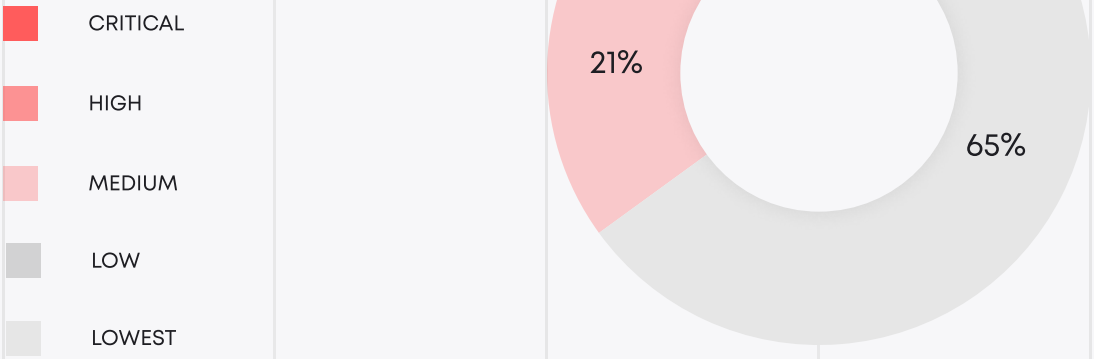
Auditors approved code as testable within the industry standard.

The audit scope includes all tests and scripts, documentation, and requirements presented by the **Promethium** team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies, and includes testable code from manual and exploratory rounds.

However, to ensure the security of the contract, the **Blaize.Security** team suggests that the **Promethium** team follow post-audit steps:

1. launch **active protection** over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
2. launch a **bug bounty program** to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 21 problems were found. 21 issues were fixed or verified by the Promethium team.

	FOUND	FIXED/VERIFIED
Critical	0	0
High	0	0
Medium	5	5
Low	3	3
Lowest	13	13

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

Blaize.Security auditors start the audit by developing an **auditing strategy** - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

Manual audit stage

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

For advanced components:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

Testing stage:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

EXECUTIVE SUMMARY

The Blaize Security team received a set of smart contracts from the Promethium team. These contracts represent a module of rebalancing positions in various external DeFi protocols with the main contract of Vault type, inheriting the ERC-4626 standard.

The audit's objective was to ensure that these contracts functioned correctly and maintained a known security level. Our auditors meticulously examined each line of code, cross-referencing it with a vulnerabilities checklist. They validated the business logic of the contracts and ensured that they followed best practices - especially regarding gas spending. Any issues discovered were promptly communicated to the Promethium team and eventually resolved or verified. The setup and deployment scripts of the contracts were also subjected to rigorous auditing.

During the manual portion of the audit, we identified two medium-severity issues, three low-severity issues, and nine informational-severity issues. The medium-severity issue pertained to a potential denial-of-service attack due to missing validation for the passed amount for withdrawing shares, as it was possible to request a withdrawal even for zero shares. Promethium fixed this issue by validating the passed share amount for greater than zero. Another medium-severity issue was connected to the fee paid to a previous treasury in case of changing treasury. The Promethium team has resolved that issue.

Low-severity findings were related to issues in adjusting contract settings, floating pragma, and lack of validation. The Promethium team resolved all of these issues. Most informational issues were due to unoptimized code. All findings were discussed with the Promethium team over several communication iterations, and almost all of them were resolved, and two informational issues were verified.

Another part of our audit process involved reviewing the native tests the Promethium team prepared and creating a set of custom test scenarios. The development team provided comprehensive tests covering all core contract logic. However, Blaize Security supplemented this with its own unit tests and additional scenarios to cover complex functionality in the core and pool manager contracts. The complete set of unit tests can be found in the Code Coverage and Test Result sections.

It should be mentioned that there were three additional findings in the preliminary report. They were due to possible front-running attacks, flashloan attacks, and rounding issues. All these issues were discussed with the Promethium team and were explicitly tested by Blaize Security engineers. None of the edge-case tests pointed out the attack scenario, which could be profitable for the attacker. Thus, all three issues were removed from the final report. The threat of the possibly malicious adaptor calls in the rebalance was addressed by the auditors. Promethium team implemented mitigation of that issue by validating selector of the adaptor call.

Also, Promethium team implemented upgradeability of the contracts, thus including significant element of centralization. Nevertheless, during the last review, auditors checked the migration from the standard contracts to upgradeable contracts. Though, still it is a controllable backdoor, thus contracts fail the appropriate check.

2ND ITERATION

During the second audit iteration, our team has been working on auditing Promethium backend components. Promethium backend handles rebalance activities in a protocol, interacting with smart contracts, managing the rebalance strategies, and withdrawals queue.

Auditors conducted a general analysis of the backend services of the protocol. It is a well-structured system with the following components:

- Rebalance Workers: proposer, approver, and executor worker handle respectively proposing, approving, and executing rebalance transactions – making use of the Safe protocol and Safe API Kit.
- The central package manages the shared logic of the application: ABIs, pool wrappers, configs, subgraph, rebalance strategy, etc.
- API, which provides routes for pools, balances, and wallet data + contains a worker that updates token prices and pool APRs.

Audit of the backend focused on:

- Gas usage and monitoring of rebalance workers.
- Storage and management of private keys.
- APR calculation approaches.
- Rebalance strategy and its execution.
- Deployment strategy and assumptions.
- Withdrawal queue management.

Auditors meticulously analyzed core components, ensuring potential risks were identified. Our findings cover both security-related aspects and core business logic decisions. The main issues highlighted include gas monitoring for workers, APR calculation mismatches, migrating from storage of private keys in environment variables to the KMS services, the 'greedy' rebalance strategy, general questions of deployment security, and the limited withdrawal queue.

In addition, auditors conducted a separate verification of the APR calculation logic used in the project since APR calculation drives decision-making in the current Promethium rebalance strategy. The Promethium protocol supports adapters for AaveV3, CompoundV3, DForce, Granary, Lodestar, RadiantV2, Tender, WePiggy. While the protocol uses standard approaches for Aave, Compound and DForse, Granary and Radiant (based on their documentation), other protocols got a unified approach based on the supply rate per block.

The rebalancing strategy is a pivotal logic within Promethium's backend, driving its core operations. At its essence, the strategy revolves around determining the most profitable pools based on APR. All available pools are assessed, with their APRs systematically retrieved and sorted. The strategy is rather direct: assets are consolidated towards the most profitable pool. While this approach prioritizes immediate gains, it's essential to note its inherent nature of leaning towards a single, top-performing pool, which may limit diversification. Though, the team added 2 Medium-risk issues related to the calculations into the report and provides additional checks regarding the correctness of calculations. The team provided an upgrade in additional PR to address these issues. Though still, APR **does not include** native token rewards - but the team confirmed to have this feature in a future release.

Throughout our examination of the project, it was clear that the code was well structured. The codebase is organized, making it easier to navigate and understand. It follows best practices, demonstrating a balance between functionality and clarity. The team has shown a solid strategy for the mitigation of risks of keys' storage vulnerabilities and future logic optimization.

	RATING
Security	9.5
Logic optimization	9.4
Code quality	10
Test coverage*	9.8
Total	9.75

Note: Final mark is based on both audit iterations, including the latest code updates with contracts upgradeability.
* Test coverage shows the overall impression on the testable code based on existing set of unit tests, all checks performed by auditors, and the general overview after the testing stage.

PROTOCOL OVERVIEW

Rebalancer.sol:

The **Rebalancer** contract is a sophisticated financial instrument that allows users to deposit assets into a vault, interacting with various positions to earn returns. It provides functionalities for rebalancing assets, requesting withdrawals, and managing fees. The contract also ensures compliance with the ERC-4626 standard. Key functions include rebalance, which adjusts the asset distribution based on a given matrix, and _fullfitWithdrawals, which processes queued withdrawal requests.

rebalance function adjusts the asset distribution of the vault based on a provided distribution matrix. Rebalancing positions mechanism undermines the process of executing transactions on external protocols. Transactions before the rebalancing are packed in the encoded form. As these transactions might be malicious, there is validation for function call which was encoded. It also ensures that the asset balance after rebalancing doesn't fall below a certain threshold and then fulfills all pending withdrawal requests.

_fullfitWithdrawals processes all queued withdrawal requests. It calculates the amount of assets corresponding to the shares requested for withdrawal, deducts any applicable fees, and transfers the assets to the user.

Registry.sol:

The **Registry** contract acts as a storage hub for the vault, maintaining a list of positions (protocols) and iTokens (interest-bearing tokens) the vault can interact with. It also provides functionality for adding or removing positions and iTokens, ensuring the vault's adaptability to changing financial landscapes. The Registry contract has a role-based access control mechanism, ensuring only authorized entities can modify its state.

addPosition and **addIToken** allow the contract owner to add new protocols (positions) and interest-bearing tokens (iTokens) that the vault can interact with. The owner also can remove iTokens using the removeIToken function. They ensure that the added entities are not already present and that the contract doesn't exceed predefined limits for positions and iTokens.

PriceRouter.sol:
The PriceRouter contract functions as a pricing oracle, determining the value of a given amount of an iToken in terms of its underlying asset. It supports multiple tokens and iTokens, and its primary function, getTokenValue, calculates the equivalent value of an amount of iToken in its base token.

getTokenValue takes a token, its corresponding iToken, and an amount as input parameters. It returns the equivalent value of the given iToken amount in terms of the base token. The function supports multiple tokens and iTokens, and it uses predefined logic for each pair to determine the value.

LIST OF VALUABLE ASSETS

Rebalancer.sol

- Underlying asset: Represented by the asset() function.
- FeeData: Contains data about the performance fee, withdrawal fee, and the treasury address. Changes in these can affect the returns for users.
- poolLimitSize: The maximum size of the pool. This limits the total assets the contract can handle.
- lockedShares: Represents the shares of users that are locked and awaiting withdrawal.
- withdrawQueue: A queue of withdrawal requests from users. This is crucial as it represents pending user withdrawals.

Registry.sol

- iTokens: A list of interest-bearing tokens the vault can hold. These tokens represent the underlying asset in various protocols.
- positions: A list of protocols or positions the vault can interact with. These are the platforms where the vault places assets to earn returns.
- depositsPaused: A boolean indicating if deposits are currently paused. This is crucial as it can halt all deposits into the contract.

PriceRouter.sol

- Base tokens: usdt, usdc_e, wbtc, weth, and arb. These are the tokens for which the contract provides pricing information when converted to iTokens.
- Token addresses (usdt, usdc_e, etc.): Addresses are used to identify the base tokens when determining the value of iTokens. Changes or errors in these addresses can lead to incorrect pricing information.

ROLES AND RESPONSIBILITIES

Rebalancer.sol

1. Deployer:
The deployer is responsible for the deployment of Rebalancer.sol. The deployer must pass valid initial parameters such as the asset address, the name of the share symbol of the vault, the list of protocols (positions) it can interact with, the list of interest tokens it can interact with and hold, the list of addresses that get the rebalancer role, the address of the price oracle, the list of users to be whitelisted, and the maximum size of the of the vault (pool). The deployer becomes the owner during deployment.

2. Owner:
The owner is responsible for setting up all the parameters and variables of the vault, adding new interest tokens. The owner receives the fee collected by the vault.

Only owner can call the following functions:
- addIToken()
- claimFee()
- setFee()
- setPoolLimit()

3. User:
User is the main actor of the protocol. He delivers assets to the vault for receiving further rewards. In exchange for the assets delivered, the user receives shares in the vault. The user can return the shares and get the assets delivered back.

Any user can call the following functions:
- methods from òpenzeppelin`standards: ERC4626, ERC20, AccessControl
- getAvailableFee()
- maxRedeem()

- requestWithdraw()
- totalAssets()

4. **Rebalancer:**

The rebalancer plays a key role. It is responsible for rebalancing assets based on a given matrix.

The following functions can be called by only rebalancer:

- rebalance()

5. **Oracle:**

Oracle provides the value of a certain number of iTokens relative to the underlying asset.

Registry.sol

1. **Owner:**

The owner is responsible for configuring all registry parameters and variables, adding new interest tokens and deleting existing ones. Adding new protocols (positions) and deleting existing ones. Pause or restore the work of the registry.

Only owner can call the following functions:

- addIToken()
- addPosition()
- removeIToken()
- removePosition()
- setPause()

RBAC.sol

1. **Owner:**

The owner is responsible for pausing and restoring work of the whitelist.

Only owner can call the following functions:

- disableWhitelist()

DEPLOYMENT SCRIPT

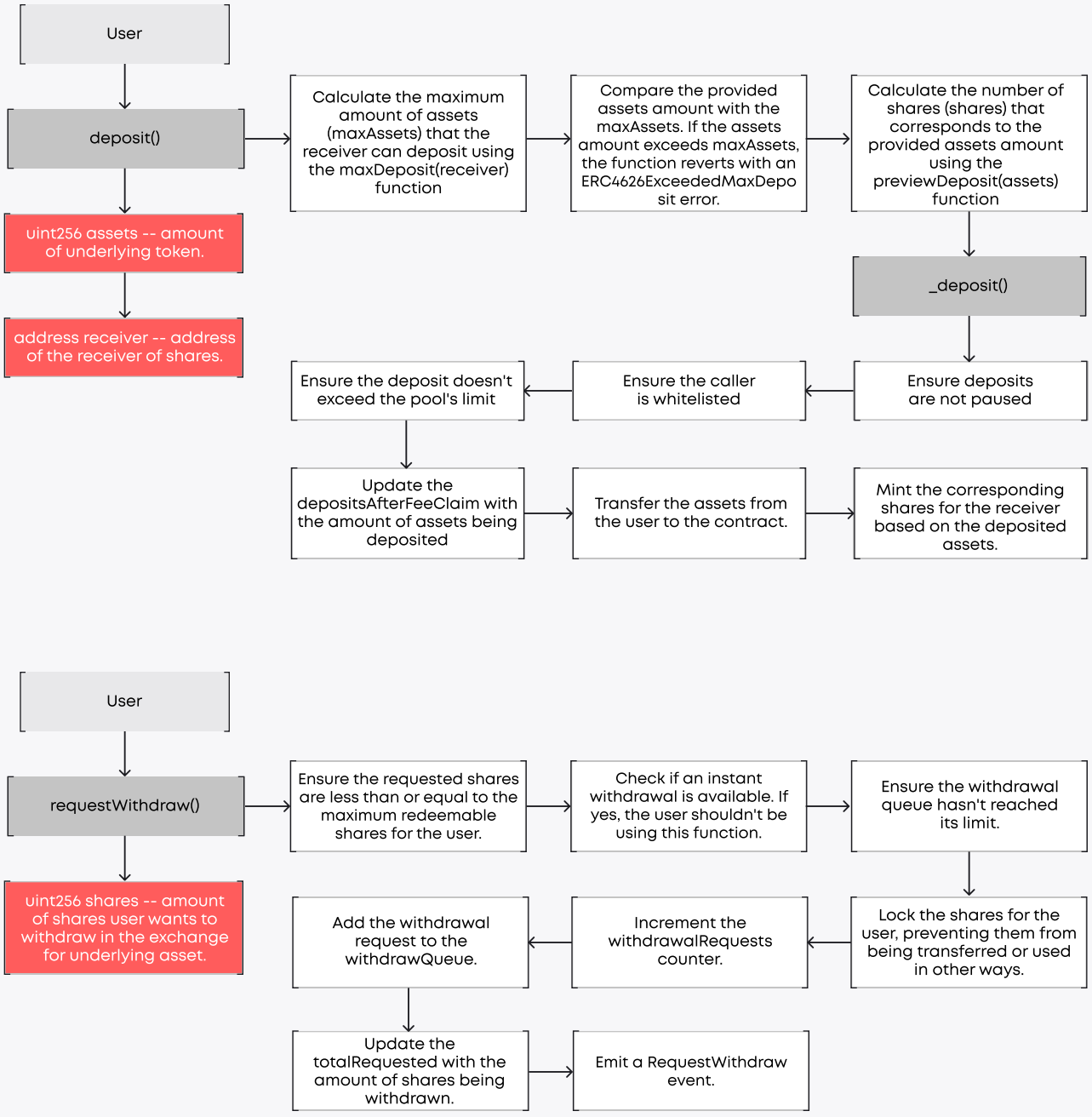
The audited deployment script is located at ./scripts/arbitrum/deploy-rebalancer.ts.
The script deploys 5 instances of Rebalancer.sol for: USDT, USDC.c, wBTCH, WETH, ARB.

The initial storage variables are initialized with the following values:

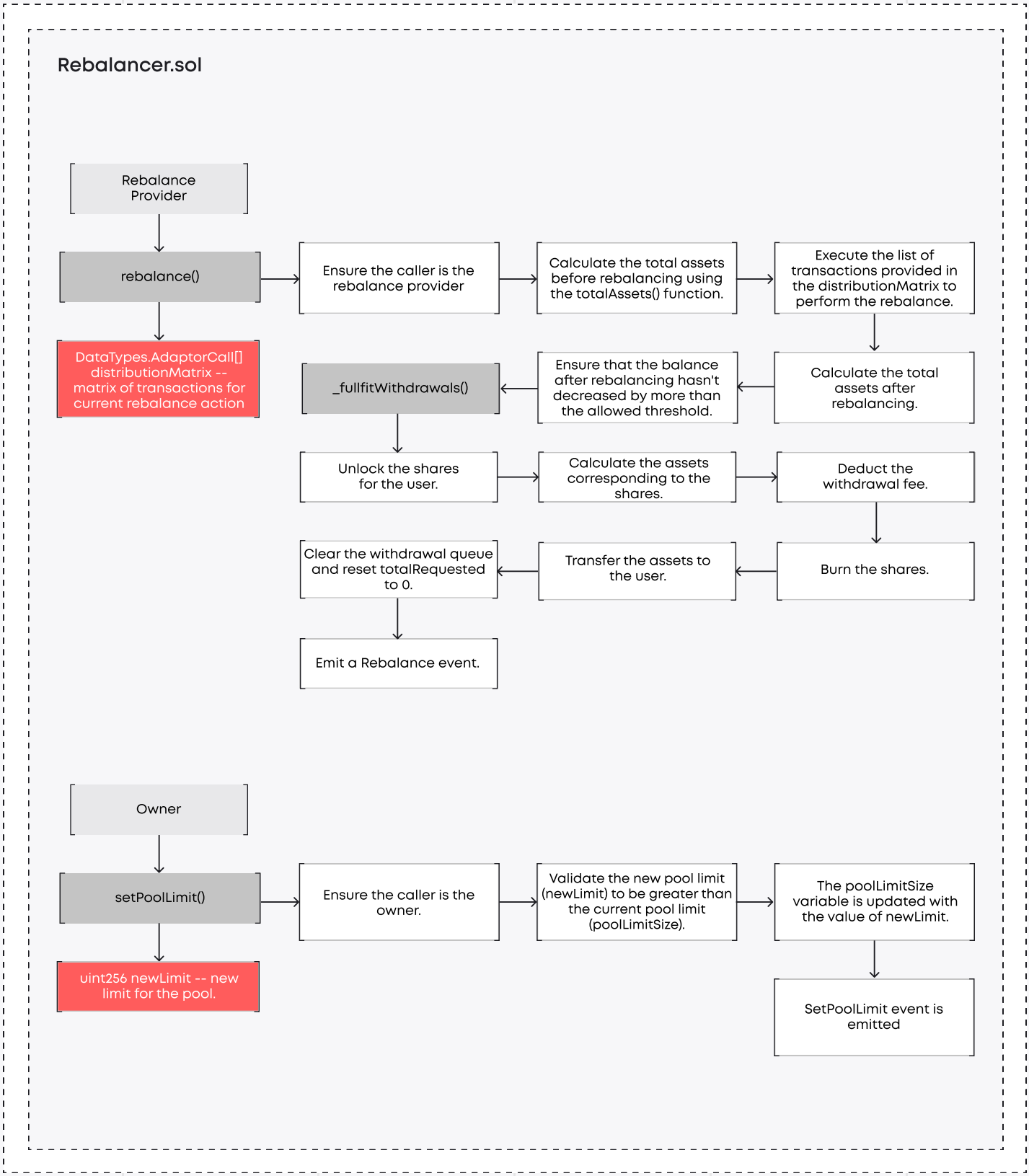
- asset: underlying asset
- name: name of the vault share token
- symbol: symbol of the vault share token
- positions: a list of protocols (positions) with which the vault can interact
- iTokens: a list of interest tokens that the vault can hold
- rebalanceProvider: a list of addresses that receive the role of rebalancer
- router: address of the price oracle
- whitelist: a list of users to be added to the whitelist
- poolLimits: maximum storage size (pool)

PROMETHIUM SCHEME

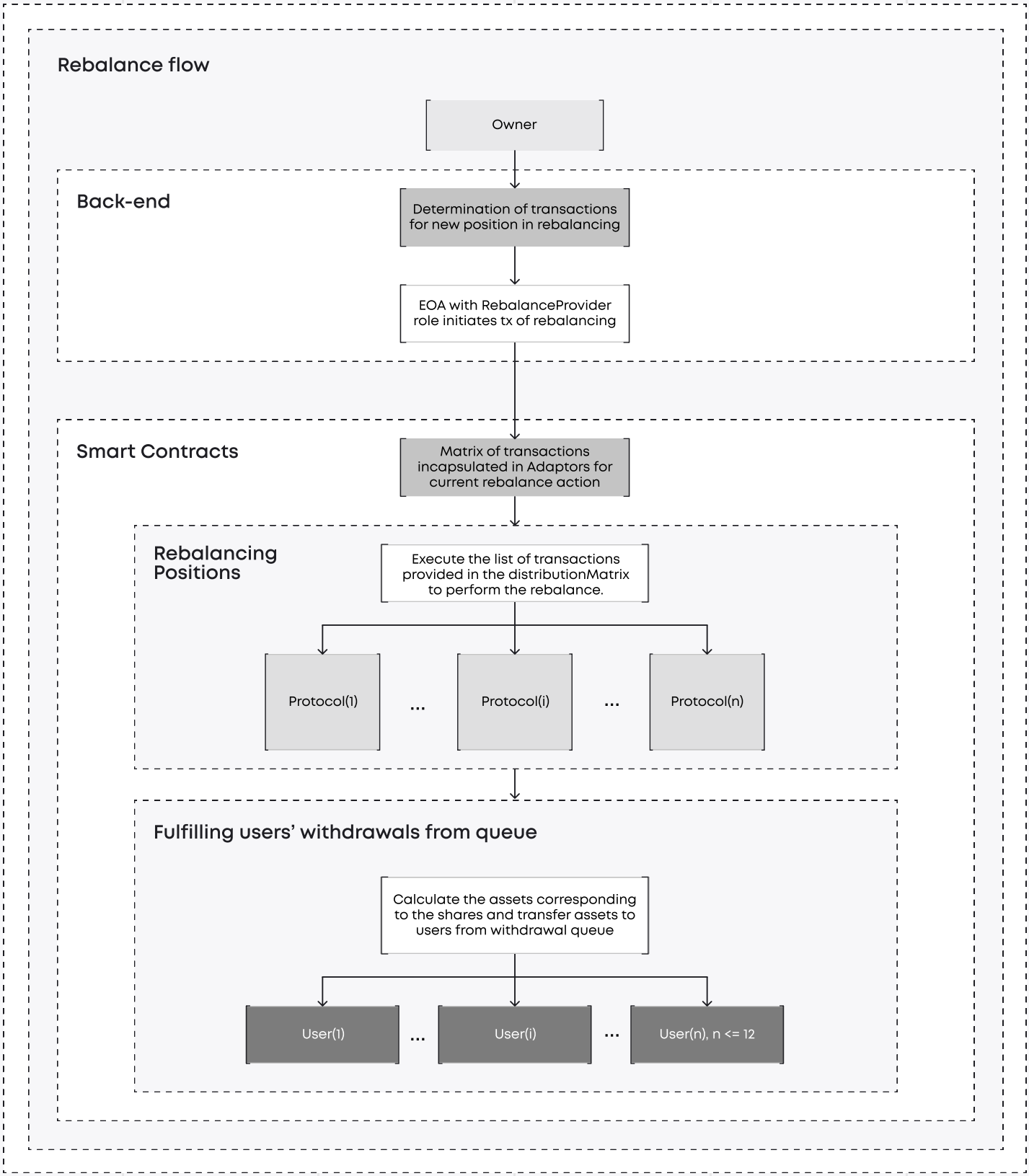
Rebalancer.sol



PROMETHIUM SCHEME

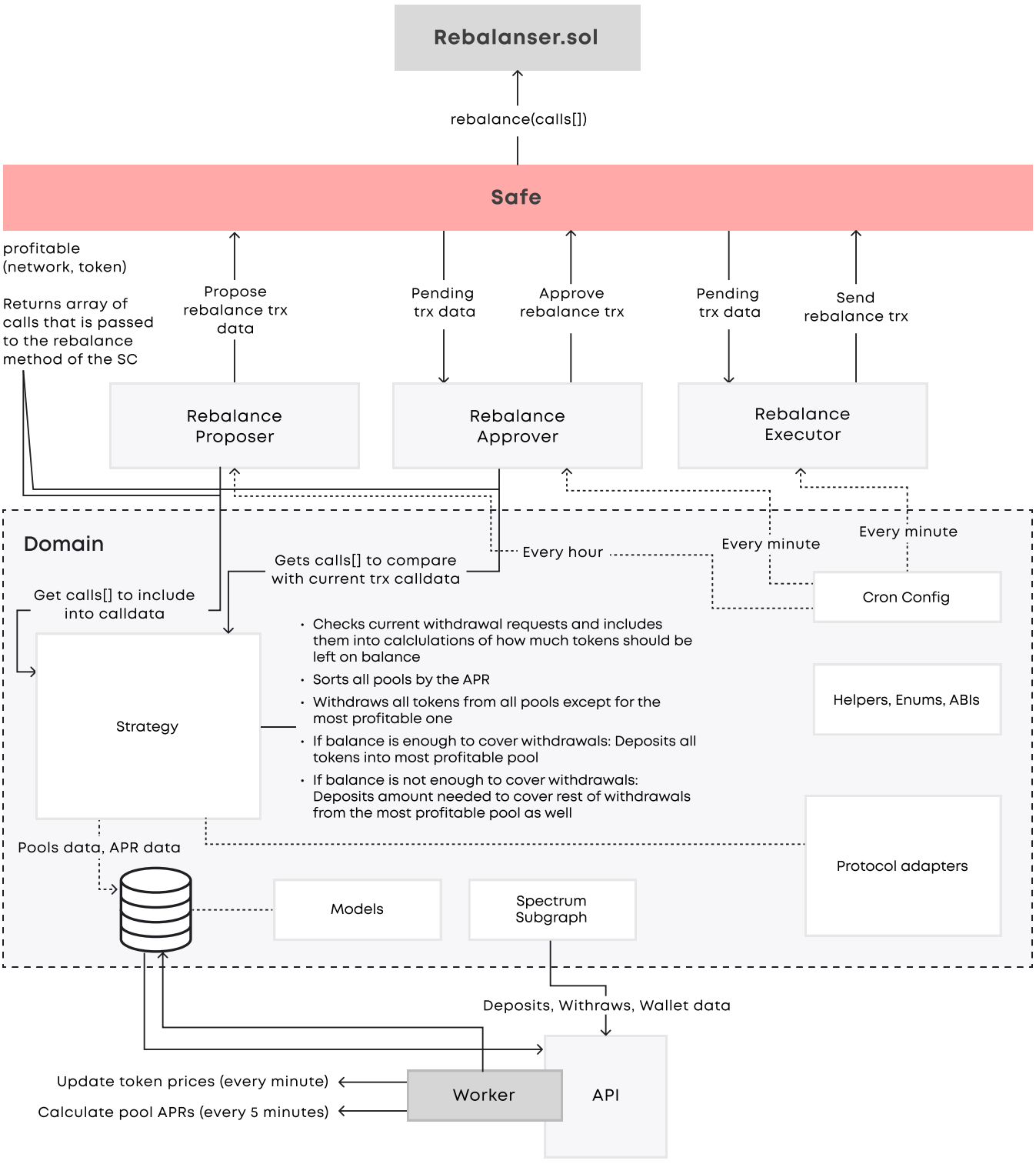


PROMETHIUM SCHEME



PROMETHIUM SCHEME

BE general scheme



LIST OF POTENTIAL THREATS

- Flashloan Attack Vulnerability

The Rebalancer system is potentially vulnerable to flashloan attacks during rebalancing and withdrawal processes. Specifically, the totalAssetsWithoutFee() function, which calculates the total asset amount using the PriceRouter, relies on third-party protocol exchange rates. The calculated asset amount will be incorrect if these rates are manipulated through a flashloan attack.

- Inaccurate Asset Conversion

The Rebalancer system uses the convertToAssets method to determine the exact amount of underlying assets for withdrawals. As per ERC-4626 security considerations, functions like convertToShares() and convertToAssets() should ideally be estimated and not relied upon for exact conversions. In the _fullfitWithdrawals() function, if convertToAssets doesn't provide an accurate conversion, users might receive an incorrect amount of assets during withdrawals. This threat arises from the potential inaccuracy of these conversion functions, which might not always reflect the true value, especially if they are used beyond their intended "view-only" purpose.

- Inflation Attack Susceptibility

The system's current implementation is potentially exposed to inflation attacks, where an attacker can manipulate the underlying asset balance. This type of attack, detailed in the provided article, primarily exploits rounding issues during initial deposits in the vault. Consequently, an attacker can siphon off funds from the vault, affecting other users. This threat is rooted in the system's handling of rounding and the lack of protective measures during initial deposits.

- Possible backdoor in the Adapter calls

In every rebalances, asset is put in the positions array, meaning that it can be put into rebalance matrix. Thus, every method on the token can executed during rebalance(). This open a backdoor for REBALANCE_PROVIDER_ROLE, as it can prepare an approval call, grant approval in asset from contract to any account. And this account will be able to transfer funds from Rebalancer. Issue is actual in case account with REBALANCE_PROVIDER_ROLE is compromised or its private key is exposed. To mitigate that issue, function _isExecuteLegal() was added in order to validate the adaptor call in rebalancing positions. _isExecuteLegal() determines the legitimacy of a specific call based on provided data. It checks if the data.adaptor matches a certain asset or if it's recognized as a protocol. For the asset match, it verifies if the call's function selector corresponds to the approve function of the IERC20 interface and if the spender is a recognized protocol. For the protocol match, it checks if the call's function selector aligns with either the deposit or withdraw functions associated with that adaptor. If neither condition is met, the function returns false and the whole rebalance-transaction will revert.

Note! The list of these threats represents potential issues that usually occur in similar protocols. None of them were confirmed during the tests, so they were not included in the findings list.

- Unauthorized Access and Single-point-of-Failure

The upgradability of the contracts, added during the last protocol upgrade, introduces a logic that could potentially create additional risks of single-point-of-failure. Thus, in case of unauthorized access to the deployers private key (or loss of the access to it), there is a high risk of the unauthorized upgrade which prevent further control over the protocol and bring threat to the kept funds.
Also, despite current roles system for contracts and KMS usage for backend works well, it depends on production setup and keys distribution

COMPLETE ANALYSIS. 1ST ITERATION

MEDIUM-1	✔Resolved

Possible Denial-of-Service attack.

Rebalancer.sol: requestWithdraw().

Anyone can request a withdrawal regardless of the amount of shares they will withdraw. Currently, it is even possible to pass 0 shares. As a result, an attacker can set up an automotive script that will fill the withdrawal requests array up to its limit before valid users are to request withdrawal. The issue is marked as medium-risk since implementing such an attack requires significant efforts and gas spending and doesn't lead to a loss of funds rather than the denial of service for users. Also, though it is still possible for owners to just make a rebalance and allow users to withdraw instantly, it affects the ability to earn yield since the funds will need to be stored on the contract before users withdraw them.

Recommendation:

Consider adding a minimum amount of shares a single user has to request for withdrawal.

Post-audit. Validation for the amount of shares to be greater than zero was added.

MEDIUM-2	✔Resolved

Fee is paid to a previous treasury in case it is changed.
Rebalancer.sol: setFee().
In case a previous treasury stored in the variable `FeeData` becomes deprecated or malicious and the owner decides to update it, the fee will still be transferred to the old treasury. Only after that, the new treasury will be set: the fee is paid at line 158, and the `FeeData` is set at line 159. However, the `FeeData` should be set after the fee payment as the percentage is also updated.

Recommendation:

Consider adding a separate setter for a treasury in order to set in before transferring the fees.

Post-audit. Separate setter for treasury was added.

LOW-1	✔Resolved

Fixed Solidity version should be used.

For now, all contracts use solidity ^0.8.9 - a floating version that does not correspond to best practices. Thus, the compilation may include unstable versions of compilers with major bugs introduced during intermediate releases. The contracts should be deployed with the same compiler version and options with which they have been most tested. Locking the pragma version helps ensure the contract is not accidentally deployed using a different version. Thus, it is recommended to use the last stable version - which is now 0.8.21.

Recommendation:

Use the last stable Solidity version.

Post-audit. Version 0.8.21 is now used.

LOW-2	✔Resolved

Parameters lack validation.

Rebalancer.sol: constructor(); Registry.sol: constructor(); PriceRouter.sol: initialize().
Either not all or none of the parameters passed to these functions are validated. This can lead to unexpected behavior in further use, which is why it is recommended to validate parameters. For example, addresses should be validated not to be equal to zero addresses and numeric parameters should be validated no to be equal to 0.

Recommendation:

Add the necessary validation.

Post-audit. All the necessary validations were added.

LOW-3	✔Resolved

Pool limit size cannot be decreased.

Registry.sol: setPoolLimit().
The pool limit can only be increased with the corresponding validation of the new pool limit to be greater than the current one. This might be a business logic feature, but pool size does not have a determined limit that would not be exceeded. If the pool limit size is set accidentally to a greater value than expected, it cannot be changed in the future. Therefore, it may result in unexpected vault behavior. It also should be mentioned that if it is not expected to decrease the pool limit, then it is more suitable for that function to have the name “increasePoolLimit()”, because it can only be increased.

Recommendation:

Validation for the new pool limit to be greater than the current pool limit was removed, and the pool limit can now be decreased.

Post-audit. Pool limit now can be decreased as well as increased.

LOWEST-1	✔ Resolved

Function visibility should be changed from “public” to “external”.

Rebalancer.sol: setPoolLimit(), setFee(), requestWithdraw();
PriceRouter.sol: initialize(), getTokenValue();
RBAC.sol: disableWhitelist();
Registry.sol: setPause()
It is recommended to set the visibility of these functions to external. This provides a better understanding of functions’ purpose and decreases the cost of execution. Also, some of the setters usually perform additional actions that are not necessary to be performed within other functions, such as the constructor. For example, setFee() function also performs claims of fees, which is redundant during deployment.
setPoolLimit and setFee functions are called in the constructor, and there are no more calls from inside the contract. However, setting the pool limit and fee directly in the constructor will be cheaper than calling the setPoolLimit() and setFee(). Therefore, setPoolLimit() and setFee() functions’ visibility can be changed to “external”.

Recommendation:

Set a value for pool limit in the constructor directly and change visibility for setPoolLimit().

Post-audit. Functions’ visibility was changed to “external”.

LOWEST-2	✔ Verified

Custom errors should be used.

All that has revert(), should revert custom error.
Starting from the 0.8.4 version of Solidity it is recommended to use custom errors instead of storing error message strings in storage and use “require” statements. Using custom errors is more efficient in terms of gas spending and increases code readability.

Recommendation:

Use custom errors.

Post-audit. Promethium verified that issue and decided not to change the codebase regarding that problem.

LOWEST-3	✔ Resolved

Lack of events.

Registry.sol: setPause();
Rebalancer.sol: _withdrawRequested(), event “Withdraw” of ERC4626 should be called;
Rebalancer.sol: setFee().
Essential functions such as setters should emit events to keep track of historical data changes. Also, in the case of _withdrawRequested(), it is recommended to call the native event of ERC4626 since some data aggregators may rely on these events.

Recommendation:

Emit events in the corresponding functions.

Post-audit. All the important events were added.

LOWEST-4	✔Resolved

Whitelist can't be enabled.

RBAC.sol: disableWhitelist().
The function provides the ability to disable the allowlist. However, there is no function for a reversed action to enable an allowlist. This means that once the allowlist is disabled, it can't be enabled back. The issue doesn't impact the security level of the contract, though it should be verified that once the allowlist is disabled, it can't be enabled. The issue is marked as Info since it may reflect the desired business logic choice. Thus it requires verification from the team.

Recommendation:

Verify that once the whitelist is disabled, it can't be enabled **OR** add an ability to enable whitelist.

Post-audit. The Promethium team verified that once the whitelist is disabled, it can't be enabled again.

LOWEST-5	✔Resolved

Variables should be marked as constants.

Registry.sol: POSITIONS_LIMIT, ITOKENS_LIMIT.
Variables are set only once during deployment and are not changed afterward. Thus, it is recommended that these variables be declared as constants to improve the contract's readability or add setters for these variables.

Recommendation:

Declare variables as constants **OR** add setters for them.

Post-audit. Variables are marked as constants now.

LOWEST-6	✔ Resolved

Visibility of storage variables is missed.

Rebalancer.sol: lastBalance, depositsAfterFeeClaim, withdrawalsAfterFeeClaim.
Visibility for the following variables is missed. Though, by default, the visibility of variables is private, it is recommended to mark the visibility explicitly to improve the code's readability.

Recommendation:

Mark the visibility of variables explicitly.

Post-audit. Variables have private explicit visibility now.

LOWEST-7	✔ Verified

Unused struct field.

DataTypes.sol: struct `withdrawRequest`, field `id`.
The field `id`is used only for an event WithdrawRequested and is not used for any contract logic. Though the id still may be essential for the protocol and used outside, this should be verified.

Recommendation:

Verify the necessity of the field `id`of struct `withdrawRequest`.

Post-audit. The Promethium team verified that that struct field is used by their back-end so it will remain.

LOWEST-8	✔Resolved

Storage optimization for constant variables.

Rebalancer.sol: variables `REBALANCE_THRESHOLD`, `WITHDRAW_QUEUE_LIMIT`, `feeDecimals`.
Since the variables are marked as `constant` and their values are known, you can compact the data so that they occupy only one box in the `bytes32` storage

Recommendation:

Change the data dimension for a variable: `REBALANCE_THRESHOLD` uint256 -> uint64;
`WITHDRAW_QUEUE_LIMIT` uint256 -> uint32;
`feeDecimals` uint256 -> uint32

Post-audit. Data dimension was changed for given variables.

LOWEST-9	✔Resolved

Change writing style for a constant variable.

Rebalancer.sol: `feeDecimals`.
This does not affect the security and operation of the protocol at all, but it is better to follow the general rules of naming style for constant variables. It will improve the readability of the contract.

Recommendation:

Rename `feeDecimals` to `FEE_DECIMALS`

Post-audit. `feeDecimals` was renamed to `FEE_DECIMALS`.

	DataTypes.sol **RBAC.sol** **Rebalancer.sol** **Registry.sol** **PriceRouter.sol**
✓ Re-entrancy	Pass
✓ Access Management Hierarchy	Pass
✓ Arithmetic Over/Under Flows	Pass
✓ Delegatecall Unexpected Ether	Pass
✓ Default Public Visibility	Pass
✓ Hidden Malicious Code	Pass
✓ Entropy Illusion (Lack of Randomness)	Pass
✓ External Contract Referencing	Pass
✓ Short Address/Parameter Attack	Pass
✓ Unchecked CALL Return Values	Pass
✓ Race Conditions/Front Running	Pass
✓ General Denial Of Service (DOS)	Pass
✓ Uninitialized Storage Pointers	Pass
✓ Floating Points and Precision	Pass
✓ Tx.Origin Authentication	Pass
✓ Signatures Replay	Pass
✓ Pool Asset Security (backdoors in the underlying ERC-20)	Fail

COMPLETE ANALYSIS . 2ND ITERATION

MEDIUM-1	✔ Verified

Rebalance proposer, approver and executor workers may run out of gas and cause the protocol to stop rebalance activities

The proposer, approver, and executor workers use the Safe API kit to (respectively) propose, approve, and execute rebalance transactions once per hour / once per minute, depending on worker configuration. This requires the sender to pay gas for the propose, approve, and execute transactions to reach the Safe.sol SC.
In the current implementation, there is no way of knowing that workers' balances are running low. This may result in transactions failing due to low gas and rebalance activities blocked for the whole protocol.

Recommendation:

We recommend having a monitoring system that will check workers balances with a certain frequency and notify protocol maintainers when worker balances get lower than a certain threshold.

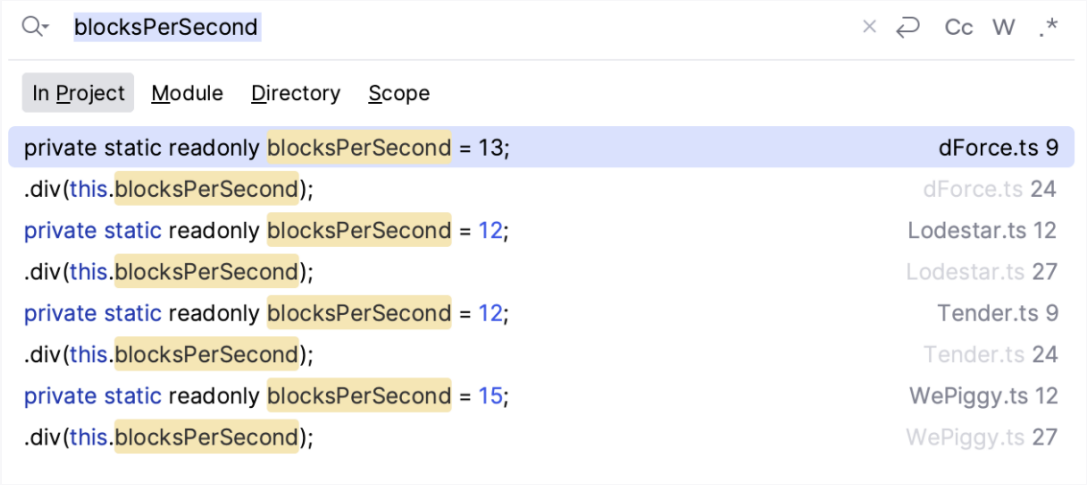
Post-audit:

The Promethium team verified the existence of telegram loggers for workers balances and other important logs.

MEDIUM-2	✔ Resolved

Protocol pool adapters contain different numbers of blocks per second, which affect APR calculations

Blocks per second parameter occurs in four pool adapters:
libs/domain/src/protocol-adapter/protocols/dForce.ts:9
libs/domain/src/protocol-adapter/protocols/Lodestar.ts:12
libs/domain/src/protocol-adapter/protocols/Tender.ts:9
libs/domain/src/protocol-adapter/protocols/WePiggy.ts:12



Recommendation:

Please review and unify blocks per second value, ~15 blocks / minute according to the arbitrum docs – as correctness of the logic of APR calculation is extremely important to the protocol in order to work correctly, as it is a key decision making factor in the rebalance process.The ideal way to configure it in non-ambiguous way would be to configure blocks per second value once per each network supported, alongside with other network configurations

Post-audit:

Promethium team updated the formula to use the number of blocks per year specific for each protocol.

MEDIUM-3	✔ Resolved

APR calculation formulas in dForce, Lodestar, Tender and WePiggy pool adapters may be incorrect

All of the mentioned adapters currently use following formula for APR calculation

```
const apr = (
  await Pool__factory.connect(
    poolAddress,
    providers[network]
  ).supplyRatePerBlock({ blockTag: blockNumber })
)
  .mul(this.secondsInYear)
  .div(this.blocksPerSecond);
```

Let's consider an example using following values:
rewardPerBlock = 2, blocksPerSecond = 15
Using the formula above to calculate the yield per minute:
yieldPerMinute = rewardPerBlock * 60 / blocksPerSecond = 8
Whereas this formula makes more intuitive sense:
yieldPerMinute = rewardPerBlock * 60 * blocksPerSecond = 2 * 60 * 15 = 1800
Similarly, calculating the annual yield would require multiplying the supplyRatePerBlock by the blocks per year value.

Recommendation:

Please consider changing the formula to the following one:

```
const apr = (
  await Pool__factory.connect(
    poolAddress,
    providers[network]
  ).supplyRatePerBlock({ blockTag: blockNumber })
)
  .mul(this.secondsInYear)
  .mul(this.blocksPerSecond);
```

Post-audit:

Promethium team updated the formula to use the number of blocks per year specific for each protocol. Though, still, the formula does not include the native rewards.

INFO-1	✔ Verified

Private keys of workers are stored in the environment variables

Currently, proposer, approver and executor workers make use of the private keys stored in the environment variables. We normally advise usage of key management services, such as **AWS KMS** for an additional security layer.

Recommendation:

Please verify that you are aware of the risks that come with manually managing private keys of the crucial protocol workers OR consider using a key management service instead.

Post-audit:

The Promethium team verified the necessity of KMS integration.

INFO-2	✔ Verified

Rebalance strategy includes sending all tokens to the single the most profitable pool

All of the APRs per pool are retrieved:
libs/domain/src/strategies.ts:58

```
const aprs = await Promise.all(
 internalPools[network][token].map((pool) =>
  ProtocolAdapter.getApr(network, token, pool, blockNumber)
 )
);
```

And this array is sorted by the APR value:
libs/domain/src/strategies.ts:64

Withdrawal calls are added for each pool except the most profitable one:
libs/domain/src/strategies.ts:90

```
pools.forEach((pool, index) => {
 if (BigNumber.from(pool.balance).isZero() || index == pools.length - 1)
  return;

 calls.push({
  adaptor: pool.contractAddress,
  callData: ProtocolAdapter.withdraw(network, token, pool,
pool.balance),
 });
 toWithdraw = toWithdraw.add(pool.balance);
});
```

All the pool balance is (except one to cover the withdrawals and the reserves), are deposited onto the most profitable pool:
libs/domain/src/strategies.ts:142

```
 calls.push({
  adaptor: pools[pools.length - 1].contractAddress,
  callData: ProtocolAdapter.deposit(
   network,
   token,
   pools[pools.length - 1],
   toSupply
  ),
 });
```

This leads to a straightforward rebalance strategy, in which all of the protocol assets are consolidated in a single (per token) most profitable pool, introducing little diversification or resilience.

Recommendation:

Please verify that 'greedy' strategy is your goal in this scenario and you don't plan to include a more robust strategy that would introduce a degree of diversification, f.e. 75/20/5 distribution among 3 most profitable pools.

Post-audit:

The Promethium team verified the "greedy" approach.

INFO-3	✔ Verified

Deployment strategy assumptions

We assume, that all of the back-end instances are deployed in an independent manner, meaning:

- Each service runs on its own separate infrastructure instance.
- Each service has their own domain package rolled out, the database is not shared, workers are independently calculating decision-making data like APRs on each instance etc.
- Each service has its own private key.

Recommendation:

Please let us know if the assumptions above are correct, we will resolve this issue accordingly.

Post-audit:

The Promethium team verified, that the services do not share any data directly with each other (other than with AWS KMS). Everything that is used for calculation is obtained by each service from the blockchain and Safe API on its own.

INFO-4	✔ Verified

Withdrawal waiting queue is only 10 places per hour

According to the Rebalance smart contract logic, all withdrawals that cannot be carried out immediately are to be placed in a Withrdrawal queue, that has a limited length of 10:

contracts/Rebalancer.sol:29

```
uint32 public constant WITHDRAW_QUEUE_LIMIT = 10;
```

contracts/Rebalancer.sol:113

```
function requestWithdraw(uint256 shares) external nonReentrant {
---
 require(withdrawQueue.length < WITHDRAW_QUEUE_LIMIT, "Withdraw
queue limit exceeded");
 lockedShares[msg.sender] += shares;

 withdrawalRequests++;
 withdrawQueue.push(DataTypes.withdrawRequest(msg.sender,
shares, withdrawalRequests));

 totalRequested += shares;
---
}
```

The queue is emptied in the _fullfitWithdrawals() function during the rebalance process.

The rebalance itself is only initiated only once per hour, according to the worker cron configurations:
libs/domain/src/cron-config.ts:3

```
export const rebalanceProposerCronExpression =
CronExpression.EVERY_HOUR;
```

This means that only 10 withdrawal requests can be submitted per hour for all of the protocol users.

Recommendation:

Please verify that this number is sufficient and that is the known business decision.

Post-audit:

The team verified the queue approach and has an upgrade for the system in roadmap.

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

PriceRouter

initialize contract
✓ Should be deployed (499ms)
getTokenValue method
✓ Should return the number of iTokens 1:1
✓ Should return an error for an unsupported iToken (76ms)
✓ Should return an error for an unsupported asset

Registry

initialize contract
✓ Should be deployed (567ms)
getPositions method
✓ Should return the protocol addresses
getITokens method
✓ Should return the IToken addresses
addPosition method
✓ Should return the IToken addresses (40ms)
✓ Should return an access error 'owner only'
addPosition method
✓ Should add a new protocol (39ms)
✓ Should return an access error 'owner only'
✓ Should return an error for an already existing protocol
✓ Should return an error of exceeding the limit (281ms)
removePosition method
✓ Should delete the existing protocol
✓ Should return an access error 'owner only'
addIToken method
✓ Should add a new IToken (45ms)
✓ Should return an access error 'owner only'
✓ Should return an error for an already existing IToken
✓ Should return an error of exceeding the limit (158ms)
removeIToken method
✓ Should delete the existing IToken (54ms)
✓ Should return an access error 'owner only'
✓ Should return a non-zero balance error (48ms)

setPause method
✓ Should be on pause
✓ Should not be on pause (38ms)
✓ Should return an access error 'owner only'
disableWhitelist method
✓ Should return the protocol addresses
✓ Should return an access error 'owner only'
✓ Should return an error for an already disabled whitelist
setPoolLimit method
✓ Should set a new limit pool
✓ Should return an access error 'owner only'
✓ Should return an error for the limit
setUserDepositiLimit method
✓ Should set a new user deposit limit
✓ Should return an access error 'owner only'
✓ Should return an error for the limit

Rebalancer

initialize contract
✓ Should be deployed (3482ms)
deposit method
✓ Should add a deposit (7261ms)
withdraw method
✓ Should withdraw assets (11092ms)
requestWithdraw method
✓ Should create a withdrawal request (13355ms)
✓ Should be an error of exceeding the limit (31473ms)
rebalance method
✓ Should make a rebalancing (10857ms)
claimFee method
✓ Should issue a performance fee (8515ms)
setFee method
✓ Should change the commission settings (460ms)
setFeeTreasury method
✓ Should change the address treasury (88ms)
addIToken method
✓ Should add a new iToken (126ms)
setPoolLimit method
✓ Should set a new limit pool (73ms)

- ✓ Should return an error for the limit

totalAssets method

- ✓ Should return the total assets supply (8364ms)

Scenario 1: inflation attack

- ✓ Should not be profitable (16347ms)

Scenario 2: flash loans attack

- ✓ Should not affect total supply (1318ms)

49 passing (2m)

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY PROMETHIUM TEAM

RBAC contract

Deployment

- ✓ Should return the correct rebalance provider role
- ✓ Should return the correct whitelisted role

Rebalancer contract

Deployment

- ✓ Should set the correct performance fee
- ✓ Should set the correct withdrawal fee
- ✓ Should set the correct treasury
- ✓ Total assets should be equal to 0 after deployment (50ms)

Total Assets function

- ✓ Should correctly calculate total assets (1182ms)
- ✓ Should correctly calculate total assets when the part of the token deposited to the pool (2200ms)

Get available fee function

- ✓ Performance fee should be equal to 0 after the deposit (1044ms)
- ✓ Should correctly calculate performance fee (1076ms)
- ✓ Performance fee should be 0 after claim (160ms)
- ✓ Performance fee should be equal to 0 when pool balance become lower (1975ms)

Total assets without fee function

- ✓ Should correctly calculate total pool balance (1199ms)

Rebalance function

- ✓ Should revert when to the rebalance provider is trying to execute (1189ms)
- ✓ Should revert when balance became too low after rebalance (1587ms)
- ✓ Should emit after rebalance (2414ms)

Request withdrawal function

- ✓ Should revert when user is trying to reedeem more shares than he has (1108ms)
- ✓ Should revert when the balance of pool is enough for instant withdraw (2466ms)
- ✓ Should not allow request when the queue is full (3528ms)
- ✓ Should lock shares after withdraw (2407ms)
- ✓ Should set the correct user maxReedem after request (2381ms)
- ✓ Shouldn't allow to transfer when shares are locked (2561ms)

- ✓ Should increase the total amount of shares to withdraw while next rebalance (2244ms)
- ✓ Should emit after withdrawal request (2240ms)
- ✓ Request id should be unique (2628ms)

Fullfit withdrawals function

- ✓ Should not left locked shares after completing withrawal (4312ms)
- ✓ Withraw queue should be empty after rebalance (4870ms)
- ✓ Should emit after a completing a request (4164ms)
- ✓ Total requested shares should be equal to zero after rebalance (4319ms)
- ✓ Should burn user shares after completing a withdrawal (4223ms)

Claim fee function

- ✓ Should revert if not the owner is trying to claim the fee
- ✓ Should claim the correct amount of tax (167ms)
- ✓ Should emit after claiming a fee (163ms)

Deposit overriding

- ✓ Should not allow to add itoken if the router does not support it
- ✓ Should revert when not whitelist person is trying to deposit (42ms)

Set fee function

- ✓ Should revert when not the owner is trying to set the fee
- ✓ Should revert when trying to set performance fee higher than max
- ✓ Should revert when trying to set withdrawal fee higher than max
- ✓ Should claim fee before setting the new value (667ms)
- ✓ Should change the fees

Registry contract

Deployment

- ✓ Should not set the correct amount of position
- ✓ Should not set the correct positions
- ✓ Should set the corect amount of iTokens
- ✓ Should set the corect iTokens
- ✓ Should set the correct rebalancer role after
- ✓ Should set price router address
- ✓ Should whitelist wallets

Add position function

- ✓ Should revert when not the owner is trying to set a new position
- ✓ Should revert if the adaptor is already added
- ✓ Should add a non itoken position
- ✓ Should emit an event after adding a position
- ✓ Should not allow to add more positions than limit (449ms)

Remove position function

- ✓ Should remove the position from allowed adaptors list (60ms)
- ✓ Should revert if not the owner is trying to remove position
- ✓ Adaptor should become not allowed after removing a positions (80ms)
- ✓ Should emit after removing position (77ms)
- ✓ Shouldn't remove any extra position (82ms)

AddIToken function

- ✓ Should revert when not the owner is trying to set a new Itoken
- ✓ Should revert if the adaptor is already added
- ✓ Should add an iToken
- ✓ Should not allow to add more positions than limit (284ms)
- ✓ Should emit an event after adding a position

Remove IToken function

- ✓ Should revert if not the owner is trying to remove iPosition
- ✓ Should revert if the owner is trying to remove position with non zero balancer
- ✓ Shouldn't remove any extra itoken (146ms)
- ✓ Adaptor should become not allowed after removing an itoken (178ms)
- ✓ Should emit after removing an iToken (138ms)

Deposites pause function

- ✓ Should revert if not the owner is trying to pause
- ✓ Should pause
- ✓ Should unpause (77ms)

72 passing (1m)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
DataTypes.sol	100	100	100
RBAC.sol	80	58.33	80
Rebalancer.sol	92.59	79.17	89.47
Registry.sol	100	95.45	100
PriceRouter.sol	0	0	0

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.