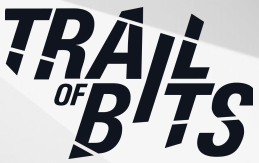

Rio Network Liquid Restaking Token

Security Assessment

April 19, 2024

Prepared for:
Solimander
Rio Network

Prepared by: **Bo Henderson, Elvis Skoždopolj, and Alexis Challande**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Alexis Challande, Consultant
alexis.challande@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 25, 2024	Pre-project kickoff call
February 5, 2024	Status update meeting #1
February 13, 2024	Delivery of report draft
February 15, 2024	Report readout meeting
April 19, 2024	Delivery of comprehensive report with fix review appendix

Executive Summary

Engagement Overview

Rio Network engaged Trail of Bits to review the security of its liquid restaking tokens (LRTs). LRTs are built on top of the EigenLayer protocol, which allows the tokens that are staked to secure one protocol to be restaked to secure another protocol. Rio Network LRTs represent shares in an aggregated pool of funds that are restaked via Eigenlayer.

A team of two consultants conducted the review from January 29 to February 9, 2024, for a total of four engineer-weeks of effort. Our testing efforts focused on the deposit, withdrawal, rebalance, and administrative functions that support the LRT pools. With full access to source code and documentation, we performed static and dynamic testing of the protocol contracts in the `rio-monorepo` repository, using automated and manual processes. We did not consider any other packages in this repository during this review.

Observations and Impact

Our review of Rio Network's LRTs did not uncover any critical issues. Although some issues are of high severity, such as the lack of validation for the sacrificial deposit amount (TOB-RIO-4), these issues require the administrator to make mistakes or require specific market conditions to occur, so they would be difficult for an attacker to exploit.

However, the system needs to be tested more thoroughly prior to deployment. Some issues, such as the heap invariants not holding upon node removal (TOB-RIO-3), indicate that the restaking system's test suite still suffers from significant gaps. Furthermore, mutation testing revealed numerous input validation checks and access controls that lack testing, a sample of which can be found in appendix D.

Apart from the test coverage, the implementation of the restaking portion of this project appears to be relatively mature. NatSpec comments are thorough, and high-level documentation is available publicly. However, the governance portion was not in scope during this review, and its implementation and integration with the restaking pool contracts would benefit from a security assessment.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Rio Network take the following steps prior to mainnet deployment:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Expand the test suite.** Some significant gaps in test coverage were identified by mutation tests. Write additional tests that assert the expected behavior of every branch of the business logic, and then use code coverage assessments to confirm that gaps in test coverage have been closed.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Low	1
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Timing	1

Project Goals

The engagement was scoped to provide a security assessment of the liquid restaking token application. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there any error-prone or incorrect steps in the issuance of a new LRT?
- Can any exchange rates be manipulated by an attacker?
- Is all user-provided input properly validated?
- Are any access controls missing or do any roles have unsuitable permissions?
- Does the logic underlying low-level utilities, such as the heap, conform to specifications?
- Is the system vulnerable to share inflation attacks?
- Can the system functions be front run, back run, or sandwich attacked to extract value?
- Are there any rounding issues that would allow a user to extract value from the system?

Project Targets

The engagement involved a review and testing of the following target.

rio-monorepo

Repository	https://github.com/rio-org/rio-monorepo
Version	`56814257613dec52cd4295b0f1f94c7afe162048`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Coordinator**. This component is the primary entry point for all end-user interactions. We checked for missing input validation and timing issues among deposit, withdrawal, and rebalance operations.
- **Deposit pool**. Funds are sent to this component when users deposit assets. During rebalancing, funds are pulled from this pool to fulfill withdrawal requests, and the remaining balance is deposited into EigenLayer. We traced the flow of funds through this contract to check whether funds can become stuck or be diverted by an attacker.
- **Withdrawal queue**. EigenLayer strategies feature a seven-day delay during fund withdrawal, and the withdrawal queue component of RioNetwork manages the queueing and claim processes at either end of this delay. We reviewed the transition between epochs and the role of this component in the settlement process to verify whether funds are handled safely.
- **Asset registry**. This component allows administrators to add or remove assets from the staking pool associated with an LRT. We reviewed the conversion functions that manage the exchange rate between assets, shares, and the unit of account to check whether they can be manipulated in favor of an attacker by donating tokens.
- **Issuer**. This component deploys new LRT contracts and all associated contracts supporting the restaking pool. We reviewed the contract deployment and initialization processes to verify whether proper guardrails are in place.
- **Operator delegator**. This is the adapter layer that sits between EigenLayer and Rio Network, and all interactions with EigenLayer occur through this component. We reviewed the EigenLayer documentation and selected parts of the EigenLayer implementation to check whether all actions taken by the operator delegator interact with EigenLayer according to best practices and specifications.
- **Operator registry**. This component allocates and deallocates funds among a set of operators. We checked that there are no error-prone steps in the registration, activation, and deactivation of operators. We reviewed the allocation and deallocation methods to verify whether internal accounting is consistent while distributing funds evenly among the available, active operators.

- **Ancillary contracts**. The `RioLRT` and `RioLRTRewardsDistributor` contracts are straightforward and provide an ERC-20 implementation and reward division logic respectively. We reviewed their access controls and interactions with the other system components.

  The `OperatorUtilizationHeap` and `OperatorOperations` libraries, however, implement more complicated algorithms and critical business logic. We wrote additional fuzz tests for the `OperatorUtilizationHeap` library to verify whether its implementation matches the expected specifications and checked whether the fund movement methods in `OperatorOperations` are transferring assets safely.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- All rio-`monorepo` packages other than the `protocol` package were explicitly out of scope.
- The governance contracts in the `protocol` package were explicitly out of scope.
- Rio Network depends on the security of Eigenlayer. Although we reviewed the interactions between these two systems and investigated Eigenlayer to the extent necessary to understand the LRT system, Eigenlayer code was not included in the scope of this review.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Appendix E
`slither-mutate`	A deterministic mutation generator that detects gaps in test coverage	Appendix D
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix E
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix E

Areas of Focus

Our automated testing and verification work focused on verifying the properties of the `OperatorUtilizationHeap` contract and assessing the test suite coverage.

Test Results

The results of this focused testing are detailed below.

`OperatorUtilizationHeap`. The `OperatorUtilizationHeap` contract implements a min-max heap to prioritize operators for allocation and deallocation, depending on their utilization.

Property	Tool	Result
The root node of the heap is the smallest value in the heap.	Echidna	Passed
The children nodes of the root are the largest values in the heap.	Echidna	TOB-RIO-3
A node on the minimum level of the heap is smaller than or equal to its child nodes.	Echidna	TOB-RIO-3
A node on the maximum level of the heap is larger than or equal to its child nodes.	Echidna	TOB-RIO-3

`slither-mutate`: The table below displays the portion of each type of mutant for which all unit tests passed. The presence of valid mutants indicates that there are gaps in test coverage because the test suite did not catch the introduced change.

- Uncaught revert mutants replace a given expression with a `revert` statement and indicate that the line is not executed during testing.
- Uncaught comment mutants comment out a given expression and indicate that the effects of this line are not checked by any assertions.
- Uncaught tweak mutants indicate that the expression being executed features edge cases that are not covered by the test suite.

The `rio-monorepo/packages/protocol/contracts` subdirectory is the root for all target paths listed below. Targets that are out of scope (e.g., governance contracts) or that produced zero analyzed mutants (e.g., interfaces) were omitted from mutation testing analysis.

Target	Uncaught Reverts	Uncaught Comments	Uncaught Tweaks
`restaking/base/RioLRTCore.sol`	8%	6%	14%
`restaking/RioLRT.sol`	23%	61%	37%
`restaking/RioLRTAVSRegistry.sol`	88%	100%	100%

restaking/RioLRTAssetRegistry.sol	15%	33%	N/A
restaking/RioLRTCoordinator.sol	4%	20%	23%
restaking/RioLRTDepositPool.sol	0%	20%	24%
restaking/RioLRTIssuer.sol	0%	20%	12%
restaking/RioLRTOperatorDelegator.sol	9%	44%	42%
restaking/RioLRTOperatorRegistry.sol	24%	40%	37%
restaking/RioLRTRewardDistributor.sol	0%	30%	17%
restaking/RioLRTWithdrawalQueue.sol	7%	38%	30%
utils/Array.sol	0%	0%	50%
utils/Asset.sol	0%	20%	10%
utils/LRTAddressCalculator.sol	9%	0%	48%
utils/Memory.sol	50%	45%	N/A
utils/OperatorOperations.sol	0%	8%	27%
utils/OperatorRegistryV1Admin.sol	32%	28%	45%
utils/OperatorUtilizationHeap.sol	1%	0%	16%
utils/ValidatorDetails.sol	50%	74%	N/A

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	A modern version of Solidity is used to prevent overflows in arithmetic, and although unchecked blocks would benefit from more descriptive inline comments, they are low risk. However, we identified a risk of overflow while casting between signed and unsigned integers (TOB-RIO-1). Precision-losing operations explicitly round in favor of the protocol.	**Satisfactory**
Auditing	The system consistently emits events that include enough data for effective off-chain monitoring. Inline comments specify when each event is expected to be emitted; however, more documentation describing the implications of each event would be helpful. Details regarding the off-chain monitoring services were not apparent, and no incident response plan is provided in public documentation.	**Moderate**
Authentication / Access Controls	Privileges are divided among granular roles, which each follow the principle of least privilege. We did not identify any missing or misconfigured access controls, but depending on how the governance account is structured, a two-step process for ownership transfer may help prevent irrevocable mistakes (TOB-RIO-2).	**Satisfactory**
Complexity Management	Functions have well-defined responsibilities and do not overuse nested operations or feature high cyclomatic complexity. The naming convention is clear and consistent across variable and function names. We did not identify any duplicate code.	**Satisfactory**

Decentralization	All system logic, including the management of funds, can be upgraded by a single account without giving users a chance to opt-out. Although the governance mechanism was not in scope, an upgrade process managed by the Rio Network decentralized autonomous organization (DAO) with a timelock may contribute to improved decentralization compared to the owner role being held by an externally owned account (EOA) or multisignature account. Risks related to privileged roles and the upgrade process should be thoroughly documented once the governance component is integrated.	**Moderate**
Documentation	Code is thoroughly commented and all methods feature detailed NatSpec comments. Public, high-level documentation and usage instructions are available.	**Satisfactory**
Low-Level Manipulation	Assembly is used extensively in the `ValidatorDetails` contract and sparingly in the `Memory` and `RioLRTOperatorRegistry` contracts. However, we did not identify any issues in these components.	**Satisfactory**
Testing and Verification	Tests are thorough, featuring both unit and fuzz testing. However, mutation testing uncovered some gaps in test coverage, including a lack of tests for input validation and access controls. Some issues could have been identified earlier by more thorough tests (TOB-RIO-3).	**Moderate**
Transaction Ordering	The system is sensitive to timing issues, especially surrounding the withdrawal queue and rebalancing of funds. We identified a single low-severity issue related to timing attacks (TOB-RIO-6). The system would benefit from further investigation and more rigorous specification of timing risks.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Unsafe casting of signed to unsigned integers	Data Validation	Medium
2	Lack of two-step process for ownership transfers	Data Validation	Informational
3	Heap node removal leads to incorrect allocation and deallocation of shares	Data Validation	High
4	Missing validation for sacrificial deposit amount	Data Validation	High
5	Missing upper bound on rebalanceDelay system parameter	Data Validation	Informational
6	Reward distribution could be susceptible to sandwich attacks	Timing	Low

Detailed Findings

1. Unsafe casting of signed to unsigned integers

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RIO-1
Target: `packages/protocol/contracts/utils/OperatorRegistryV1Admin.sol`, `packages/protocol/contracts/restaking/RioLRTOperatorRegistry.sol`, `packages/protocol/contracts/restaking/RioLRTOperatorDelegator.sol`	

Description

Some values that could be negative are cast to `uint256` values, putting the system at risk of underflows. Other values that are `uint256` values are unsafely cast to `uint128` values, introducing a risk of overflows.

The `getEigenPodShares` method is called by the `getETHUnderManagement` method in the `RioLRTOperatorDelegator` contract and the `queueOperatorStrategyExit` method in the `OperatorRegistryV1Admin` contract. The `getEigenPodShares` method could return negative values if the `podOwnerShares` variable is less than zero (documentation).

In such cases, the casts to `uint256` values shown in figures 1.1 and 1.2 could underflow and cause the `getETHUnderManagement` method, for example, to return the wrong value.

```
uint256 sharesToExit;
if (strategy == BEACON_CHAIN_STRATEGY) {
    // It is not possible to exit ETH with precision less than 1 Gwei.
    sharesToExit = reducePrecisionToGwei(uint256(delegator.getEigenPodShares()));
} else {
    sharesToExit = operator.shareDetails[strategy].allocation;
}
```

Figure 1.1: Unsafe cast in the `queueOperatorStrategyExit` method (`OperatorRegistryV1Admin.sol#L192-L198`)

```
function getETHUnderManagement() external view returns (uint256) {
    return uint256(getEigenPodShares()) + address(eigenPod).balance;
}
```

Figure 1.2: Unsafe cast in the `getETHUnderManagement` method (`RioLRTOperatorDelegator.sol#L112-L114`)

Another instance of unchecked casting happens in the `RioLRTOperatorRegistry` contract's `allocateStrategyShares` method, where a `uint256` value is cast to a `uint128` value (integer truncation).

```
allocations = new OperatorStrategyAllocation[](s.activeOperatorCount);
while (remainingShares > 0) {
    OperatorDetails storage operator = s.operatorDetails[heap.getMin().id];
    OperatorShareDetails memory operatorShares = operator.shareDetails[strategy];

    // If the allocation of the operator with the lowest utilization rate is maxed
out,
    // then exit early. We will not be able to allocate to any other operators.
    if (operatorShares.allocation >= operatorShares.cap) break;

    uint256 newShareAllocation = FixedPointMathLib.min(operatorShares.cap -
operatorShares.allocation, remainingShares);
    uint256 newTokenAllocation =
IStrategy(strategy).sharesToUnderlyingView(newShareAllocation);
    allocations[allocationIndex] = OperatorStrategyAllocation(
        operator.delegator,
        newShareAllocation,
        newTokenAllocation
    );
    remainingShares -= newShareAllocation;

    uint128 updatedAllocation = operatorShares.allocation +
uint128(newShareAllocation);

    operator.shareDetails[strategy].allocation = updatedAllocation;
    heap.updateUtilization(OperatorUtilizationHeap.ROOT_INDEX,
updatedAllocation.divWad(operatorShares.cap));

    unchecked {
        ++allocationIndex;
    }
}
```

Figure 1.3: Unsafe casting in the `allocateStrategyShares` and `deallocateStrategyShares` methods (`RioLRTOperatorRegistry.sol#L371` and `RioLRTOperatorRegistry.sol#L511`)

For this vulnerability to be exploited, the token supply must be at least 2**191. This number can change based on the value of the `MAX_RESTAKED_BALANCE_GWEI_PER_VALIDATOR` constant defined in the `EigenPod` contract (currently 32,000,000,000 gwei).

In addition, exploitation would require negative shares to be returned from the `getEigenPodShares` function, an empty balance in the deposit pool, and more than 32 ether in the operator shares.

Exploit Scenario
The above conditions are met. An attacker asks for a withdrawal of 1 token and obtains all the shares in the operator.

Due to time constraints, it was not possible to develop a fully working exploit, and we may have missed a constraint that would render the exploitation impossible.

Recommendations
Short term, use the `SafeCast` library to verify underflows in casting operations.

Long term, use a Semgrep rule to detect casting, and verify the results to help prevent these issues (appendix E).

2. Lack of two-step process for ownership transfers	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RIO-2
Target: `protocol/contracts/*`	

Description
When called, the `transferOwnership` function immediately sets the contract owner to the provided address. The use of a single step to make such a critical change is error-prone; if the function is called with erroneous input, the results could be irrevocable or difficult to recover from.

The issue is prevalent in the following contracts:

- `RioLRT`
- `RioLRTAssetRegistry`
- `RioLRTAVSRegistry`
- `RioLRTCoordinator`
- `RioLRTDepositPool`
- `RioLRTIssuer`
- `RioLRTOperatorRegistry`
- `RioLRTRewardDistributor`
- `RioLRTWithdrawalQueue`

Although the `owner` role is expected to be held by a DAO contract, transfer of ownership is a high-risk operation that will most likely rarely happen. The requirement for a separate acceptance step before role transfer is completed would guarantee that the address of the new owner contract is correct and that it can properly execute transactions on the contracts it will be administering.

Exploit Scenario

Alice invokes `transferOwnership` to change the contract owner but accidentally enters the wrong address. She permanently loses the ability to set important system parameters and upgrade the contract.

Recommendations

Short term, implement a two-step process for all irrecoverable critical operations. Consider using the `Ownable2StepUpgradeable` OpenZeppelin dependency instead of `OwnableUpgradeable` for ownership management.

Long term, identify and document all possible actions that can be taken by privileged accounts, along with their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

3. Heap node removal leads to incorrect allocation and deallocation of shares

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-RIO-3
Target: `protocol/contracts/utils/OperatorUtilizationHeap.sol`	

Description

Removing an operator from the `RioLRTOperatorRegistry` contract will result in the incorrect prioritization of operators when allocating and deallocating shares.

The protocol uses the `OperatorUtilizationHeap` library to manage the operator priority queue when allocating and deallocating shares. This library defines a min-max heap that contains minimum values on even levels of the heap (starting from the root as level 0), and maximum values on odd levels of the heap. The heap must adhere to the following properties:

1. The root node of the heap is the smallest value in the heap.
2. The children of the root node are the largest values in the heap.
3. A node on a minimum level is smaller than or equal to its child nodes.
4. A node on a maximum level is larger than or equal to its child nodes.

However, the properties of the heap are not maintained due to the `remove` function failing to check and modify the node positions above the removed node:

```
function remove(Data memory self, uint8 index) internal pure {
    if (index < ROOT_INDEX || index > self.count) revert INVALID_INDEX();

    self._remove(index);
    // missing self._bubbleUp(index);
    self._bubbleDown(index);
}
```

Figure 3.1: The `remove` function (OperatorUtilizationHeap.sol#L94–L99)

As a result, the allocation and deallocation of shares to operators will be incorrect, potentially leading to a denial of service.

Exploit Scenario

Alice removes an operator from the heap. This replaces the contents at the operator's index in the heap with the contents of the last index of the heap, which has a utilization of zero. Due to the `remove` function not preserving the heap properties, the `getMax` function will return the wrong value, preventing the deallocation of shares because the system assumes that all other operators must have a smaller utilization. As a result, users are unable to withdraw their shares.

Recommendations

Short term, add the line `self._bubbleUp(index)` to the `remove` function of the `OperatorUtilizationHeap` contract.

Long term, define system- and function-level invariants and use a stateful smart contract fuzzer such as Echidna, Medusa, or Foundry's invariant mode to test the invariants and ensure that they hold under various conditions.

4. Missing validation for sacrificial deposit amount

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RIO-4
Target: `protocol/contracts/restaking/RioLRTIssuer.sol`	

Description

The `RioLRTIssuer` contract uses a sacrificial deposit to prevent inflation attacks on the vault. However, no check is performed on the amount deposited.

Because the protection induced by the sacrificial deposit is proportional to its value, it could be beneficial to enforce a minimum amount of tokens.

```
function _deposit(IRioLRTCoordinator coordinator, address asset, uint256 amount)
internal {
    if (asset == ETH_ADDRESS) {
        if (amount != msg.value) revert INVALID_ETH_PROVIDED();
        coordinator.depositETH{value: amount}();
        return;
    }

    IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
    IERC20(asset).approve(address(coordinator), amount);

    coordinator.deposit(asset, amount);
}
```

Figure 4.1: The `_deposit` function (RioLRTIssuer#163–174)

Exploit Scenario

An administrator wrongly sets the sacrificial deposit to `0` or a small value. An attacker then performs an inflation attack on the vault when they see a user deposit.

Recommendations

Short term, enforce a minimum default value for the sacrificial deposit. For instance, Uniswap hard codes a 1,000 wei value on its contracts.

Long term, add documentation to explain the protection induced by the sacrificial deposit, and add reminders to periodically check whether that protection is sufficient.

5. Missing upper bound on rebalanceDelay system parameter

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RIO-5
Target: `protocol/contracts/restaking/RioLRTCoordinator.sol`	

Description

The `setRebalanceDelay` function allows the contract owner to specify the `rebalanceDelay` system parameter that is used to define how often the `rebalance` function of the `RioLRTCoordinator` contract can be called. This function is used to finalize deposits and withdrawals from the system.

However, there is no upper bound on the state variable, so `rebalanceDelay` could be set to any value. An excessive `rebalanceDelay` value (e.g., resulting from a typo) would prevent users from finalizing their deposits and withdrawals and may not be noticed until it caused disruptions.

```
function setRebalanceDelay(uint24 newRebalanceDelay) external onlyOwner {
    _setRebalanceDelay(newRebalanceDelay);
}

function _setRebalanceDelay(uint24 newRebalanceDelay) internal {
    rebalanceDelay = newRebalanceDelay;

    emit RebalanceDelaySet(newRebalanceDelay);
}
```

Figure 5.1: The `setRebalanceDelay` and `_setRebalanceDelay` functions (`RioLRTCoordinator.sol#L144–L146`)

Recommendations

Short term, set an upper bound for the `rebalanceDelay` system parameter. The upper bound should be high enough to encompass any reasonable value but low enough to catch mistakenly or maliciously entered values that would result in disruptions to normal contract operation.

Long term, carefully document the caller-specified values that dictate important properties of the contracts, and ensure that they are properly constrained.

6. Reward distribution could be susceptible to sandwich attacks

Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-RIO-6
Target: `protocol/contracts/restaking/*`	

Description

The increase in value of user shares due to accrued staking rewards could be sandwich attacked to extract profit.

The protocol allows users to deposit assets in return for minting shares of a restaking token that increases in value based on the balance of the `RioLRTDepositPool` contract and the balance deposited into EigenLayer. Accrued execution layer and consensus layer rewards will flow from EigenLayer to the `RioLRTRewardDistributor` contract and through it to the `RioLRTDepositPool` contract, leading to an increase in the value of the restaking share tokens.

However, if the `rebalanceDelay` period has passed, a user could front run the transaction, which would increase the value of the shares by depositing tokens to the `RioLRTCoordinator` contract. The user could then back run the same transaction by requesting a withdrawal and then immediately finalize the withdrawal by calling the `rebalance` function. This would result in the user extracting a small amount of profit (up to the amount that the shares increased by), without their assets actually being deposited into EigenLayer, and would reduce the accrued rewards of other depositors.

However, the Rio Network protocol team plans to deploy an off-chain automation that will call the `rebalance` function whenever the `rebalanceDelay` passes, further decreasing the probability of this attack.

Exploit Scenario

Eve conducts a distributed denial-of-service attack against bots that periodically call the `verifyAndProcessWithdrawals` method on the `EigenPod` contract used by Rio Network and the `rebalance` method on Rio Network's coordinator. Eventually, the validator balances accrue a large excess over 32 ether, and the delay between `rebalance` calls passes. Eve executes the `verifyAndProcessWithdrawals` method on the `EigenPod` contract and then waits for the withdrawal delay to pass. Once it does, she executes a transaction that performs the following actions:

1. Take out an ETH flash loan.
2. Call `deposit` on the `RioLRTCoordinator` contract and deposit ETH until the deposit cap is reached, minting an equivalent amount of shares.
3. Execute the `claimDelayedWithdrawal` function on the `EigenPod` contract to sweep funds through the Rio Network reward distributor and to the deposit pool.
4. Call the `requestWithdrawal` function of the `RioLRTCoordinator` contract, withdrawing Eve's entire balance of the restaking token shares. Because the deposit pool contains sufficient funds, her withdrawal is immediately eligible for fulfillment.
5. Finalize the withdrawal by calling the `rebalance` function of the `RioLRTCoordinator` contract.
6. Pay back Eve's flash loan.

Due to owning a very large number of shares at the time when rewards were accrued, Eve gains a disproportionately large share of these rewards without ever having her funds allocated to EigenLayer operators.

Recommendations

Short term, consider delaying the accrual of rewards to an account for one epoch so that a rebalance must occur before rewards start being earned. One way to accomplish this would be by separating the deposit of assets and the minting of restaking token shares into a two-step process and ensuring that share tokens are minted only after the next rebalance.

Long term, investigate how timing attacks could influence the system, and document a clear specification of all timing-related risks.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, implementing them may enhance code readability and prevent the introduction of vulnerabilities in the future.

- **Shadowing variables is error-prone.** Although we did not identify any security risks as a result of shadowed variables, reusing existing variable names obfuscates developer intentions and could lead to the introduction of mistakes when the code changes. The following function parameters shadow state variables:
    - The `token` parameter in the `stakeERC20` method of `RioLRTOperatorDelegator.sol`
    - The `owner` parameter in the `allowance` method of `RioLRT.sol`
- **Incorrect NatSpec comment**. The `notice` line of the NatSpec comment associated with the `convertToSharesFromAsset` method in the `RioLRTAssetRegistry` contract appears to have been incorrectly copied from a different method.

D. Mutation Testing

This appendix outlines how we conducted mutation testing and highlights some of the most actionable results.

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite against each change. Changes that result in test failures indicate adequate test coverage, while changes that do not cause tests to fail indicate gaps in test coverage. Mutation testing allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

We used an experimental new mutation tool, `slither-mutate`, to conduct our mutation testing campaign. This tool is custom made for Solidity and features higher performance and fewer false positives than existing tools such as universalmutator.

Although this tool has not been formally released yet, it is open source and available for review in the `slither/tools/mutator` subdirectory on the `dev` branch of the `slither` repository.

The mutation campaign was run against the smart contracts using the following commands:

```
cd rio-monorepo/packages/protocol
slither-mutate ./contracts \
  --test-cmd='forge test' \
  --solc-remap='@openzeppelin=./node_modules/@openzeppelin,
@solady=./node_modules/@solady'
```

Figure D.1: A Bash script that runs a mutation testing campaign against each Solidity file in the `contracts` directory

Consider the following notes about the above commands:

- The overall runtime is approximately 12 hours on a consumer-grade laptop.
- The `--test-cmd` flag specifies the command to run to assess mutant validity. A `--fail-fast` flag will automatically be added to forge test commands to improve runtime.
- The `--solc-remap` flag tells `crytic-compile` where to find the necessary contract dependencies.

An abbreviated, illustrative example of a mutation test output file is shown in figure D.2.

```
INFO:Slither-Mutate:Mutating contract RioLRTWithdrawalQueue
INFO:Slither-Mutate:[RR] Line 117: 'amountsOut = new uint256[](requestLength)' ==>
'revert()' --> VALID
INFO:Slither-Mutate:[CR] Line 49: 'return currentEpochsByAsset[asset]' ==> '//return
currentEpochsByAsset[asset]' --> VALID
INFO:Slither-Mutate:[CR] Line 131: 'onlyCoordinator' ==> '//onlyCoordinator' -->
VALID
INFO:Slither-Mutate:[MIA] Line 95: '!epochWithdrawals.settled' ==> 'false' --> VALID
INFO:Slither-Mutate:[MIA] Line 99: 'userSummary.claimed' ==> 'false' --> VALID
```

Figure D.2: Abbreviated output from the mutation testing campaign on `RioLRTWithdrawalQueue.sol`

In summary, the following features of the `RioLRTWithdrawalQueue` contract lack sufficient tests:

- The `claimWithdrawalsForManyEpochs` method is not executed by any tests, as indicated by the uncaught revert mutant on line 117.
- No tests expect to operate in any epoch besides epoch zero. When line 49 is commented out, the `getCurrentEpoch` method will always return zero without causing any test failures.
- The failure case of the `onlyCoordinator` function modifier is not tested.
- Failure cases of input validation, such as those on lines 95 and 99 of the `claimWithdrawalsForEpoch` method, are not sufficiently tested.

We recommend that the Rio Network team review the existing tests and add verification that will catch the aforementioned types of mutations.

E. Automated Analysis Tool Configuration

Slither

We used Slither to detect common issues and anti-patterns in the codebase. Although Slither did not discover any severe issues during this review, integrating Slither into the project's testing environment can help find other issues that may be introduced during further development and will help improve the overall quality of the smart contracts' code.

```
slither . --no-fail-pedantic --exclude naming-convention,solc-version,similar-names
--filter-paths lib,test,node_modules
```

Figure E.1: An example Slither configuration

Integrating `slither-action` into the project's CI pipeline can automate this process.

Echidna

We used Echidna to perform fuzzing on the `OperatorUtilizationHeap` library, which is critical to the correct operation of the protocol. Our fuzzing campaign focused on verifying four essential properties of this component:

1. The root node must be the smallest node in the heap,
2. The children of the root node must be the largest nodes in the heap,
3. A node on a minimum level of the heap must be smaller than or equal to its child nodes.
4. A node on the maximum level of the heap must be larger than or equal to its child nodes.

Echidna managed to break the latter three properties by randomly inserting and removing nodes from the heap, which uncovered an issue with the implementation of the `remove` function (TOB-RIO-3). The fuzzing campaign was run using the following command:

```
echidna ./test/echidna/HeapTest.sol --contract HeapTest --corpus-dir echidna-heap
--test-mode assertion --test-limit 100000
```

Figure E.2: Command to run Echidna on the `OperatorUtilizationHeap` properties

Semgrep

The Semgrep tool identifies known anti-patterns and potential security vulnerabilities based on a library of lints. Semgrep can be installed with `brew install semgrep` or similar. We ran the tool with a rule designed to check unsafe casting, which uncovered one issue (TOB-RIO-1):

```
rules:
    - id: casting
      languages:
        - solidity
      message: Casting instance detected
      severity: INFO
      patterns:
        - pattern: $METHOD
        - metavariable-regex:
            metavariable: $METHOD
            regex: u?int(8|16|32|64|128|256)?\(
```

Figure E.3: The unsafe casting Semgrep rule

The Solidity support in Semgrep is still experimental.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From March 5 to March 7, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Rio Network team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the six issues described in this report, Rio Network has resolved five issues and has not resolved the one remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Unsafe casting of signed to unsigned integers	Resolved
2	Lack of two-step process for ownership transfers	Unresolved
3	Heap node removal leads to incorrect allocation and deallocation of shares	Resolved
4	Missing validation for sacrificial deposit amount	Resolved
5	Missing upper bound on rebalanceDelay system parameter	Resolved
6	Reward distribution could be susceptible to sandwich attacks	Resolved

Detailed Fix Review Results

TOB-RIO-1: Unsafe casting of signed to unsigned integers
Resolved. The `getETHUnderManagement` method of the `RioLRTOperatorDelegator` contract now features an explicit check that results in the function returning zero if the sum of the `EigenPod` shares and the balance is less than zero.

Similarly, the `queueOperatorStrategyExit` method features an explicit zero check that will leave the `sharesToExit` value at zero if the `EigenPod` shares are negative.

Lastly, the two affected methods of the `RioLRTOperatorRegistry` contract now use the `SafeCast` library instead of native casting to ensure these operations revert on overflow.

TOB-RIO-2: Lack of two-step process for ownership transfers
Unresolved. Rio Network has provided the following context:

> *Won't fix. Contracts will be owned by the RioDAO, ensuring that all changes in ownership undergo the comprehensive proposal process, providing sufficient oversight.*

TOB-RIO-3: Heap node removal leads to incorrect allocation and deallocation of shares
Resolved. A `_bubbleUp` call has been added to the `remove` method of the `OperatorUtilizationHeap` library. Additionally, the coverage of the fuzz tests for this library have been improved by the addition of a new parameter that will specify a random element to remove instead of removing only the minimum or maximum element.

TOB-RIO-4: Missing validation for sacrificial deposit amount
Resolved. A new `MIN_SACRIFICIAL_DEPOSIT` constant has been introduced and set to a value of `1,000`. This value is now used during an additional check in the `RioLRTIssuer` contract's `_deposit` method; the new check requires the sacrificial deposit to be above this minimum. Tests have been added to verify that this lower bound is being enforced appropriately.

TOB-RIO-5: Missing upper bound on rebalanceDelay system parameter
Resolved. A new `MAX_REBALANCE_DELAY` constant has been added to the `Constants.sol` file and set to a value of 3 days. This value is used by the `RioLRTCoordinator` contract's `_setRebalanceDelay` method to enforce an upper limit on the `rebalanceDelay` parameter. Tests have been added to verify that this upper bound is being enforced appropriately.

TOB-RIO-6: Reward distribution could be susceptible to sandwich attacks
Resolved. A new check has been added to the `RioLRTCoordinator` contract's `rebalance` method to verify that the message sender is `tx.origin`. As a result, smart contracts can no longer call the `rebalance` method, and flash loans cannot influence the rebalance process.

This change has the side effect of disqualifying some legitimate use cases and only indirectly mitigates the underlying risk by preventing the use of flash loans; flash-bot transaction bundles could still conduct sandwich attacks, although this would require a large amount of up-front capital.

However, the changes provided have an extremely small surface area for introducing new bugs. Alternative solutions that resolve this issue more fundamentally would require an overhaul of the internal accounting and would risk introducing new issues. Because this attack is not likely to be economically practical without flash loans, we consider the changes to fully resolve the issue. Thorough monitoring and robust agents that reliably call the `rebalance` method will further mitigate the risks described by this issue.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.