CASIMIR SELFSTAKE Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Hans

0kage

Assisting Auditors

July 10, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Casimir is a non-custodial Ethereum restaking platform built on EigenLayer's infrastructure, using Distributed Validator Technology (DVT). As a permissionless platform, Casimir is designed to align with the foundational principles of the Ethereum protocol and the broader Web3.0 ecosystem, focusing on secure, decentralized staking and digital asset management.

Key Features:

Validator Management: Validators on Casimir are openly registered, collateralized operators utilizing Distributed Validator Technology (DVT), which ensures high performance and reliability.

Key Security: Keys are managed through a trustless, zero-coordination distributed key generation (DKG) process, enhancing security and reducing the risk of mismanagement.

Restaking: Casimir leverages the robust EigenLayer infrastructure for restaking, allowing stakers to deposit ETH directly to selected Ethereum operators without the issuance of a Liquid Restaking Token. This method significantly reduces counterparty risks.

Withdrawal Flexibility: The platform supports full withdrawals, providing stakers with complete control over their assets without compromising on the accessibility of their funds.

By eliminating the need for Liquid Restaking Tokens and focusing on decentralized, secure key management, Casimir minimizes risks for users and strengthens the overall decentralization and efficiency of Ethereum staking.

Smart Contract Architecture Casimir's architecture is built around three core smart contracts: CasimirFactory, CasimirRegistry, and CasimirManager, designed to facilitate flexible and secure staking strategies on the Ethereum blockchain.

CasimirFactory

This contract acts as the central hub for registering multiple staking strategies. Each strategy defined by CasimirFactory specifies various parameters such as the size of the operator cluster, minimum collateral required from operators, reward fee etc. This modular approach allows for the customization of staking conditions to suit diverse staker needs and risk profiles.

CasimirRegistry and CasimirManager

For each strategy, a dedicated instance of CasimirManager and CasimirRegistry is deployed. These instances handle specific roles:

CasimirRegistry: Manages the operational aspects of validator nodes. This includes the registration, removal, and resharing of operator responsibilities. It ensures that the network of validators is maintained efficiently and securely.

CasimirManager: Facilitates user interactions such as staking and unstaking. This contract is the point of interface for users who wish to engage with the staking protocols defined by the CasimirFactory.

Beacon Upgradeability The protocol utilizes Beacon Upgradeability for managing future upgrades to both the Manager and Registry contracts. This upgradeability feature ensures that the system can evolve with EigenLayer and adapt to new requirements or improvements without sacrificing the integrity of ongoing operations.

Reporter A protocol-controlled address, known as the "Reporter," performs critical operational functions. These include creating, activating, syncing, and removing validators on EigenLayer, claiming rewards, and managing reports that update key protocol metrics such as rewards. This role is vital for the continuous and autonomous functioning of the system.

5 Audit Scope

Audit scope was limited solidity contracts that govern the core on-chain operations of Casimir. The following contracts were included in the scope of the audit

```
src/CasimirFactory.sol
src/CasimirManager.sol
src/CasimirRegistry.sol
src/libraries/Arrays.sol
src/libraries/Proxies.sol
src/interfaces/ICasimirCore.sol
src/interfaces/ICasimirFactory.sol
src/interfaces/ICasimirManager.sol
src/interfaces/ICasimirRegistry.sol
```

6 Executive Summary

Over the course of 19 days, the Cyfrin team conducted an audit on the CASIMIR SELFSTAKE smart contracts provided by CASIMIR LABS. In this period, a total of 27 issues were found.

Casimir Overview

Casimir is a non-custodial Ethereum restaking platform built on EigenLayer's infrastructure, using SSV Network's Distributed Validator Technology (DVT). Casimir leverages the SSV network to run a validator via a cluster of independent operators and EigenLayer for restaking ETH deposited by users. The audit we conducted focused on three main Solidity contracts of Casimir: CasimirFactory, CasimirRegistry, and CasimirManager.

Findings & Mitigations

- Critical Risks: We found six critical vulnerabilities that could allow attackers to cause significant damage at a low cost. These issues could:
  - Prevent users from retrieving their stakes or withdrawing their funds permanently.
  - Allow attackers to stop protocol from correctly updating rewards.

- High Risk: We identified six high risk vulnerabilities that could lead to financial losses for users or the platform through:
  - Mistakes in tracking funds that could result in lost rewards or inaccurate account balances.
  - Operators losing their security deposits.

All the vulnerabilities identified during the audit were fixed and verified.

Documentation

The technical documentation that clearly outlines the roles, process flows, and trust assumptions was not comprehensive. While the Casimir team provided us documentation during the audit, we recommend that a more rigorous document that details the complex reporting process and accounting calculations be maintained. This document should be version-controlled with regular updates as and when the accounting/reporting and core logic is upgraded by the Casimir team.

Testing

Casimir built its test suite on Foundry. Core protocol functionality is tested via a single end-to-end test that can be run on either mainnet or holesky forks. The test uses on-chain EigenLayer and SSV contracts, the external dependencies that Casimir contracts rely on.

The audit team highlighted certain unrealistic assumptions in the test setup that could potentially miss key edge cases and recommended suitable changes to make the tests more robust. Also, the audit team felt that a single end-to-end "happy path" test would not be sufficient to test all possible edge case scenarios that could potentially break the protocol functionality. It is recommended that the Casimir team increase the test coverage so that specific scenarios can be comprehensively tested.

Off-chain Operations

A significant part of Casimir's operations rely on off-chain automation. A range of operations including choosing operators for validator registration, activating validators with correct proofs, withdrawing validators, providing correct inputs to the reporting process, executing various stages of the reporting lifecycle, verifying and claiming rewards on Eigen Layer, removing and resharing operators, etc., are expected to be performed by "Reporter" role. Consistency, sequence, and timing of these operations rely on robust off-chain execution and monitoring which were out-of-scope for this audit. We recommend that the Casimir team also get an independent external audit done on its off-chain operations that guarantees performance, accuracy, and security.

Centralization Concerns

The audit team also identified a large degree of centralization, especially around the "Reporter" role. We also note that key contracts are using the Beacon Proxy Upgradeable pattern to manage future upgrades to the manager and registry contracts. While Casimir has a roadmap to progressively increase the level of decentralization, the audit team recommends implementation of security best practices that protect stakers from centralization risks.

Casimir team acknowledges centralization risks and plans to implement contract upgrades that significantly reduce the intervention of the "Reporter".

Conclusion

Casimir has taken an innovative approach to offering liquid staking and restaking services to Web3 users, uniquely doing so without the use of a separate token. This innovation has led to original engineering solutions. However, this approach has also introduced a considerable level of complexity to the protocol. Such complexity inevitably increases the potential security risks, as evidenced by a significant number of critical issues relative to the size of the codebase (measured in number of source lines of code, or nSLOC). All the vulnerabilities identified during the audit were successfully mitigated and verified. Casimir team acknowledges centralization risks and plans to progressively upgrade their contracts to reduce such risks.

Despite the rigorous efforts of our time-boxed audit conducted by two auditors, there remains a statistical likelihood that some vulnerabilities have gone undetected. Therefore, we recommend that a follow-up audit be conducted in the form of an open competition, leveraging the expertise of a broad community of independent auditors to ensure a more comprehensive examination of the protocol.

Summary

Project Name	CASIMIR SELFSTAKE
Repository	casimir-contracts
Commit	486c2b23113c...
Audit Timeline	Apr 23 - May 17th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	6
High Risk	6
Medium Risk	5
Low Risk	5
Informational	5
Gas Optimizations	0
Total Issues	27

Summary of Findings

[C-1] Attacker can cause a DOS during unstaking by intentionally reverting the transaction when receiving ETH	Resolved
[C-2] Function `claimEffectiveBalance()` may consistently revert, making it impossible to complete queue withdrawals	Resolved
[C-3] Delayed rewards can be claimed without updating internal accounting	Resolved
[C-4] Anyone can submit proofs via EigenPod `verifyAndProcessWithdrawals` to break the accounting of `withdrawRewards`	Resolved
[C-5] Front-run withdrawValidator by submitting proofs can permanently DOS validator unstaking on EigenLayer	Resolved
[C-6] Incorrect accounting of `tipBalance` can indefinitely stall report execution	Resolved
[H-1] Function `getTotalStake()` fails to account for pending validators, leading to inaccurate accounting	Resolved
[H-2] Hardcoded cluster size in `withdrawValidator` can cause losses to operators or protocol for strategies with larger cluster sizes	Resolved
[H-3] A malicious staker can force validator withdrawals by instantly staking and unstaking	Resolved
[H-4] Operator is not removed in Registry when validator has `owedAmount == 0`	Resolved
[H-5] Accounting for `rewardStakeRatioSum` is incorrect when a delayed balance or rewards are unclaimed	Resolved

[H-6] Incorrect accounting of `reportRecoveredEffectiveBalance` can prevent report from being finalized when a validator is slashed	Resolved
[M-1] Infinite loop in the `exitValidators()` prevents users from calling `requestUnstake()`	Resolved
[M-2] Multiple unstake requests can cause denial of service because withdrawn balance is not adjusted after every unstake request is fulfilled	Resolved
[M-3] Spamming `requestUnstake()` to cause a denial of service in the unstake queue	Resolved
[M-4] Users could avoid loss by frontrunning to request unstake	Resolved
[M-5] Centralization risks with a lot of power vested in the `Reporter` role	Acknowledged
[L-1] Operator can set his operatorID status to active by transferring 0 Wei	Resolved
[L-2] Reporter trying to reshare a pending validator will lead to denial of service	Resolved
[L-3] Missing implementation for EigenPod `withdrawNonBeaconChainETHBalanceWei` in CasimirManager	Resolved
[L-4] Function `withdrawRewards()` may lead to inaccuracy in `delayedRewards` if there's no withdrawal to process	Resolved
[L-5] Incorrect test setup leads to false test outcomes	Resolved
[I-1] Missing validations when initializing CasimirRegistry	Resolved
[I-2] ReentrancyGuardUpgradeable is not used in CasimirFactory and CasimirRegistry	Resolved
[I-3] The period check in `getNextUnstake()` always returns true	Resolved
[I-4] Unused function `validateWithdrawalCredentials()`	Resolved
[I-5] `getUserStake` function failure for non-staker accounts	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Attacker can cause a DOS during unstaking by intentionally reverting the transaction when receiving ETH

Description: The function `fulfillUnstake()` is used internally to fulfill unstake requests for users. It performs a low-level call to the `userAddress` to transfer ETH and reverses the transaction if the transfer fails. Moreover, the contract processes all unstake requests in a First-In-First-Out (FIFO) queue, meaning it must process earlier requests before handling later ones.

An attacker could exploit this by intentionally triggering a revert on the `receive()` function. This action would cause `fulfillUnstake()` to revert and block the entire unstake queue.

```
function fulfillUnstake(address userAddress, uint256 amount) private {
    (bool success,) = userAddress.call{value: amount}(""); // @audit DOS by reverting on `receive()`
    if (!success) {
        revert TransferFailed();
    }
    emit UnstakeFulfilled(userAddress, amount);
}
```

Impact: This can result in a Denial of Service for all unstake requests, thereby locking users' funds.

Recommended Mitigation: Consider using the Pull-over-Push pattern. Reference: https://fravoll.github.io/solidity-patterns/pull_over_push.html

Casimir: Fixed in cdbe7b1

Cyfrin: Verified.

7.1.2 Function `claimEffectiveBalance()` may consistently revert, making it impossible to complete queue withdrawals

Description: The function attempts to remove the withdrawal at index `0`, while it uses the withdrawal at index `i` to call `completeQueuedWithdrawal()`. Since each withdrawal can only be completed once, the `delayedEffectiveBalanceQueue[]` list will eventually contain withdrawals that have already been completed. When the function tries to complete a withdrawal that has already been completed, it invariably reverts.

```
for (uint256 i; i < delayedEffectiveBalanceQueue.length; i++) {
    IDelegationManager.Withdrawal memory withdrawal = delayedEffectiveBalanceQueue[i];
    if (uint32(block.number) - withdrawal.startBlock > withdrawalDelay) {
        delayedEffectiveBalanceQueue.remove(0); // @audit Remove withdrawal at index 0
        claimedEffectiveBalance += withdrawal.shares[0];
        eigenDelegationManager.completeQueuedWithdrawal(withdrawal, tokens, 0, true); // @audit
        ↪ Complete withdrawal of index i
    } else {
        break;
    }
}
```

Impact: The `claimEffectiveBalance()` function consistently reverts, making it impossible to complete queue withdrawals and therefore locking ETH.

Proof of Concept: Consider the following scenario:

1. Initially, the `delayedEffectiveBalanceQueue[]` list includes five withdrawals `[a, b, c, d, e]`.
2. The `claimEffectiveBalance()` function is called.

- In the first loop iteration `i = 0`, withdrawal `a` is removed and completed. The list now becomes `[b, c, d, e]`.
- In the second loop iteration `i = 1`, withdrawal `b` is removed, but withdrawal `c` is completed. The list now becomes `[c, d, e]`.
- In the third loop iteration `i = 2`, the function checks withdrawal `e` and assumes the withdrawal delay has not yet been reached. The loop breaks at this point and the function stops.

3. The next time the `claimEffectiveBalance()` function is called.
   - In the first loop iteration `i = 0`, the function tries to remove and complete withdrawal `c`. However, since withdrawal `c` has already been completed, the call to `completeQueuedWithdrawal()` will revert.

Recommended Mitigation: Consider using a consistent index for checking, removing and completing withdrawals.

Casimir: Fixed in 35fdf1e

Cyfrin: Verified.

7.1.3 Delayed rewards can be claimed without updating internal accounting

Description: The `claimRewards()` function is designed to claim delayed withdrawals from the EigenLayer Delayed Withdrawal Router and to update accounting variables such as `delayedRewards` and `reservedFeeBalance`.

```
function claimRewards() external {
    onlyReporter();

    uint256 initialWithdrawalsBalance = address(eigenWithdrawals).balance;
    eigenWithdrawals.claimDelayedWithdrawals(
        eigenWithdrawals.getClaimableUserDelayedWithdrawals(address(this)).length
    );
    uint256 claimedAmount = initialWithdrawalsBalance - address(eigenWithdrawals).balance;
    delayedRewards -= claimedAmount;

    uint256 rewardsAfterFee = subtractRewardFee(claimedAmount);
    reservedFeeBalance += claimedAmount - rewardsAfterFee;
    distributeStake(rewardsAfterFee);

    emit RewardsClaimed(rewardsAfterFee);
}
```

However, this function can be bypassed by directly executing the claim on the EigenLayer side via the `DelayedWithdrawalRouter::claimDelayedWithdrawals()` function. This function allows the caller to claim withdrawals for a specified recipient, with the recipient's address provided as an input. If the `CasimirManager` contract address is used as the `recipient`, the claim is made on its behalf.

Impact: This process does not update the accounting variables, leading to inaccurate accounting within the contract. Even though the rewards have been claimed, they are still accounted for in the `delayedRewards`, resulting in an incorrect total stake value.

Proof of Concept: EigenLayer contract that handles delayed withdrawal claims can be found here

Recommended Mitigation: Consider altering the way the contract manages rewards claims. This could be achieved by moving the accounting for claimed reward amounts to the `receive()` function, and by only filtering funds received from the `eigenWithdrawals` contract.

Casimir: Fixed in 4adef64

Cyfrin: Verified.

7.1.4 Anyone can submit proofs via EigenPod `verifyAndProcessWithdrawals` to break the accounting of `withdrawRewards`

Description: `CasimirManager::withdrawRewards` is an `onlyReporter` operation that performs the key tasks below:

1. Submits proofs related to the partial withdrawal of a validator at a given index.
2. Updates the `delayedRewards` based on the last element in the array of `userDelayedWithdrawalByIndex`.

Note that anyone, not just the pod owner, can submit proofs directly to `EigenPod::verifyAndProcessWithdrawals`. In such a case, the `delayedRewards` will not be updated, and the subsequent accounting during report finalization will be broken.

Any attempt to withdraw rewards by calling `CasimirManager::withdrawRewards` will revert because the withdrawal has already been processed. Consequently, `delayedRewards` will never be updated.

This same issue is applicable when submitting proofs for processing a full withdrawal. Critical accounting parameters that are updated in `CasimirManager::withdrawValidator` are effectively bypassed when proofs are directly submitted to EigenLayer.

Impact: If `delayedRewards` is not updated, the `rewardStakeRatioSum` and `latestActiveBalanceAfterFee` accounting will be broken.

Recommended Mitigation: EigenLayer does not restrict access to process withdrawals only to the pod owner. To that extent, access control to `CasimirManager::withdrawRewards` can always be bypassed. Assuming that all withdrawals will happen only through a reporter, consider adding logic that directly tracks the `eigenWithdrawals.delayedWithdrawals` and `eigenWithdrawals.delayedWithdrawalsCompleted` on EigenLayer to calculate delayedRewards.

Casimir: Fixed in eb31b43

Cyfrin: Verified.

7.1.5 Front-run withdrawValidator by submitting proofs can permanently DOS validator unstaking on EigenLayer

Description: An attacker can observe the mempool and front-run the `CasimirManager::withdrawValidator` transaction by submitting the same proofs directly on Eigen Layer.

Since the proof is already verified, the `CasimirManager::withdrawValidator` transaction will revert when it tries to submit the same proofs. Submitting an empty proof to bypass proof verification also does not work because the `finalEffectiveBalance` will always be 0, preventing the queuing of withdrawals.

```
function withdrawValidator(
    uint256 stakedValidatorIndex,
    WithdrawalProofs memory proofs,
    ISSVClusters.Cluster memory cluster
) external {
    onlyReporter();

   // ..code...

    uint256 initialDelayedRewardsLength = eigenWithdrawals.userWithdrawalsLength(address(this));
    ↪  //@note this holds the rewards
    uint64 initialDelayedEffectiveBalanceGwei = eigenPod.withdrawableRestakedExecutionLayerGwei();
    ↪  //@note this has the current ETH balance

>   eigenPod.verifyAndProcessWithdrawals(
        proofs.oracleTimestamp,
        proofs.stateRootProof,
        proofs.withdrawalProofs,
        proofs.validatorFieldsProofs,
        proofs.validatorFields,
```

```
        proofs.withdrawalFields
    ); //@audit reverts if proof is already verified

    {
        uint256 updatedDelayedRewardsLength = eigenWithdrawals.userWithdrawalsLength(address(this));
        if (updatedDelayedRewardsLength > initialDelayedRewardsLength) {
            IDelayedWithdrawalRouter.DelayedWithdrawal memory withdrawal =
                eigenWithdrawals.userDelayedWithdrawalByIndex(address(this),
                ↪  updatedDelayedRewardsLength - 1);
            if (withdrawal.blockCreated == block.number) {
                delayedRewards += withdrawal.amount;
                emit RewardsDelayed(withdrawal.amount);
            }
        }
    }

    uint64 updatedDelayedEffectiveBalanceGwei = eigenPod.withdrawableRestakedExecutionLayerGwei();
>   uint256 finalEffectiveBalance =
        (updatedDelayedEffectiveBalanceGwei - initialDelayedEffectiveBalanceGwei) * GWEI_TO_WEI;
        ↪  //@audit if no proofs submitted, this will be 0
    delayedEffectiveBalance += finalEffectiveBalance;
    reportWithdrawnEffectiveBalance += finalEffectiveBalance;
}

//... code
```

Impact: At a negligible cost, an attacker can prevent validator withdrawals on EigenLayer, creating an insolvency risk for Casimir.

Recommended Mitigation: Consider making the following changes to `CasimirManager::withdrawValidators`:

- Split the function into two separate functions, one for full-withdrawal verifications and another for queuing verified withdrawals.
- Listen to the following event emission in `EigenPod::_processFullWithdrawal` and filter the events emitted with the recipient as `CasimirManager`.

```
event FullWithdrawalRedeemed(
    uint40 validatorIndex,
    uint64 withdrawalTimestamp,
    address indexed recipient,
    uint64 withdrawalAmountGwei
);
```

- Introduce a try-catch while verifying withdrawal for each applicable validator index. If a proof is already verified, update the `stakedValidatorIds` array and reduce `requestedExits` in the catch section.
- Once all withdrawals are verified, use the event emissions to create a `QueuedWithdrawalParams` array that will be sent to the second function that internally calls `eigenDelegationManager.queueWithdrawals(params)`.
- Update the `delayedEffectiveBalance` and `reportWithdrawnEffectiveBalance` at this stage.

Note: This assumes that the `reporter` is protocol-controlled.

Casimir: Fixed in eb31b43

Cyfrin: Verified. Withdrawal proofs are decoupled from queuing withdrawals on EigenLayer. This successfully mitigates the Denial of Service risk reported in this issue. It is noted however that the effective balance is hardcoded as 32 ether.

It is recommended that the effective balance is passed as a parameter by monitoring the full withdrawal event on EigenLayer.

7.1.6 Incorrect accounting of `tipBalance` can indefinitely stall report execution

Description: The `receive` fallback function in `CasimirManager` increases `tip` balance if the sender is not the `DelayedWithdrawalRouter`. The implicit assumption here is that all withdrawals, full or partial, are routed via the `DelayedWithdrawalRouter`. While this assumption is true incase of partial withdrawals (rewards), this is an incorrect assumption for full withdrawals where the sender is not the `DelayedWithdrawalRouter` but the `EigenPod` itself.

```
    receive() external payable {
        if (msg.sender != address(eigenWithdrawals)) {
            tipBalance += msg.value; //@audit tip balance increases even incase of full withdrawals

            emit TipsReceived(msg.value);
        }
    }
```

Note that the owner can fully withdraw any tips via the `CasimirManager:claimTips`

```
    function claimTips() external {
        onlyReporter();

        uint256 tipsAfterFee = subtractRewardFee(tipBalance);
        reservedFeeBalance += tipBalance - tipsAfterFee;
        tipBalance = 0;
        distributeStake(tipsAfterFee);
        emit TipsClaimed(tipsAfterFee);
    }
```

When tips are claimed by owner, the `withdrawnEffectiveBalance` which was incremented while claiming full withdrawals via `CasimirManager::claimEffectiveBalance` will be out-of-sync with the actual ETH balance. Effectively, the key invariant here that `CasimirManager.balance >= withdrawnEffectiveBalance` can get violated.

```
    function claimEffectiveBalance() external {
        onlyReporter();

        IStrategy[] memory strategies = new IStrategy[](1);
        strategies[0] =
        ↪  IDelegationManagerViews(address(eigenDelegationManager)).beaconChainETHStrategy();
        IERC20[] memory tokens = new IERC20[](1);
        tokens[0] = IERC20(address(0));

        uint256 withdrawalDelay = eigenDelegationManager.strategyWithdrawalDelayBlocks(strategies[0]);
        uint256 claimedEffectiveBalance;
        for (uint256 i; i < delayedEffectiveBalanceQueue.length; i++) {
            IDelegationManager.Withdrawal memory withdrawal = delayedEffectiveBalanceQueue[i];
            if (uint32(block.number) - withdrawal.startBlock > withdrawalDelay) {
                delayedEffectiveBalanceQueue.remove(0);
                claimedEffectiveBalance += withdrawal.shares[0];
                eigenDelegationManager.completeQueuedWithdrawal(withdrawal, tokens, 0, true);
            } else {
                break;
            }
        }

        delayedEffectiveBalance -= claimedEffectiveBalance;
        withdrawnEffectiveBalance += claimedEffectiveBalance; //@audit this accounting entry should be
        ↪  backed by ETH balance at all times
    }
```

Impact: Accounting error in `tipBalance` calculation can cause failure of `fulfilUnstakes` as the `withdrawnEffectiveBalance` is out-of-sync with actual ETH balance in CasimirManager. This can potentially stall report execution from time to time.

Recommended Mitigation: Consider incrementing `tipBalance` only if the sender is neither the `DelayedWithdrawalRouter` nor the `EigenPod`

Casimir: Fixed in 4adef64

Cyfrin: Verified.

7.2 High Risk

7.2.1 Function `getTotalStake()` fails to account for pending validators, leading to inaccurate accounting

Description: The `getTotalStake()` function is the core accounting function to calculate the total stake of the `CasimirManager`. It's used to compute the change for `rewardStakeRatioSum` within the `finalizeReport()` function.

```
function getTotalStake() public view returns (uint256 totalStake) {
  // @audit Validators in pending state is not accounted for
  totalStake = unassignedBalance + readyValidatorIds.length * VALIDATOR_CAPACITY +
  ↪  latestActiveBalanceAfterFee
      + delayedEffectiveBalance + withdrawnEffectiveBalance + subtractRewardFee(delayedRewards) -
      ↪  unstakeQueueAmount;
}
```

This function aggregates the stakes from various sources, including the `32 ETH` from each "ready validator" (`readyValidatorIds.length * VALIDATOR_CAPACITY`) and the ETH staked in "staked validators" (`latestActiveBalanceAfterFee`).

However, it fails to account for the ETH in pending validators.

A validator must go through three steps to become active/staked:

1. Every time users make a deposit, the contract checks if the unassigned balance has reached `32 ETH`. If it has, the next validator ID is added to `readyValidatorIds`.
2. The reporter calls `depositValidator()` to deposit `32 ETH` from the validator into the beacon deposit contract. In this step, the validator ID moves from `readyValidatorIds` to `pendingValidatorIds`.
3. The reporter calls `activateValidator()`, which moves the validator ID from `pendingValidatorIds` to `stakedValidatorIds` and updates `latestActiveBalanceAfterFee` to reflect the total stake in the beacon chain.

As shown, the `getTotalStake()` function accounts for validators in steps 1 and 3, but ignores the stake of validators in the pending state (step 2). In the current design, there is nothing that stops report finalization if pending validators > 0.

Impact: Function getTotalStake() will return the value of total stake less than it should be. The result is rewardStakeRatioSum calculation will be incorrect in a scenario where all pending validators are not activated before report finalization.

```
uint256 totalStake = getTotalStake();

rewardStakeRatioSum += Math.mulDiv(rewardStakeRatioSum, gainAfterFee, totalStake);

rewardStakeRatioSum += Math.mulDiv(rewardStakeRatioSum, gain, totalStake);

rewardStakeRatioSum -= Math.mulDiv(rewardStakeRatioSum, loss, totalStake);
```

Recommended Mitigation: Consider adding `pendingValidatorIds.length * VALIDATOR_CAPACITY` to function `getTotalStake()`.

Casimir: Fixed in 10fe228

Cyfrin: Verified.

7.2.2 Hardcoded cluster size in `withdrawValidator` can cause losses to operators or protocol for strategies with larger cluster sizes

Description: `CasimirManager::withdrawValidator` calculates the owed balance on withdrawal, ie. shortfall from the initial 32 ether. It then tries to recover the owed amount from the operators tagged to the validator. However,

while calculating recovery amount, a hardcoded cluster size of `4` is used.

```
function withdrawValidator(
    uint256 stakedValidatorIndex,
    WithdrawalProofs memory proofs,
    ISSVClusters.Cluster memory cluster
) external {
    onlyReporter();

    // ... more code

    uint256 owedAmount = VALIDATOR_CAPACITY - finalEffectiveBalance;
    if (owedAmount > 0) {
        uint256 availableCollateral = registry.collateralUnit() * 4;
        owedAmount = owedAmount > availableCollateral ? availableCollateral : owedAmount;
>       uint256 recoverAmount = owedAmount / 4; //@audit hardcoded operator size
        for (uint256 i; i < validator.operatorIds.length; i++) {
            registry.removeOperatorValidator(validator.operatorIds[i], validatorId, recoverAmount);
        }
    }

    // .... more code
```

Impact: This has 2 side effects:

1. Operators lose higher % of collateral balance in strategies with large cluster sizes
2. On the other hand, since `owedAmount` is capped to `4 * collateralUnit`, it is also likely that protocol ends up recovering less than it should.

Recommended Mitigation: Consider using the `clusterSize` of the strategy instead of a hardcoded number.

Casimir: Fixed in 7497e8c.

Cyfrin: Verified.

7.2.3 A malicious staker can force validator withdrawals by instantly staking and unstaking

Description: When a user unstakes via `CasimirManager::requestUnstake`, the number of required validator exits is calculated using the prevailing expected withdrawable balance as follows:

```
function requestUnstake(uint256 amount) external nonReentrant {
    // code ....
    uint256 expectedWithdrawableBalance =
        getWithdrawableBalance() + requestedExits * VALIDATOR_CAPACITY + delayedEffectiveBalance;
    if (unstakeQueueAmount > expectedWithdrawableBalance) {
        uint256 requiredAmount = unstakeQueueAmount - expectedWithdrawableBalance;
>       uint256 requiredExits = requiredAmount / VALIDATOR_CAPACITY; //@audit required exits calculated
↪   here
        if (requiredAmount % VALIDATOR_CAPACITY > 0) {
            requiredExits++;
        }
        exitValidators(requiredExits);
    }

    emit UnstakeRequested(msg.sender, amount);
}
```

Consider the following simplified scenario:

`unAssignedBalance = 31 ETH withdrawnBalance = 0 delayedEffectiveBalance = 0 requestedExits = 0`

Also, for simplicity, assume the `deposit fees = 0%`

Alice, a malicious validator, stakes 1 ETH. This allocates the unassigned balance to a new validator via `distributeStakes`. At this point, the state is:

`unAssignedBalance = 0 ETH withdrawnBalance = 0 delayedEffectiveBalance = 0 requestedExits = 0`

Alice instantly places an unstake request for 1 ETH via `requestUnstake`. Since there is not enough balance to fulfill unstakes, an existing validator will be forced to withdraw from the Beacon Chain. After this, the state will be:

`unAssignedBalance = 0 ETH withdrawnBalance = 0 delayedEffectiveBalance = 0 requestedExits = 1`

Now, Alice can repeat the attack, this time by instantly depositing and withdrawing 64 ETH. At the end of this, the state will be:

`unAssignedBalance = 0 ETH withdrawnBalance = 0 delayedEffectiveBalance = 0 requestedExits = 2`

Each time, Alice only has to lose the deposit fee & gas fee but can grief the genuine stakers who lose their potential rewards & the operators who are forcefully kicked out of the validator.

Impact: Unnecessary validator withdrawal requests grief stakers, operators and protocol itself. Exiting validators causes a loss of yield to stakers and is very gas intensive for protocol.

Recommended Mitigation:

- Consider an unstake lock period. A user cannot request unstaking until a minimum time/blocks have elapsed after deposit.
- Consider removing ETH from `readyValidators` instead of exiting validators first -> while active validators are already accruing rewards, ready Validators have not yet started the process. And the overhead related to removing operators, de-registering from the SSV cluster is not needed if ETH is deallocated from ready validators.

Casimir: Fixed in 4a5cd14

Cyfrin: Verified.

7.2.4 Operator is not removed in Registry when validator has `owedAmount == 0`

Description: `CasimirManager::withdrawValidator()` function is designed to remove a validator after a full withdrawal. It checks whether the final effective balance of the removed validator is sufficient to cover the initial 32 ETH deposit. If for some reason such as slashing, the final effective balance is less than 32 ETH, the operators must recover the missing portion by calling `registry.removeOperatorValidator()`.

```
uint256 owedAmount = VALIDATOR_CAPACITY - finalEffectiveBalance;
if (owedAmount > 0) {
    uint256 availableCollateral = registry.collateralUnit() * 4;
    owedAmount = owedAmount > availableCollateral ? availableCollateral : owedAmount;
    uint256 recoverAmount = owedAmount / 4;
    for (uint256 i; i < validator.operatorIds.length; i++) {
        // @audit if owedAmount == 0, this function is not called
        registry.removeOperatorValidator(validator.operatorIds[i], validatorId, recoverAmount);
    }
}
```

However, the `removeOperatorValidator()` function also has the responsibility to update other operators' states, such as `operator.validatorCount`. If this function is only called when `owedAmount > 0`, the states of these operators will not be updated if the validator fully returns 32 ETH.

Impact: The `operator.validatorCount` will not decrease in the `CasimirRegistry` when a validator is removed. As a result, the operator cannot withdraw the collateral for this validator, and the collateral will remain locked in the `CasimirRegistry` contract.

Recommended Mitigation: The function `registry.removeOperatorValidator()` should also be called with `recoverAmount = 0` when `owedAmount == 0`. This will free up collateral for operators.

Casimir: Fixed in d7b35fc

Cyfrin: Verified.

7.2.5 Accounting for `rewardStakeRatioSum` is incorrect when a delayed balance or rewards are unclaimed

Description: The current accounting incorrectly assumes that the delayed effective balance and delayed rewards will be claimed before any new report begins. This is inaccurate as these delayed funds require a few days before they can be claimed. If a new report starts before these funds are claimed, the `reportSweptBalance` will account for them again. This double accounting impacts the `rewardStakeRatioSum` calculation, leading to inaccuracies.

Impact: The accounting for `rewardStakeRatioSum` is incorrect, which leads to an inaccurate user stake. Consequently, users may receive more ETH than anticipated upon unstaking.

Proof of Concept: Consider the following scenario:

1. Initially, we assume that one validator (32 ETH) is staked and the beacon chain reward is 0.105 ETH. The report gets processed.

```
// Before start
latestActiveBalanceAfterFee = 32 ETH
latestActiveRewards = 0

// startReport()
reportSweptBalance = 0 (rewards is in BeaconChain)

// syncValidators()
reportActiveBalance = 32.105 ETH

// finalizeReport()
rewards = 0.105 ETH
change = rewards - latestActiveRewards = 0.105 ETH
gainAfterFee = 0.1 ETH
=> rewardStakeRatioSum is increased
=> latestActiveBalanceAfterFee = 32.1

sweptRewards = 0
=> latestActiveRewards = 0.105
```

2. The beacon chain sweeps 0.105 ETH rewards. This is followed by processing another report.

```
// Before start
latestActiveBalanceAfterFee = 32.1 ETH
latestActiveRewards = 0.105

// startReport()
reportSweptBalance = 0.105 (rewards is in EigenPod)

// syncValidators()
reportActiveBalance = 32 ETH

// finalizeReport()
rewards = 0.105 ETH
change = rewards - latestActiveRewards = 0
=> No update to rewardStakeRatioSum and latestActiveBalanceAfterFee

sweptRewards = 0.105
=> latestActiveBalanceAfterFee = 32 ETH (subtracted sweptReward without fee)
```

```
=> latestActiveRewards = rewards - sweptRewards = 0
```

3. Suppose no actions take place, which means the rewards is still in EigenPod and not claimed yet. The next report gets processed.

```
// Before start
latestActiveBalanceAfterFee = 32 ETH
latestActiveRewards = 0

// startReport()
reportSweptBalance = 0.105 (No action happens so rewards is still in EigenPod)

// syncValidators()
reportActiveBalance = 32 ETH

// finalizeReport()
rewards = 0.105 ETH
change = rewards - latestActiveRewards = 0.105
=> rewardStakeRatioSum is increased
=> latestActiveBalanceAfterFee = 32.1

sweptRewards = 0.105
=> latestActiveBalanceAfterFee = 32 ETH (subtracted sweptReward without fee)
=> latestActiveRewards = rewards - sweptRewards = 0
```

Since no actions occur between the last report and the current one, the values of `latestActiveBalanceAfterFee` and `latestActiveReward` remain the same. However, the `rewardStakeRatioSum` value increased from nothing. If this reporting process continues, the `rewardStakeRatioSum` could infinitely increase. Consequently, the core accounting of user stakes becomes incorrect, and users could receive more ETH than expected when unstaking.

Recommended Mitigation: Review the accounting logic to ensure that the delayed effective balance and delayed reward are only accounted for once in the reports.

Casimir Fixed in eb31b43

Cyfrin Verified.

7.2.6 Incorrect accounting of `reportRecoveredEffectiveBalance` can prevent report from being finalized when a validator is slashed

Description: When a validator is slashed, a loss is incurred. In the `finalizeReport()` function, the `rewardStakeRatioSum` and `latestActiveBalanceAfterFee` variables are reduced to reflect this loss. The change could be positive if the rewards are larger than the slashed amount, but for simplicity, we'll focus on the negative case. This is where the loss is accounted for.

```
} else if (change < 0) {
    uint256 loss = uint256(-change);
    rewardStakeRatioSum -= Math.mulDiv(rewardStakeRatioSum, loss, totalStake);
    latestActiveBalanceAfterFee -= loss;
}
```

However, any loss will be recovered by the node operators' collateral in the `CasimirRegistry`. From the users' or pool's perspective, there is no loss if it is covered, and users will receive compensation in full. The missing accounting here is that `rewardStakeRatioSum` and `latestActiveBalanceAfterFee` need to be increased using the `reportRecoveredEffectiveBalance` variable.

Impact: Users or pools suffer a loss that should be covered by `reportRecoveredEffectiveBalance`. Incorrect accounting results in `latestActiveBalanceAfterFee` being less than expected. This in certain scenarios could

lead to arithmetic underflow & prevent report from being finalized. Without report finalization, new validators cannot be activated & a new report period cannot be started.

Proof of Concept: Consider following scenario - there is an underflow when last validator is withdrawn that prevents report from being finalized.

Report Period 0 2 validators added

Report Period 1 Rewards: 0.1 per validator on BC. Withdrawal: 32 Unstake request: 15

```
=> start
Eigenpod balance = 32.1
reportSweptBalance = 32.1

=> syncValidator
reportActiveBalance = 32.1
reportWithdrawableValidators = 1

=>withdrawValidator
delayedRewards = 0.1
Slashed = 0
Report Withdrawn Effective Balance = 32
Delayed Effective Balance = 32
Report Recovered Balance = 0

=> finalize
totalStake = 49
expectedWithdrawalEffectiveBalance = 32
expectedEffectiveBalance = 32

Rewards = 0.2

rewardStakeRatioSum = 1004.08
latestActiveBalanceAfterFee (reward adj.) = 64.2
swept rewards = 0.1

latestActiveBalanceAfterFee (swept reward adj) = 64.1
latestActiveBalanceAfterFee (withdrawals adj) = 32.1
latestActiveRewards = 0.1
```

Report Period 2 unstake request: 20 last validator exited with slashing of 0.2

```
=> start
Eigenpod balance = 63.9 (32.1 previous + 31.8 slashed)
Delayed effective balance = 32
Delayed rewards = 0.1

reportSweptBalance = 96

=> sync validator
reportActiveBalance = 0
reportWithdrawableValidators = 1

=> withdraw validator
Delayed Effective Balance = 63.8 (32+ 31.8)
Report Recovered Balance = 0.2
Report Withdrawn Effective Balance = 31.8 + 0.2 = 32
Delayed Rewards = 0.1
```

```
=> finalizeReport
Total Stake: 29.2
expectedWithdrawalEffectiveBalance = 32
expectedEffectiveBalance = 0
rewards = 64

Change = 63.9
rewardStakeRatioSum: 3201.369
latestActiveBalanceAfterFee (reward adj) = 96
Swept rewards = 64.2

latestActiveBalanceAfterFee (swept reward adj) = 31.8
latestActiveBalanceAfterFee (adj withdrawals) = -0.2 => underflow

latestActiveRewards = -0.2
```

Arithmetic underflow here. Correct adjustment is by including `reportRecoveredBalance` in rewards. On correction, the following state is achieved:

=> finalizeReport

```
Total Stake: 29.2
expectedWithdrawalEffectiveBalance = 32
expectedEffectiveBalance = 0
rewards = 64.2 (add 0.2 recoveredEffectiveBalance)

Change = 64.1
rewardStakeRatioSum: 3208.247
latestActiveBalanceAfterFee (reward adj) = 96.2
Swept rewards = 64.2

latestActiveBalanceAfterFee (swept reward adj) = 32
latestActiveBalanceAfterFee (adj withdrawals) = 0

latestActiveRewards = 0
```

Recommended Mitigation: Consider adding `reportRecoveredEffectiveBalance` to `rewards` calculation so that recovered ETH is accounted for in `rewardStakeRatioSum` and `latestActiveBalanceAfterFee` calculations.

Casimir: Fixed in eb31b43.

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 Infinite loop in the `exitValidators()` prevents users from calling `requestUnstake()`

Description: When users call function `requestUnstake()` to request to unstake their ETH, the `CasimirManager` contract will calculate if the current expected withdrawable balance is enough to cover all the queued unstaking requests. If it is not enough, the function `exitValidators()` will be call to exit some active validators to have enough ETH to fulfill all the unstaking requests.

In the function `exitValidators()`, it will do a while loop through the `stakedValidatorIds` list. If it found an active validator, it will call the `ssvClusters` to exit this validator and also change the status from `ACTIVE` to `EXITING`. However, if the validator status is not `ACTIVE`, the loop `index` will not be updated as well, resulting in the loop keep running infinitely but not being able to reach the next validator in the `stakedValidatorIds` list.

```
function exitValidators(uint256 count) private {
    uint256 index = 0;
    while (count > 0) {
        uint32 validatorId = stakedValidatorIds[index];
        Validator storage validator = validators[validatorId];

        // @audit if status != ACTIVE, count and index won't be updated => Infinite loop
        if (validator.status == ValidatorStatus.ACTIVE) {
            count--;
            index++;
            requestedExits++;
            validator.status = ValidatorStatus.EXITING;
            ssvClusters.exitValidator(validator.publicKey, validator.operatorIds);
            emit ValidatorExited(validatorId);
        }
    }
}
```

Impact:

- If the status of first validator in the `stakedValidatorIds` list is not active, the `requestUnstake()` function will consume the caller's entire gas limit and revert.
- Could also lead to griefing attacks where a small staker can delay unstake requests of a whale staker by front-running an unstake request. While `exitValidators` will run successfully the first time, it will revert due to infinite loop when called by whale staker

Recommended Mitigation: Consider updating `count` and `index` variables to ensure the loop will break in all scenarios.

Casimir: Fixed in 2945695

Cyfrin: Verified.

7.3.2 Multiple unstake requests can cause denial of service because withdrawn balance is not adjusted after every unstake request is fulfilled

Description: A `while` loop runs over a specific number of unstake requests, and in every iteration, it checks if an unstake request is fulfillable. If it is, the unstaked amount is transferred back to the staker who requested unstaking. Honoring every unstaking request reduces the effective ETH balance in the manager, however, the `getNextUnstake` function continues to use the stale `withdrawnEffectiveBalance` while checking if the next unstake request is fulfillable.

In fact, `withdrawnEffectiveBalance` is adjusted only after the completion of the `while` loop.

```
function fulfillUnstakes(uint256 count) external {
    //@note called when report status is in fulfilling unstakes
    onlyReporter();
```

```
    if (reportStatus != ReportStatus.FULFILLING_UNSTAKES) {
        revert ReportNotFulfilling();
    } //@note ok => report has to be in this state

    uint256 unstakedAmount;
    while (count > 0) {
        count--;

>       (Unstake memory unstake, bool fulfillable) = getNextUnstake(); //@audit uses the stale
↪   withdrawn balance
        if (!fulfillable) {
            break;
        }

        unstakeQueue.remove(0);
>       unstakedAmount += unstake.amount; //@audit unstakedAmount is increased here
>       fulfillUnstake(unstake.userAddress, unstake.amount); //@audit even after ETH is transferred,
↪   withdrawn balance is same
    }

    (, bool nextFulfillable) = getNextUnstake();
    if (!nextFulfillable) {
        reportStatus = ReportStatus.FINALIZING;
    }

>   if (unstakedAmount <= withdrawnEffectiveBalance) { //@audit withdrawn balance and unassigned
↪   balance adjustment happens here
        withdrawnEffectiveBalance -= unstakedAmount;
    } else {
        uint256 remainder = unstakedAmount - withdrawnEffectiveBalance;
        withdrawnEffectiveBalance = 0;
        unassignedBalance -= remainder;
    }

    unstakeQueueAmount -= unstakedAmount;
}
```

Impact: The `fulfillUnstakes()` function may fulfill more requests than the allowable withdrawable balance. This could cause the function to overflow and revert at the end.

Proof of Concept: Recommended Mitigation: Consider updating `withdrawnEffectiveBalance` after an unstake request has been fulfilled by the `fulfillUnstakes()` function.

Casimir: Fixed in 9f8920f

Cyfrin: Verified.

7.3.3 Spamming `requestUnstake()` to cause a denial of service in the unstake queue

Description:

- The function `requestUnstake()` allows users to request any amount, even `amount = 0`.
- The contract processes all unstake requests in a First-In-First-Out (FIFO) queue, handling earlier requests before later ones.
- The `remove()` function has a time complexity of O(n), which consumes gas.

This means an attacker could repeatedly call `requestUnstake()` to enlarge the unstake queue, causing the gas consumption of `fulfillUnstake()` to exceed the block gas limit.

Impact: Excessive gas usage when calling `fulfillUnstake()` could exceed the block gas limit, causing a DOS.

Recommended Mitigation: Consider setting a minimum unstake amount for the `requestUnstake()` function that is substantial enough to make spamming impractical.

Casimir: Fixed in 4a5cd14

Cyfrin: Verified.

7.3.4 Users could avoid loss by frontrunning to request unstake

Description: A loss can occur when a validator is slashed, which is reflected in the `finalizeReport()` function. If the `change` is less than 0, this indicates that a loss has occurred. Consequently, the accounting updates the `rewardStakeRatioSum` to decrease the stake value of all users in the `CasimirManager`.

```
} else if (change < 0) {
    uint256 loss = uint256(-change);
    rewardStakeRatioSum -= Math.mulDiv(rewardStakeRatioSum, loss, totalStake);
    latestActiveBalanceAfterFee -= loss;
}
```

However, users can avoid this loss by front-running an unstake request. This is because they can create and fulfill an unstake request within the same `reportPeriod`. If users anticipate a potential loss in the next report (by watching the mempool), they can avoid it by requesting to unstake. The contract processes all unstake requests in a First-In-First-Out (FIFO) queue, meaning reporters must fulfill earlier requests before later ones.

```
function getNextUnstake() public view returns (Unstake memory unstake, bool fulfillable) {
    // @audit Allow to create and fulfill unstake within the same `reportPeriod`
    if (unstakeQueue.length > 0) {
        unstake = unstakeQueue[0];
        fulfillable = unstake.period <= reportPeriod && unstake.amount <= getWithdrawableBalance();
    }
}
```

Impact: This can lead to unfairness. The front-runner can avoid losses while retaining all profits.

Recommended Mitigation: Consider implementing a waiting or delay period for unstake requests before they can be fulfilled. Do not allow the unstake request to be fulfilled in the same `reportPeriod` in which it was created. Additionally, considering adding a small user fee for unstaking.

Casimir: Fixed in 28baa81

Cyfrin: Verified.

7.3.5 Centralization risks with a lot of power vested in the `Reporter` role

Description: In the current design, the `Reporter`, a protocol-controlled address, is responsible for executing a number of mission-critical operations. Only `Reporter` operations include starting & finalizing a report, selecting & replacing operators, syncing/activating/withdrawing and depositing validators, verifying & claiming rewards from EigenLayer, etc. Also noteworthy is the fact that the timing and sequence of these operations are crucial for the proper functioning of the Casimir protocol.

Impact: With so many operations controlled by a single address, a significant part of which are initiated off-chain, the protocol is exposed to all the risks associated with centralization. Some of the known risks include:

- Compromised/lost private keys that control the `Reporter` address
- Rogue admin
- Network downtime
- Human/Automation errors associated with the execution of multiple operations

Recommended Mitigation: While we understand that the protocol in the launch phase wants to retain control over mission-critical parameters, we strongly recommend implementing the following even at the launch phase:

- Continuous monitoring of off-chain processes
- Reporter automation via a multi-sig

In the long term, the protocol should consider a clear path towards decentralization.

Casimir: Acknowledged. We plan to implement the expected EigenLayer checkpoint upgrade that significantly reduces the intervention of the reporter while syncing validator balances.

Cyfrin: Acknowledged.

7.4 Low Risk

7.4.1 Operator can set his operatorID status to active by transferring 0 Wei

Description: When withdrawing collateral, logic checks if collateral balance is 0 & makes the operator Id inactive.

```
function withdrawCollateral(uint64 operatorId, uint256 amount) external {
        onlyOperatorOwner(operatorId);

        Operator storage operator = operators[operatorId];
        uint256 availableCollateral = operator.collateralBalance - operator.validatorCount *
        ↪  collateralUnit; //@note can cause underflow here if validator count > 0
        if (availableCollateral < amount) {
            revert InvalidAmount();
        }

        operator.collateralBalance -= amount;
        if (operator.collateralBalance == 0) {
            operator.active = false;
        }

        (bool success,) = msg.sender.call{value: amount}("");
        if (!success) {
            revert TransferFailed();
        }

        emit CollateralWithdrawn(operatorId, amount);
    }
```

However while depositing, there is no check on the amount deposited. An operator can deposit 0 Wei and set operatorID to active. Deposit and withdrawal states are inconsistent.

```
     function depositCollateral(uint64 operatorId) external payable {
        onlyOperatorOwner(operatorId);

        Operator storage operator = operators[operatorId];
        if (!operator.registered) {
            operatorIds.push(operatorId);
            operator.registered = true;
            emit OperatorRegistered(operatorId);
        }
        if (!operator.active) {
>            operator.active = true; //@audit -> can make operator active even with 0 wei
        }
        operator.collateralBalance += msg.value;

        emit CollateralDeposited(operatorId, msg.value);
    }**
```

Impact: Inconsistent logic when adding and removing validators.

Recommended Mitigation: Consider checking that collateral amount in the `depositCollateral` function

Casimir: Fixed in 109cf2a

Cyfrin: Verified.

7.4.2 Reporter trying to reshare a pending validator will lead to denial of service

Description: A validator can reshare an operator if its either in `PENDING` or `ACTIVE` status. When resharing is executed for a validtor in `PENDING` state , the existing operators are removed from the SSV cluster -> however, no such operators are registered in the first place. This is because SSV registration does not happen when `depositStake` is called.

```
function reshareValidator(
        uint32 validatorId,
        uint64[] memory operatorIds,
        uint64 newOperatorId,
        uint64 oldOperatorId,
        bytes memory shares,
        ISSVClusters.Cluster memory cluster,
        ISSVClusters.Cluster memory oldCluster,
        uint256 feeAmount,
        uint256 minTokenAmount,
        bool processed
    ) external {
        onlyReporter();

        Validator storage validator = validators[validatorId];
        if (validator.status != ValidatorStatus.ACTIVE && validator.status != ValidatorStatus.PENDING) {
            revert ValidatorNotActive();
        }

       // ... code

        uint256 ssvAmount = retrieveFees(feeAmount, minTokenAmount, address(ssvToken), processed);
        ssvToken.approve(address(ssvClusters), ssvAmount);
>        ssvClusters.removeValidator(validator.publicKey, validator.operatorIds, oldCluster); //@audit
↪   validtor key is not registered when the validator is in pending state
       ssvClusters.registerValidator(validator.publicKey, operatorIds, shares, ssvAmount, cluster);
        ↪  //@audit new operators registered

        validator.operatorIds = operatorIds;
        validator.reshares++;

        registry.removeOperatorValidator(oldOperatorId, validatorId, 0);
        registry.addOperatorValidator(newOperatorId, validatorId);

        emit ValidatorReshared(validatorId);
    }
```

`SSVCluster::removeValidator` reverts when it can't find a validator data to remove.

```
    function removeValidator(
        bytes calldata publicKey,
        uint64[] memory operatorIds,
        Cluster memory cluster
    ) external override {
        StorageData storage s = SSVStorage.load();

        bytes32 hashedCluster = cluster.validateHashedCluster(msg.sender, operatorIds, s);
        bytes32 hashedOperatorIds = ValidatorLib.hashOperatorIds(operatorIds);

        bytes32 hashedValidator = keccak256(abi.encodePacked(publicKey, msg.sender));
        bytes32 validatorData = s.validatorPKs[hashedValidator];

        if (validatorData == bytes32(0)) {
>            revert ISSVNetworkCore.ValidatorDoesNotExist(); //@audit reverts when no key exists
```

```
        }

        if (!ValidatorLib.validateCorrectState(validatorData, hashedOperatorIds))
            revert ISSVNetworkCore.IncorrectValidatorStateWithData(publicKey);

        delete s.validatorPKs[hashedValidator];

        if (cluster.active) {
            StorageProtocol storage sp = SSVStorageProtocol.load();
            (uint64 clusterIndex, ) = OperatorLib.updateClusterOperators(operatorIds, false, false, 1,
            ↪  s, sp);

            cluster.updateClusterData(clusterIndex, sp.currentNetworkFeeIndex());

            sp.updateDAO(false, 1);
        }

        --cluster.validatorCount;

        s.clusters[hashedCluster] = cluster.hashClusterData();

        emit ValidatorRemoved(msg.sender, operatorIds, publicKey, cluster);
    }
```

Impact: An operator requesting a deactivation after initial deposit cannot be reshared.

Recommended Mitigation: Consider either of the 2 options:

- If resharing at PENDING stage needs to be supported, then register operators in `depositStake`
- If resharing at PENDING stage should not be supported, disallow resharing for validators in `Status.PENDING` in the `reshareValidator`

Casimir: Fixed in cd03c74

Cyfrin: Verified.

7.4.3 Missing implementation for EigenPod `withdrawNonBeaconChainETHBalanceWei` in CasimirManager

Description: EigenLayer has a function `EigenPod::withdrawNonBeaconChainETHBalanceWei` that is intended to be called by the pod owner to sweep any ETH donated to EigenPod. Currently, there seems to be no way to withdraw this balance from EigenPod.

Impact: Donations to EigenPod are essentially stuck while the pod is active.

Recommended Mitigation: Consider adding a function to `CasimirManager` that sweeps the `nonBeaconChainETH` balance and sends it to `distributeStakes`, similar to `CasimirManager::claimTips`.

Casimir: Fixed in 790817a

Cyfrin: Verified.

7.4.4 Function `withdrawRewards()` may lead to inaccuracy in `delayedRewards` if there's no withdrawal to process

Description: In the `CasimirManager`, the `withdrawRewards()` function can be used by the reporter to process swept validator rewards. The reporter must provide `WithdrawalProofs`, which the function uses to call `eigenPod.verifyAndProcessWithdrawals()`.

```
function withdrawRewards(WithdrawalProofs memory proofs) external {
    onlyReporter();
```

```
    eigenPod.verifyAndProcessWithdrawals(
        proofs.oracleTimestamp,
        proofs.stateRootProof,
        proofs.withdrawalProofs,
        proofs.validatorFieldsProofs,
        proofs.validatorFields,
        proofs.withdrawalFields
    );

    // @audit Not check if the delayed withdrawal length has changed or not
    uint256 delayedAmount = eigenWithdrawals.userDelayedWithdrawalByIndex(
        address(this), eigenWithdrawals.userWithdrawalsLength(address(this)) - 1
    ).amount;
    delayedRewards += delayedAmount;

    emit RewardsDelayed(delayedAmount);
}
```

The `verifyAndProcessWithdrawals()` function processes a list of withdrawals and sends them as one withdrawal to the delayed withdrawal router. However, it only creates a new withdrawal in the delayed router if the sum of the amount to send is non-zero.

```
if (withdrawalSummary.amountToSendGwei != 0) {
    _sendETH_AsDelayedWithdrawal(podOwner, withdrawalSummary.amountToSendGwei * GWEI_TO_WEI);
}
```

So, if the reporter calls `withdrawRewards()` with no withdrawals, i.e., empty `withdrawalFields` and `validatorFields`, the delayed withdrawal router will not create a new entry. However, as `withdrawRewards()` always takes `delayedAmount` as the latest entry from the delayed withdrawal router, it actually retrieves an old amount that has already been accounted for.

Impact: If the reporter mistakenly calls `withdrawRewards()` with no withdrawals, `delayedRewards` will account for the previous delayed amount again, leading to incorrect accounting.

Recommended Mitigation: Consider following the pattern in the `withdrawValidator()` function. It checks if the length of `eigenWithdrawals.userWithdrawalsLength()` changes before adding the amount to `delayedRewards`.

```
uint256 initialDelayedRewardsLength = eigenWithdrawals.userWithdrawalsLength(address(this));
uint64 initialDelayedEffectiveBalanceGwei = eigenPod.withdrawableRestakedExecutionLayerGwei();

eigenPod.verifyAndProcessWithdrawals(
    ...
);

{
    uint256 updatedDelayedRewardsLength = eigenWithdrawals.userWithdrawalsLength(address(this));
    if (updatedDelayedRewardsLength > initialDelayedRewardsLength) {
        IDelayedWithdrawalRouter.DelayedWithdrawal memory withdrawal =
            eigenWithdrawals.userDelayedWithdrawalByIndex(address(this), updatedDelayedRewardsLength -
            ↪  1);
        if (withdrawal.blockCreated == block.number) {
            delayedRewards += withdrawal.amount;

            emit RewardsDelayed(withdrawal.amount);
        }
    }
}
```

Casimir: Fixed in 81cb7f1

Cyfrin: Verified.

7.4.5 Incorrect test setup leads to false test outcomes

Description: `IntegrationTest.t.sol` includes an integrated test that verifies the entire staking lifecycle. However, the current test setup, in several places, advances the blocks using foundry's `vm.roll` but neglects to adjust the timestamp using `vm.warp`.

This allows the test setup to claim rewards without any time delay.

IntegrationTest.t.sol Line 151

```
>       vm.roll(block.number + eigenWithdrawals.withdrawalDelayBlocks() + 1); //@audit changing block
↪  without changing timestamp
        vm.prank(reporterAddress);
>       manager.claimRewards(); //@audit claiming at the same timestamp

        // Reporter runs after the heartbeat duration
        vm.warp(block.timestamp + 24 hours);
        timeMachine.setProofGenStartTime(0.5 hours);
        beaconChain.setNextTimestamp(timeMachine.proofGenStartTime());
        vm.startPrank(reporterAddress);
        manager.startReport();
        manager.syncValidators(abi.encode(beaconChain.getActiveBalanceSum(), 0));
        manager.finalizeReport();
        vm.stopPrank();
```

Moving blocks without updating the timestamp is an unrealistic simulation of the blockchain. As of EigenLayer M2, `WithdrawDelayBlocks` are 50400, which is approximately 7 days. By advancing 50400 blocks without changing the timestamp, tests overlook several accounting edge cases related to delayed rewards. This is especially true because each reporting period lasts for 24 hours - this means there are 7 reporting periods before a pending reward can actually be claimed.

Impact: An incorrect setup can provide false assurance to the protocol that all edge cases are covered.

Recommended Mitigation: Consider modifying the test setup as follows:

- Run reports without instantly claiming rewards. This accurately reflects events on the real blockchain.
- Consider adjusting time whenever blocks are advanced.

Casimir: Fixed in 290d8e1

Cyfrin: Verified.

7.5 Informational

7.5.1 Missing validations when initializing CasimirRegistry

Description: During Beacon Proxy deployment, CasimirRegistry can be deployed with 0 cluster size and 0 collateral.

Recommended Mitigation: Consider validating inputs for cluster size and collateral.

Casimir: Mitigated in 37f3d34.

Cyfrin: Verified.

7.5.2 ReentrancyGuardUpgradeable is not used in CasimirFactory and CasimirRegistry

Description: CasimirRegistry and CasimirFactory inherit ReentrancyGuardUpgradeable but nonRentrant modifier is unused in bopth contracts.

Recommended Mitigation: Considering removing `ReentrancyGuardUpgradeable` inheritance in `CasimirRegistry` and `CasimirFactory`

Casimir Fixed in e403b8b

Cyfrin Verified.

7.5.3 The period check in `getNextUnstake()` always returns true

Description: The function `getNextUnstake()` is used to get the next unstake request in the queue, while also verifying if the request can be fulfilled. One of the condition to make the request fulfillable is `unstake.period <= reportPeriod`.

```
function getNextUnstake() public view returns (Unstake memory unstake, bool fulfillable) {
    if (unstakeQueue.length > 0) {
        unstake = unstakeQueue[0];
        fulfillable = unstake.period <= reportPeriod && unstake.amount <= getWithdrawableBalance();
    }
}
```

However, given the current codebase, the `unstake.period` will always less or equal to `reportPeriod`. This is because the “unstake.period`will be assigned with`reportPeriod‘ when the unstake request is created/queued but the value of ‘reportPeriod‘ is intended to be only increasing overtime.

```
unstakeQueue.push(Unstake({userAddress: msg.sender, amount: amount, period: reportPeriod}));
...
reportPeriod++;
```

Impact: The check `unstake.period <= reportPeriod` has no effect since it always returns true.

Recommended Mitigation: Consider reviewing the logic in function `getNextUnstake()` and removing the period check if it is unnecessary.

Casimir: Mitigated in 28baa81.

Cyfrin: Verified.

7.5.4 Unused function `validateWithdrawalCredentials()`

Description: The function `validateWithdrawalCredentials()` in CasimirManager is private and isn’t called anywhere in the contract.

```
function validateWithdrawalCredentials(address withdrawalAddress, bytes memory withdrawalCredentials)
↪  // @audit never used
    private
    pure
{
    bytes memory computedWithdrawalCredentials = abi.encodePacked(bytes1(uint8(1)), bytes11(0),
    ↪  withdrawalAddress);
    if (keccak256(computedWithdrawalCredentials) != keccak256(withdrawalCredentials)) {
        revert InvalidWithdrawalCredentials();
    }
}
```

Recommended Mitigation: Consider removing the unused function.

Casimir: Fixed in d6bd8da.

Cyfrin: Verified.

7.5.5 `getUserStake` function failure for non-staker accounts

Description: The function `getUserStake` in the smart contract throws an error when invoked for an address that does not have any stakes. Specifically, the function fails due to a division by zero error. This occurs because the divisor, `users[userAddress].rewardStakeRatioSum0`, can be zero if `userAddress` has never staked, leading to an unhandled exception in the `Math.mulDiv` operation.

```
function getUserStake(address userAddress) public view returns (uint256 userStake) {
    userStake = Math.mulDiv(users[userAddress].stake0, rewardStakeRatioSum,
    ↪  users[userAddress].rewardStakeRatioSum0);
}
```

Recommended Mitigation: Consider modifying the `getUserStake` function to include a check for a zero divisor before performing the division. If `users[userAddress].rewardStakeRatioSum0` is zero, the function should return a stake of 0 to avoid the division by zero error.

Casimir: Fixed in 27c09f5

Cyfrin: Verified.