SMART CONTRACTS REVIEW

zokyo

September 23rd 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING RHO LABS

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 2 High issues: 2 resolved = 0 points deducted
- 6 Medium issues: 4 acknowledged and 2 resolved = - 8 points deducted
- 19 Low issues: 11 acknowledged and 8 resolved = - 11 points deducted
- 7 Informational issues: 3 acknowledged and 4 resolved = 0 points deducted

Thus, 100 - 8 - 11 = 81

TECHNICAL SUMMARY

This document outlines the overall security of the Rho Labs smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Rho Labs smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Rho Labs team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Rho Labs repository:
Repo: https://github.com/RhoLabs/rho-collateral-contracts

Last commit - 033c8b6e9d9c8ce5cff201f7666fe48c1d78488f

Contracts under the scope:

- ./configuration/ContractProvider.sol
- ./configuration/IContractProviderErrors.sol
- ./configuration/MainContracts.sol
- ./configuration/Configurator.sol
- ./configuration/IContractProvider.sol
- ./configuration/IConfigurator.sol
- ./libraries/IterableIdAddressSet.sol
- ./libraries/ContractIds.sol
- ./libraries/IIterableIdAddressSetErrors.sol
- ./utils/ITimeProviderErrors.sol
- ./utils/IWrappedNativeToken.sol
- ./utils/IAddressValidatorErrors.sol
- ./utils/IInterfaceValidationErrors.sol
- ./utils/IPausable.sol
- ./utils/AddressValidator.sol
- ./utils/TimeProvider.sol
- ./depositary/IDepositary.sol
- ./depositary/libraries/DepositaryLogic.sol
- ./depositary/storage/DepositaryStorage.sol
- ./depositary/storage/IDepositaryStorage.sol
- ./depositary/storage/DepositaryStorageParamKeys.sol
- ./depositary/storage/IDepositaryStorageErrors.sol
- ./depositary/Depositary.sol
- ./depositary/IDepositaryErrors.sol
- ./access-control/Roles.sol
- ./access-control/DepositaryRelatedRoles.sol
- ./access-control/IAccessControlManagerErrors.sol

- ./access-control/AccessControlManager.sol
- ./access-control/IAccessControlManager.sol
- ./router/IRouterErrors.sol
- ./router/libraries/RouterLogic.sol
- ./router/IRouter.sol
- ./router/Router.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Rho Labs smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contract/s by industry leaders.
03	Thorough manual review of the codebase line by line.

Executive Summary

Rho labs has taken a unique approach to crypto-native interest rate derivatives in the context of lending-borrowing rates and staking rewards. They will empower users to hedge and profit from shifts in the crypto native rates giving tools and tactics to develop sophisticated and personalised trading strategies. Ultimately, Rho allows users to navigate market fluctuations through funding rates, reducing risks or profiting from market shifts and leveraging variations in staking rates to allow long term returns for holders through attractive yields or protect against declining network rewards.

Zokyo was tasked with the security review for the Rho collateral portion of the protocol which included the depository, router, the access control manager and configuration contracts including all their related libraries.

Overall, the code is well structured and well documented however, there are multiple layers of abstraction and may require users who are interested in how the protocol works to trace through such layers in order to gain a better understanding of the protocol as a whole. Because of these layers of abstraction, the router acts as the user facing component of the protocol. Users can freely deposit through the router however, should they wish to withdraw they are required to go through a privileged address called the withdrawal operator who will pass the users signature as a parameter to the depositary in order to process a withdrawal request. A similar approach is done for other actions of the protocol and all have different privileged operators handling these functions such as transferring internally (internal transfer operator), handling treasury operations (depositary treasury operator), or the addition and deletion of assets and depositaries including the running of contract upgrades (handled by contract admins).

Following the security review, the vulnerabilities discovered ranged from critical down to informational which are findings that dont have any security implications such as gas optimisations or best engineering practices. Most of these findings revolved around inappropriate usage of signatures, business logic errors due to lack of accounting, loss of assets and griefing. The informational findings were mostly revolving around best engineering practices.

The Zokyo security team recommends that the team at Rho Labs carefully reviews the findings and if any, the proof of concept exploit code attached and implements the suggested fixes. Following this, a fix review will take place to ensure that no further bugs are introduced and to ensure that the proof of concept code no longer works on the described pieces of code. The team at Zokyo wishes Rho Labs all the best for their deployment to the mainnet.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Rho Labs team and the Rho Labs team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

- **Critical**

  The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

- **High**

  The issue affects the ability of the contract to compile or operate in a significant way.

- **Medium**

  The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

- **Low**

  The issue has minimal impact on the contract’s ability to operate.

- **Informational**

  The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Missing Signatures From transferInternally May Allow User Assets To Be Moved Without Authorization	High	Resolved
2	Depository, when updated with new storage, enables signature replay	High	Resolved
3	Underflow In The Withdrawal Function of The Depositary Allows Users To Withdraw More Than Once	Medium	Acknowledged
4	Insufficient Validation On Owner When Withdrawing Allowing Users To Cross Withdraw	Medium	Acknowledged
5	ETH can not be credited by the treasury supplier using the method creditTreasury()	Medium	Resolved
6	If an asset is removed, deposits within that asset will be lost	Medium	Acknowledged
7	Asset's balance limit is not deleted when an asset is removed	Medium	Acknowledged
8	Whitelist Policy Inconsistency Lets Funds Flow to Non-Whitelisted Addresses	Medium	Resolved
9	Accidentally Setting The Maximum Withdrawal Id When Withdrawing Will Disallow A User From Withdrawing In The Future	Low	Acknowledged
10	Gas griefing attack on the method withdraw()	Low	Acknowledged
11	User asset balance can be set greater than the assets' assetBalance	Low	Acknowledged
12	Replayable Signatures Across Chains and Contracts	Low	Acknowledged

#	Title	Risk	Status
13	Zero-sum invariant might not hold, leading to users not being able to withdraw	Low	Acknowledged
14	An asset's assetBalance can be greater than the assetBalanceLimit	Low	Acknowledged
15	Signature replay is possible when the depository's chainID is not checked	Low	Acknowledged
16	Wrong Reliance on block.chainid in validateDepositary Function	Low	Resolved
17	Check nonZero address is missing	Low	Resolved
18	Using not locked solidity versions can lead to unexpected behaviors	Low	Resolved
19	Zero-Address Receiver Can Permanently Lock Funds (Deposit)	Low	Resolved
20	Native-Token Withdraw to Zero Address Burns ETH	Low	Resolved
21	Emergency Drains Blocked While Paused	Low	Resolved
22	Depositary Removal Can Break Operations	Low	Resolved
23	Asset Removal "Freeze" (Funds Lock)	Low	Acknowledged
24	Internal Transfer from == to Is a No-Op (Gas Waste)	Low	Resolved
25	Deposit Receiver Hygiene (Optional Safety)	Low	Acknowledged
26	Compromised Withdrawals Operator + Phished Owner Signature Can Drain Pool (No Per-User Solvency)	Low	Acknowledged
27	Treasury Credits Ignore the Asset Cap	Low	Acknowledged
28	Centralized behaviour allows for direct access to user funds via admin wallets	Informational	Acknowledged

#	Title	Risk	Status
29	Method transferInternally() can have `to` and `from` as the same address	Informational	Resolved
30	Provider Return Values Not Validated	Informational	Acknowledged
31	Withdraw Parameter Order Mismatch (Interface vs Implementation)	Informational	Resolved
32	Deadline Type Inconsistency	Informational	Resolved
33	Signature Domain Hardening (Defense-in-Depth)	Informational	Acknowledged
34	Negative Sender Balance on Internal Transfers (By Design)	Informational	Resolved

Missing Signatures From transferInternally May Allow User Assets To Be Moved Without Authorization

Description

For some function calls, signatures are required from the user in order to prove that they are the transaction initiator and that they give permission for the relative address (such as the withdrawal operator or in this case, the internal transfer operator) to act on their behalf; however, `transferInternally` does not impose any signatures at all.

Impact

The lack of signatures for internally transferring funds means that user funds can potentially be moved internally to another user without their permission. This was rated a High in severity because this may result in the theft of funds in addition to tampering with user assets without their permission. In addition to this, a compromise of the internal asset operator may allow this address to move funds internally freely to themselves and eventually withdraw.

Recommendation:

Its recommended that signatures are imposed for all operations especially internal transfers as this acts as “permission” for the relative operator to perform no more and no less than what is defined within the signature itself - ie. amount, asset, to.

Client’s comment:

We have decided to remove the internal transfer functionality, as it is not currently being used.
https://github.com/RhoLabs/rho-collateral-contracts/commit/a6af3edce61c9037a079d757a7d6a8dc5a702ec1

Depository, when updated with new storage, enables signature replay

Description

In the contract ContractProvider.sol, there are 2 methods that allow the owner to update an existing depository using depositoryId. The updated depository will have the same depositoryId, but the depository address will be changed.
Since a new depository has to be deployed for this update, this allows the new depository to have a new depository storage as well. DepositoryStorage stores one very important state regarding the withdrawal process, i.e withdrawalIds per user.
The withdrawal IDs basically act as a nonce for withdrawal signature params signed by the users. A withdrawal ID can never be repeated for a user, so that an already used signature can't be replayed to withdraw again.

However, when a depository is updated with a new depository storage, the withdrawal IDs are reset to 0 again. Because of this, the following check will fail to allow signatures that have already been used to withdraw.

```
if (withdrawalId <= lastWithdrawalId) {
      revert IDepositaryErrors.WithdrawalAlreadyProcessed(owner,
withdrawalId, lastWithdrawalId);
    }
```

Impact

Signature replay has a critical impact as it allows a user to withdraw again from the repository. This is a direct loss of funds for the protocol.

Recommendation:

Add a check in the method updateDepository and the method emergencyUpdateDepository to ensure that when a depository is updated, the depository storage is the same.

Fix:

Client added a check in the updateDepository method to ensure that depository storage remains the same, but still allows the emergencyUpdateDepository() method to allow the update of a depository with a new depository storage.

Client's comment:
We've added a check to preserve the previous storage during regular updates.
However, we believe that in emergency mode, we should retain the ability to update the storage smart contract to restore the system's functionality in case of force majeure circumstances.

Auditor's comment:
Client needs to prepare a migration plan before allowing such an update in emergency mode so that not only are all assets safely transferred to the new depository storage, but the state is not reset as well, especially the withdrawal IDs for users.

Underflow In The Withdrawal Function of The Depositary Allows Users To Withdraw More Than Once

Description

Users can freely deposit into the protocol via a depositary of their choice through the router. In addition to this, should they feel like they wish to withdraw, they can sign and submit a withdrawal request which is executed by the withdrawals operator however, because userAssetBalance and its corresponding variables are signed integers, a malicious user can underflow the balances by simply submitting another signature with an incremented withdrawalId, thus processing an illegitimate withdrawal.

Impact

This was rated critical because of how easy it is to trigger the bug in addition to this resulting in the blatant theft of funds from the protocol and its users.

Proof of Concept

The proof of concept below outlines a scenario where alice and bob deposit into the protocol. Alice withdraws once, changes the withdrawal id, signs again and asserts that funds are successfully stolen in addition to a successful underflow:

https://gist.github.com/chris-zokyo/5beae584d50714e4484fe5e571815d25

Recommendation:

It's recommended that the depositary logic is refactored so that all variables to do with user balances (especially in all layers of the `withdraw` function) are modified to uint256 datatypes as there is no reason for user balances to be signed. Should this value track profit and loss, create a separate variable for PnL and assert against this value if users have enough to withdraw. This also includes functions which operate on user balances. Finally in addition to the above, sanity checks against user balances should be made before finalising the withdrawal.

Client's comment:
Our backend validates how much a user is actually allowed to withdraw.
After the first valid request, the corresponding amount is deducted from the user's balance, and any subsequent request — regardless of its ID — will fail because the off-chain system no longer sees sufficient funds in the user's account.
Supporting negative balances for users is essential; otherwise, it would never be possible to withdraw more from the exchange than was originally deposited.

Insufficient Validation On Owner When Withdrawing Allowing Users To Cross Withdraw

Description
The withdraw function for a depositary can be accessed via the router in order to pull funds out of the protocol. This function is only accessible via the withdrawalsOperator; however, users are still required to sign their own transactions in order to process withdrawals. A user may be able to control the owner parameter for the withdrawal function in order to extract funds that may not rightfully be theirs.

Impact
A malicious user may withdraw another user's funds by passing their own signature. This was rated high in severity as opposed to critical because this relies on the malicious user having some kind of debugging experience in order to change parameters to trigger the bug.

Proof of Concept
The proof of concept below outlines a scenario where alice deposits into the protocol and bob attempts to withdraw by matching the owner to the ecrecovered user. It asserts that bob has successfully pulled funds from the contract in addition to asserting that alice's virtual balance is 10 ether (even though there is zero funds in the contract):

https://gist.github.com/chris-zokyo/8073b029bcf4c43b1c2a08d3d295cd10

Recommendation:
Assuming the double withdrawal function has been fixed, validate that the owner of the resulting signature has enough funds before processing the withdrawal request.

Hacker's Notes

This attack assumes that ordinary users can sign the transaction and pass their signatures and other parameters to the withdrawal operator via the frontend. This can easily be done with browser debuggers such as Burp Suite, Caido or Postman.

Client's comment:
The withdrawal operator is not simply a frontend component that forwards user requests to the blockchain. Instead, it is a fundamental part of the system, which verifies signatures, performs margin account risk checks, and validates whether the user has sufficient balance available for withdrawal.
In our architecture, smart contracts act only as depositories; they do not track profit and loss, open positions, or other essential data required to determine the actual withdrawable balance of a user.
This responsibility lies with the withdrawal operator. Therefore, if a user attempts to withdraw funds, they can only withdraw the balance that is legitimately available to them - not any arbitrary amount they specify.

ETH can not be credited by the treasury supplier using the method creditTreasury()

Description

In the contract Router.sol, the method creditTreasury() allows the treasury supplier to credit allowed assets directly to the depository. Also, from the RouterLogic.creditTreasury() method, it is clear that ETH should be allowed to credit as well. But the creditTreasury() method is missing the `payable` keyword.

Impact

Since this is a requirement for the protocol to deposit ETH as well, the missing payable keyword does not allow the caller to credit ETH directly, leaving them with the option to swap to WETH and then deposit.

Recommendation:

Add the `payable` keyword in the method creditTreasury().

Client's comment:

Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/2b5f15519e42a09fd598c832c1aa4bbce03039ad#diff-88942899297fe99f158ad801aba13d674e8cdbd00984ae75560a1aec98278a61R307

If an asset is removed, the on-chain accounting of deposits in that asset will be lost

Description
If users deposit a certain asset to the protocol and the asset later gets removed by the owner/admin, users will not be able to withdraw the previously deposited asset.

Impact
Users will lose their assets as they would be unable to withdraw them in the described scenario.

Proof of Concept
The below scenario outlines a situation where users deposit into the protocol, the admin removes an asset resulting in the blocking from withdrawing. This was rated a Medium in severity because users may only be temporarily blocked as admins can re-add assets or depositaries.

https://gist.github.com/chris-zokyo/fb4f547cfdb5a30513844071ef156510

Recommendation:
It's recommended that the virtual balancing of depositaries are validated for a corresponding asset before being able to remove assets or depositaries from the protocol. Alternatively, create a rescue function to rescue tokens from removed depositaries and assets.

Client's comment:
The finding's wording is inaccurate: user funds are neither deleted nor lost.
In our current implementation, the "remove asset" action functions as a temporary suspension of that asset's withdrawal route (and of new deposits as well), intended for unforeseen or emergency situations such as chain instability or token anomalies.
All user balances remain fully accounted for in our smart contract, and once the asset is re-enabled, users can withdraw their funds as usual.

Asset's balance limit is not deleted when an asset is removed

Description

The `removeAsset` function within `DepositStorage.sol` is used to remove assets. However, when removing an asset it does not set its balance limit to `0` again.

```
function removeAsset(address asset) external onlyViaRelatedDepositary {
    bool isRemoved = _assets.remove(asset);
    if (isRemoved) {
      emit AssetRemoval(msg.sender, asset);
    } else {
      revert AssetDoesNotExist(address(asset));
    }
  }
```

Impact

The `_assetToBalanceLimit` variable is read and used by other contracts within the whole protocol to perform important actions. Not updating the variable to `0` may lead to unexpected behaviors.

Recommendation:

When removing and asset, set its balance limit to 0 again.

Client's comment:

After an asset is removed or added, we still retain the ability to adjust the maximum TVL limit through the configurator. This behavior is intentional by design.

Whitelist Policy Inconsistency Lets Funds Flow to Non-Whitelisted Addresses

Description

When the user whitelist is enabled, `Router.deposit` correctly blocks deposits from or to any address that isn't whitelisted. However, the outbound paths don't enforce the same rule:

- `Router.withdraw` allows sending funds to a non-whitelisted receiver.
- `transferInternally` credits balances to a non-whitelisted target.

  This creates a policy bypass: an approved (whitelisted) user can move value to an unapproved address via withdraw or internal transfer even though direct deposit to that address is forbidden.

Scenario :

- Whitelist on. `alice` is whitelisted; `eve` is not.
- Deposit checks work:
  - `eve → alice` deposit reverts (sender not whitelisted).
  - `alice → eve` deposit reverts (receiver not whitelisted).
- Outbound checks don't:
  - `withdraw(owner=alice, receiver=eve, …)` succeeds (eve gets tokens).
  - `transferInternally(alice → eve, …)` succeeds (eve's internal balance increases).

Proof of concept : https://gist.github.com/adeshkolte/49f4e281b2a21b5691f9c4030254c5f5

Impact:

Funds can be routed to non-whitelisted addresses despite whitelist being "on".
Off-chain systems assuming "only whitelisted users can receive value" will misreport.

Recommendation:

If whitelist should apply to all counterparties:

In withdraw: require(isDepositaryUser(depId, owner) && isDepositaryUser(depId, receiver)).
In transferInternally: require(isDepositaryUser(depId, from) && isDepositaryUser(depId, to)).

Client's comment:
Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/2c476a179b3291fe52a780124d9676ad3b76f00a

LOW-1 | ACKNOWLEDGED

Accidentally Setting The Maximum Withdrawal Id When Withdrawing Will Disallow A User From Withdrawing In The Future

Description
When users withdraw from the protocol a withdrawal Id is passed to the function in order to complete the request. The withdrawal id acts as a nonce of sorts to assess whether a withdrawal has already been processed; however, these ids are arbitrary and supplied which can lead to accidents.

Impact
Users who accidentally provide the maximum withdrawal id to the withdrawal operator will never be able to use the withdrawal function again. This was rated low in severity due to the likelihood this can cause any catastrophic damage in addition to the requirement of the user hurting themselves, however, this issue still has security implications as described above.

Recommendation:
It's recommended that withdrawal ids are incremented by one automatically on a user by user bases via a mapping instead of being passed to the withdraw function.

Client's comment:
The withdrawal ID acts as an idempotency key.
We didn't enforce strictly incremental values, as the sequence is managed off-chain by the backend on a per-user basis.
This design also leaves room for future scenarios where withdrawals may occur across different chains.
The main goal is to ensure that a specific withdrawal cannot be executed twice.

Gas griefing attack on the method withdraw()

Description
In the contract Router.sol, the withdraw() method allows a user to sign the parameters to withdraw from the depository, but the actual transaction is sent by the withdrawal operator account.

Since the withdrawal operator account pays for every withdrawal, a user can withdraw very small amounts, as there is no minimum withdrawal amount check, forcing the withdrawal operator to pay gas fees multiple times.

Impact
While this is not a direct loss of funds from the depository, the protocol still has to lose funds on gas fees.

Recommendation:
Add a minimum withdrawal amount check.

Client's comment:
The client said that to mitigate this issue, they have introduced an additional withdrawal fee that the withdrawing user pays. This fee is set in a way that covers the gas costs of the operation.
On top of that, the Withdrawal Operator itself has built-in limits on acceptable gas prices, which further protect against such scenarios.

User asset balance can be set greater than the assets' assetBalance

Description
In the contract Router.sol, the method transferInternally() allows the internal transfer operator to update user asset balances. Since there is no check on how much amount will be deducted from the `from` address and added to the `to` address, the method can be used to pass any amount even greater than the assetBalance.

Since the user asset balance can be -ve, any amount can be deducted, making the `from` address to have the user asset balance hugely -ve and the `to` address to have the user asset balance hugely +ve, even greater than the assetBalance for that asset.

Impact
Although there is no security issue on-chain as per the logic, the off-chain components might utilise the user asset balances, and this can lead to unexpected issues.

Recommendation:
Add a check to ensure that the `amount` passed in the `transferInternally` method is less than the assetBalance.

Client's comment:
Client mentions that in practice, their off-chain system does not rely on the per-user balances stored in the smart contracts. They only operate with deltas for deposits and withdrawals. At this point, the user balances on-chain are more of a historical artefact, and they don't actually need to perform internal transfers.
The functionality was added just in case such a need arises in the future. It's not strictly necessary to enforce that an individual user's balance cannot exceed the total asset balance held by the contract.

Replayable Signatures Across Chains and Contracts

Description
The current signature scheme does not restrict where a signed message can be executed. As a result, signatures generated for one chain or contract can potentially be replayed on a different chain or with a different contract that uses the same signing logic. This exposes the protocol to replay attacks across multiple environments.

Impact
An attacker could reuse a valid signature from one deployment (e.g., on Chain A) to execute the same action on another deployment (e.g., Chain B), or with another contract sharing the same logic. This could result in unintended transactions being processed, including unauthorized transfers or state changes, depending on how the signature is validated.

Severity is set to Low as probability is really low. In order to take place, owner add a new depository in emergency mode since emergency mode skips the chainId check

Recommendation:
To mitigate replayability, include additional contextual parameters in the signed message:

- chainId: Ensures the signature is valid only on the intended blockchain network.
- contractAddress: Binds the signature to a specific contract deployment.

By including these parameters in the message to be signed, signatures will only be valid for the expected contract on the correct chain, preventing cross-chain and cross-contract replay attacks.

Client's comment:
The chainId is already included in the context through the depositaryId. The depositaryId will always differ across different chains, which prevents cross-chain replayability. The contract address is intentionally not included in the signature, since it may be updated.

Zero-sum invariant might not hold, leading to users not being able to withdraw

Description

The protocol has a zero-sum invariant that the total asset balance should be equal to the sum of all users' asset balances. This holds only if deposit() and withdrawal() methods are used, but the protocol has the following methods that allow special roles to make the contract not hold true to this invariant.

- transferOut() ⇒ allows the treasury manager to withdraw any amount of assets, leading to the total asset balance less than the sum of all user asset balances
- creditTreasury() ⇒ allows the treasury supplier to add any amount of assets, leading to the total asset balance more than the sum of all user asset balances

Impact

Since the treasury manager can withdraw any amount, this will make the asset balance less than the sum of all user asset balances, meaning there won't be enough funds in the depository for all users to withdraw.

Recommendation:

It is advised to use these methods with precaution so that, for the most part, the zero-sum invariant holds and users can withdraw.

Client's comment:

Client mentions that it is the responsibility of the treasury manager to ensure that the real balances are always at least as large as the virtual ones. At a minimum, the system charges fees from users and accumulates a buffer balance that users can no longer access directly.

An asset's assetBalance can be greater than the assetBalanceLimit

Description
In the contract Router.sol, the method addAsset() allows the owner to add a new asset to the depository along with the maxDeposit amount. This amount is the maximum an asset can be deposited in the repository. The method deposit() used by the users checks this requirement and returns in case the total deposited amount is greater than the assetBalanceLimit.

But the method creditTreasury can be called by the treasury supplier account to deposit any amount of assets in the depository, even greater than the assetBalanceLimit.

Impact
Although there is no security issue, this might prevent users from depositing, as the assetBalanceLimit applies to them, which creditTreasury can also use without restriction.

Recommendation:
Add a check to ensure the creditTreasury can not be used to deposit more than the assetBalanceLimit.

Client's comment:
Client mentions that the assetBalanceLimit is implemented as a safeguard specifically for regular users. Off-chain components are not dependent on this limit in any way. The credit treasury manager is authorised to intentionally exceed the limit when necessary.

Signature replay is possible when the depository's chainID is not checked

Description

In the contract ContractProvider.sol, the method emergencyAddDepository() allows an admin to add a new depository. Since every depository has a key i.e < name, version, chainId>, and depositoryId is derived as a hash of this key, it inherently prevents signature replay cross-chain, as every chain has a different chainId, leading to no two depository IDs with the same name and version to be the same.

The withdraw() methods use this depositoryId as one of the parameters for the signed message to verify the owner of the signature, so this prevents cross-chain signature replay.

But in emergency mode, the admin can add a new depository without checking the chainId for the key.

Impact

- Admin adds a depository whose key has a chainId equal to some other chain
- The other chain already has a depository with the same depositoryId (possible if the depository has the same name and version)
- Replay of signature cross-chain since depositoryId will be the same, and the current signature logic is not using EIP-712

Recommendation:

Add a check to ensure the key.chainId == block.chainid even in emergency mode.

Client's comment:

The client focuses on ensuring that the administrator is able to restore and repair the system in case of unforeseen situations. They use a multisig account for the owner role and carefully review every call.

Wrong Reliance on block.chainid in validateDepositary Function

Description
The validateDepositary function in ContractProvider.sol currently uses block.chainid to validate the chain ID of the depositary:

```
if (key.chainId != block.chainid) {
    revert WrongDepositaryChainId();
}
```

This approach creates a critical dependency on the blockchain's chainid. If the deployed chain experiences a fork (e.g., due to reorgs or contentious hard forks), the block.chainid may change, causing the function to always revert and rendering the protocol unusable

Impact
In the event of a fork, all calls to validateDepositary will fail.
This will lock out deposits and potentially halt protocol functionality entirely.
The protocol becomes fragile and non-resilient to network-level changes outside the developers' control.
The probabilities are low this is why the severity of the issue has been set to low.

Recommendation:
Instead of directly relying on block.chainid, define the chain ID as a configurable state variable within the contract. Add a setter function (restricted to an admin role or governance) to update the chain ID if required. This allows controlled updates to the expected chain ID without introducing systemic failure when forks occur.

Client's comment:
Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/77c7464a804a7c00b286b2c198ca7f69cb3d4a69

Check nonZero address is missing

Description
The constructor within the `DepositStorage.sol` smart contract receives and set the address for the wrappedNativeAsset used by depositors. However, it is not ensured that it is not address(0).

Impact
It would lead to unexpected behaviors.

Recommendation:
Implement a check to ensure that the contract is not set to address(0)

Client's comment:
Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/d7a74289dfb26b61ce5ed2e5e5373de5d8705a13

Using not locked solidity versions can lead to unexpected behaviors

Description
Some contracts in the codebase are using unlocked Solidity pragma statements such as pragma solidity ^0.8.30;. This allows the compiler to use any version greater than or equal to 0.8.30 and less than 0.9.0. While this provides flexibility, it can introduce risks if newer compiler versions contain undiscovered bugs, behavior changes, or incompatibilities.

Impact
Future Solidity compiler versions could introduce breaking changes, security vulnerabilities, or unintended behavior.

Builds may become non-deterministic if developers use different compiler versions.

Auditability and reproducibility of the contracts are reduced.

Recommendation:
Lock the Solidity compiler to a specific, stable version to ensure deterministic builds, avoid unexpected behavior, and improve long-term maintainability and auditability of the contracts. Check every contract which is not using a locked solidity version.

Client's comment:
Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/a3d61d348c27abb44d70bdba52fc741075a9748c

Zero-Address Receiver Can Permanently Lock Funds (Deposit)

Description

In `Router.sol, deposit()` checks the receiver via `Depositary.isDepositaryUser()`. When the whitelist is disabled, this check always returns true, even for `address(0)`. A deposit to `0x0` credits balances to the zero address in `DepositaryStorage`, but since `0x0` cannot sign a withdrawal, funds are locked permanently.

Scenario:

Whitelist disabled → user deposits with `receiver = 0x0` → funds credited to `0x0` → impossible to withdraw later.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L146

Recommendation:

Add `require(receiver != address(0))` in `Router.deposit`. Optionally restrict deposits to known safe addresses or always enforce whitelist for receivers.

Client's comment:

Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/d7a74289dfb26b61ce5ed2e5e5373de5d8705a13

Native-Token Withdraw to Zero Address Burns ETH

Description
In `RouterLogic.sol`, withdraw with `unwrapNativeToken = true` does not check for `receiver != address(0)`. ETH can be sent to the zero address, effectively burning user funds.

Scenario:
Signed withdraw package with `receiver = 0x0` and `unwrapNativeToken = true` → ETH transferred to `0×0`.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L158

Recommendation:
Add `require(receiver != address(0))` in withdraw when unwrapping native tokens.

Client's comment:
Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/d7a74289dfb26b61ce5ed2e5e5373de5d8705a13

Emergency Drains Blocked While Paused

Description

In `Router.sol`, `emergencyERC20TokenTransfer` and `emergencyNativeTokenTransfer` are guarded by `whenNotPaused`. This blocks emergency drains when the Router is paused—precisely when they may be needed.

Scenario:

Admin pauses Router during incident → tries emergency drain → reverts due to pause guard.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L171

Recommendation:

Allow emergency drains during pause (remove `whenNotPaused` or add an "emergency-only" guard). Add `require(to != address(0))` to prevent accidental burns.

Client's comment:

Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/38627ef1a4aa69f21f3af95ef29ba4226a77743f

Depositary Removal Can Break Operations

Description
In `ContractProvider.sol, removeDepositary` deletes a depositary mapping without safeguards. If Router still references it, lookups return `0x0`, causing `_getDepositary()` in Router to fail and DoS deposits/withdrawals.

Scenario:
Admin removes an active depositary → Router calls revert due to zero address.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L204

Recommendation:
Restrict removal to emergencies or when balances are zero. Add explicit error in Router when `_getDepositary()` resolves to zero. Emit `DepositaryRemoved` event for monitoring.

Auditor's comments:
The described scenario is still possible but only when the emergency mode has been set. They will have to plan it before executing the action so it does not DoS.

Client's comment:
Fixed: https://github.com/RhoLabs/rho-collateral-contracts/commit/c379a66441cc8286d4f19801133397680b99b052

Asset Removal “Freeze” (Funds Lock)

Description

In `Depositary.sol/DepositaryStorage.sol, removeAsset` blocks deposits (and possibly withdrawals), freezing balances until re-whitelisted. Designed for emergencies, but risky operationally.

Scenario:

Admin removes asset → deposits revert, balances remain frozen.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L265

Recommendation:

We will follow the recommendation and make sure to notify users in advance before removing any assets.

Client’s comment:

This appears to be a duplicate of issue #7

Internal Transfer from == to Is a No-Op (Gas Waste)

Description

In `Depositary.sol, transferInternally(from, to, …)` permits `from == to`. This wastes gas and emits events with no effect.

Scenario:

Operator calls with `from == to` → balances unchanged, gas wasted.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L284

Recommendation:

Add `require(from != to)` if desired to avoid no-ops.

Client's comment:

This function is removed.
This appears to be a duplicate of issue #31

Deposit Receiver Hygiene (Optional Safety)

Description
Deposits can be sent to system addresses like the Router itself. This can confuse users or create loops.

Scenario:
User deposits to Router address → credited internally, not meaningful.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L290

Recommendation:
Add hygiene check `require(receiver != address(this))`. Optionally restrict deposits to EOAs or whitelisted contracts.

Client's comment:
We consider the receiver field to be fully under the user's control and responsibility. The system executes exactly what the user signs and submits.

Compromised Withdrawals Operator + Phished Owner Signature Can Drain Pool (No Per-User Solvency)

Description

`withdraw` verifies a valid owner signature, a fresh `withdrawalId`, and global pool coverage (`assetBalance ≥ amount`) but does not check the owner's personal balance. If an attacker controls a withdrawals-operator address and tricks a victim (the "owner") into signing a withdrawal message (e.g., via `eth_sign`/malicious UI), the system will pay out to any `receiver` even when the owner never deposited. The owner's balance becomes negative and other users' deposits implicitly fund the payout.

Scenario:

- Seed: Users deposit; pool `assetBalance = 100`.
- Compromise: Attacker has a withdrawals-operator role and obtains a phished signature from a victim address that has `0` balance.
- Action: Operator calls `withdraw(owner=victim, receiver=attacker, amount=70, …)` with a fresh `withdrawalId` and valid deadline.
- Result: Pool decreases by `70`; attacker receives `70`; victim's on-chain balance becomes –`70`. Tx succeeds because per-user solvency isn't enforced.

https://gist.github.com/adeshkolte/92b80ae75db85026b68bd5fe81693b9f

Requirements (for the attack to work):

- Caller holds WithdrawalsOperator for the depositary.
- Valid owner signature over `(depositaryId, withdrawalId, receiver, amount, asset, unwrap, metadata, deadline)`.
- Global coverage: `assetBalance ≥ amount`.
- Fresh id: `withdrawalId` strictly greater than the owner's last id.

Recommendation:

Enforce per-user solvency in `withdraw: require(ownerBalance ≥ amount);` revert otherwise.
Use Multi-sig for WithdrawOperator

Client's comment:
The WithdrawalsOperator is an internal, trusted system component, not an externally accessible role, so the risk is contained within the operational trust boundary. In addition, we are in the process of migrating the WithdrawalsOperator to a multi-sig setup, which will further reduce the likelihood of compromise.

Treasury Credits Ignore the Asset Cap

Description
User deposits respect a per-asset cap (`assetBalanceLimit`). But `creditTreasury` doesn't check that cap—it just adds to the pool. A treasury operator can push the pool over the limit. After that, every normal user deposit will fail because the "room left" calculation becomes zero (or negative).

Proof of Concept:
https://gist.github.com/adeshkolte/f9f7fbd451db89f45cc0975a159217c9

Impact:
- **On-chain:** Deposits for that asset DoS until the balance drops or the limit changes.
- **Off-chain:** Dashboards/alerts that assume "balance ≤ limit" will look wrong; headroom reads as zero/negative even though treasury could still add more.

Recommendation:
Add the same limit check in `creditTreasury` (or clamp the credited amount to the remaining capacity) and revert with a clear error if exceeded.
Document it clearly, keep **TreasurySupplier** tightly controlled, add monitoring/alerts for "over cap" state, and expose an explicit **overage** metric so ops and dashboards don't misread the numbers.

Client's comment:
The per-asset cap is a policy control intended only for external user deposits, not for privileged internal roles. Treasury credits are an operational tool and are allowed to exceed the user cap by design.

INFORMATIONAL-1 | ACKNOWLEDGED

Centralized behaviour allows for direct access to user funds via admin wallets

Description:
The main actions of the protocol are centralized under the control of operators/ administrators, who are able to execute almost any action within the system. While this setup may be intentional for operational efficiency or development agility, it introduces a central point of failure.

If the administrators' wallets, private keys, or backend infrastructure are compromised, an attacker could gain unrestricted control over the protocol. This could lead to malicious activities such as asset theft, configuration manipulation, or denial of services like user's withdrawals. Even without compromise, the high degree of centralization reduces trustlessness and makes the protocol dependent on the security and integrity of a small set of operators.

INFORMATIONAL-2 | RESOLVED

Method transferInternally() can have `to` and `from` as the same address

Description:
In the contract Router.sol, the method transferInternally() allows the internal transfer operator to update user asset balances for users without actually transferring funds b/w them. This method has 2 parameters: `to` and `from` addresses. But these addresses can be the same, as there is no check ensuring that these addresses are not the same. Although there is no security issue, it is advised to add a check.

Recommendation:
Add a check to ensure `to` and `from` addresses are not the same.

Client's comment:
The transferInternally function has been removed.

Provider Return Values Not Validated

Description:
In `Router.sol`, `_getDepositary()` trusts provider to return valid addresses. If provider returns `address(0)` or a non-compliant contract, calls revert deeper in the stack with unclear errors.

Scenario:
Provider misconfigured to return `address(0)` → Router calls revert in low-level execution.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L196

Recommendation:
Validate provider-returned addresses (non-zero, optionally ERC165) at init and before use. Add explicit custom errors.

Client's comment:
Contract Provider component is an internal part of the system, deployed and controlled within the same trust boundary as the Router and Depositary contracts. Misconfiguration of the provider would indicate a broader deployment error rather than a protocol-level vulnerability. Since the provider cannot be influenced by external actors, we consider additional runtime validation unnecessary. Such checks would only increase gas consumption without improving the security model.

Withdraw Parameter Order Mismatch (Interface vs Implementation)

Description:
`IRouter.sol` defines `withdraw(receiver, owner, …)` while `Router.sol` implements `withdraw(owner, receiver, …)`. This ABI mismatch can break integrators and invalidate signatures.

Scenario:
Integrator encodes per `IRouter` (receiver first) → Router interprets args incorrectly → signature fails.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L212

Recommendation:
Align parameter order (prefer `owner, receiver`), regenerate ABI, and add regression tests.

Deadline Type Inconsistency

Description:
In `Router.sol, deposit` uses `uint` for deadline, while `withdraw` and others use `uint64`. This inconsistency complicates integrators and tooling.

Scenario:
Client code must handle deadlines differently per function.

Recommendation:
Standardize on `uint64` across all functions.

Signature Domain Hardening (Defense-in-Depth)

Description:
In `RouterLogic.sol`, signed payloads omit the Router address and do not follow EIP-712. While `Depositary.sol` enforces `onlyViaRouter`, missing domain separation allows signature replay if depositary is reassigned to a new Router.

Scenario:
Signature valid on Router1 also valid on Router2 if provider reassigns depositary.

Recommendation:
Include Router address in signed payload or migrate to EIP-712 with domain separation.

Client's comment:
The primary context is carried by depositaryId, which is included in the signed payload and uniquely separates domains across deployments. The ability to work with different Router contracts is intentional, as we do not bind signatures to a specific Router address in order to support controlled Router upgrades without requiring users to re-sign.

Negative Sender Balance on Internal Transfers (By Design)

Description:
In `Depositary.sol, transferInternally` allows negative balances to represent P&L. This is intended, not a bug, but downstream systems must handle negatives safely.

Scenario:
User transfers more than balance → sender balance goes negative, receiver credited.

https://gist.github.com/adeshkolte/4fed71b5143fd4dba470f6faeabb3c51#file-poc-t-sol-L273

Recommendation:
Keep if required. Document clearly, add alerts for large negative balances, and ensure downstream systems handle correctly.

	./configuration/ContractProvider.sol **./configuration/IContractProviderErrors.sol** **./configuration/MainContracts.sol** **./configuration/Configurator.sol** **./configuration/IContractProvider.sol** **./configuration/IConfigurator.sol** **./libraries/IterableIdAddressSet.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	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)	Pass

	./libraries/ContractIds.sol **./libraries/IIterableIdAddressSetErrors.sol** **./utils/ITimeProviderErrors.sol** **./utils/IWrappedNativeToken.sol** **./utils/IAddressValidatorErrors.sol** **./utils/IInterfaceValidationErrors.sol** **./utils/IPausable.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	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)	Pass

	./utils/AddressValidator.sol **./utils/TimeProvider.sol** **./depositary/IDepositary.sol** **./depositary/libraries/DepositaryLogic.sol** **./depositary/storage/DepositaryStorage.sol** **./depositary/storage/IDepositaryStorage.sol** **./depositary/storage/DepositaryStorageParamKeys.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	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)	Pass

	./depositary/storage/IDepositaryStorageErrors.sol **./depositary/Depositary.sol** **./depositary/IDepositaryErrors.sol** **./access-control/Roles.sol** **./access-control/DepositaryRelatedRoles.sol** **./access-control/IAccessControlManagerErrors.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	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)	Pass

	./access-control/AccessControlManager.sol **./access-control/IAccessControlManager.sol** **./router/IRouterErrors.sol** **./router/libraries/RouterLogic.sol** **./router/IRouter.sol** **./router/Router.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	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)	Pass

We are grateful for the opportunity to work with the Rho Labs team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Rho Labs team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo