EigenLayer Contest
Findings & Analysis Report

2023-07-06

Table of contents

- Overview
    - About C4
    - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (2)
    - [H-01] Slot and block number proofs not required for verification of withdrawal (multiple withdrawals possible)
    - [H-02] It is impossible to slash queued withdrawals that contain a malicious strategy due to a misplacement of the ++i increment
- Medium Risk Findings (2)
    - [M-01] A staker with verified over-commitment can potentially bypass slashing completely
    - [M-02] A malicious strategy can permanently DoS all currently pending withdrawals that contain it
- Low Risk and Non-Critical Issues

    - [L-01] `computePhase0Eth1DataRoot` always returns an incorrect Merkle tree
    - [L-02] `processInclusionProofKeccak` does not work as expected
    - [L-03] `merkleizeSha256` doesn't work as expected
    - [L-04] `claimableUserDelayedWithdrawals` sometimes returns unclaimable `DelayedWithdrawals`, so users will see incorrect data
    - [L-05] The condition for full withdrawals in the code is different from that in the documentation
    - [L-06] Missing validation to a threshold value on full withdrawal
    - [L-07] User can stake twice on beacon chain from same eipod, thus losing funds due to same withdrawal credentials
- Gas Optimizations
    - G-01 Optimize `merkleizeSha256` function for gas-efficiency
    - G-02 Use unchecked arithmetic in `processInclusionProofKeccak` and `processInclusionProofSha256` functions
- Disclosures

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the EigenLayer smart contract system written in Solidity. The audit took place between April 27 - May 4 2023.

Wardens

48 Wardens contributed reports to the EigenLayer Audit:

1. 0xSmartContract
2. 0xTheC0der
3. 0xWaitress
4. 0xnev
5. 8olidity
6. ABA
7. Aymen0909
8. Co0nan
9. Cyfrin (PatrickAlphaC, giovannidisiena, hansfriese, 0Kage, alexroan and carlitox477)
10. Dug
11. Haipls
12. Josiah
13. MiloTruck
14. QiuhaoLi
15. RaymondFam
16. ReyAdmirado
17. Ruhum
18. SpicyMeatball
19. ToonVH
20. bin2chen
21. btk
22. bughunter007
23. bytes032
24. clayj
25. d3e4
26. evmboi32
27. ihtishamsudo

28. itsmeSTYJ
29. jasonxiale
30. juancito
31. libratus
32. naman1778
33. neutiyoo
34. niser93
35. pontifex
36. rvierdiiev
37. said
38. sashik_eth
39. tonisives
40. turvy_fuzz
41. volodya
42. windowhan001
43. yjrwkk

This audit was judged by Alex the Entreprenerd.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 2 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 15 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 13 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 EigenLayer Audit repository, and is composed of 24 smart contracts written in the Solidity programming language and includes 1393 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (2)

[H-01] Slot and block number proofs not required for verification of withdrawal (multiple withdrawals possible)

Submitted by 0xTheC0der, also found by windowhan001 and volodya

Since this is a vulnerability that involves multiple in-scope contracts and leads to more than one impact, let's start with a bug description from bottom to top.

Library `Merkle`

The methods verifyInclusionSha256(proof, root, leaf, index) and verifyInclusionKeccak(proof, root, leaf, index) will **always** return `true` if `proof.length < 32` (e.g. empty proof) **and** `leaf == root`. Although this might be intended behaviour, I see no use case for empty proofs and would `require` non-empty proofs at the library level. As of now, the user of the library is **responsible** to enforce non-zero proofs.

Library `BeaconChainProofs`

The method verifyWithdrawalProofs(beaconStateRoot, proofs, withdrawalFields), which relies on multiple calls to Merkle.verifyInclusionSha256(proof, root, leaf, index), does not `require` a minimum length of `proofs.slotProof` and `proofs.blockNumberProof`. As a consequence, considering a valid set of `(beaconStateRoot, proofs, withdrawalFields)`, the method will still succeed with **empty** slot and block number proofs, i.e. the `proofs` can be modified in the following way:

```
    proofs.slotProof = bytes("");                // empty slot proof
    proofs.slotRoot = proofs.blockHeaderRoot; // make leaf == root

    proofs.blockNumberProof = bytes("");                    // empty b
    proofs.blockNumberRoot = proofs.executionPayloadRoot; // make le
```

As a consequence, we can take a perfectly valid withdrawal proof and re-create the proof for the same withdrawal with a **different** slot and block number (according to the code above) that will still be accepted by the verifyWithdrawalProofs(beaconStateRoot, proofs, withdrawalFields) method.

Contract `EigenPod`

The method verifyAndProcessWithdrawal(withdrawalProofs, ...), which relies on a call to BeaconChainProofs.verifyWithdrawalProofs(beaconStateRoot, proofs, withdrawalFields), is impacted by a modified - but still valid - withdrawal proof in two ways.

First, the modifier proofIsForValidBlockNumber(Endian.fromLittleEndianUint64(withdrawalProofs.blockNumberRoot)) makes sure that the **block number** being proven is greater/newer than the `mostRecentWithdrawalBlockNumber`. In our case, `blockNumberRoot = executionPayloadRoot` and depending on the actual value of `executionPayloadRoot`, the `proofIsForValidBlockNumber` can be bypassed as shown in the test, see any PoC test case. As a consequence, old withdrawal proofs could be re-used with an empty `blockNumberProof` to withdraw the same funds more than once.

Second, the sub-method _processPartialWithdrawal(withdrawalHappenedSlot, ...) requires that a **slot** is only used once. In our case, `slotRoot = blockHeaderRoot` leads to a **different slot** than suggested by the original proof. Therefore, a withdrawal proof can be re-used with an empty `slotProof` to do the same partial withdrawal twice, see PoC. Depending on the actual value of `blockHeaderRoot`, a full withdrawal, instead of a partial withdrawal, will be done according to the **condition in L354**.

Insufficient validation of proofs allows multiple withdrawals, i.e. theft of funds.

Proof of Concept

The changes to the `EigenPod` test cases below demonstrate the following outcomes:
testFullWithdrawalProof:
BeaconChainProofs.verifyWithdrawalProofs(beaconStateRoot, proofs, withdrawalFields) still succeeds on empty slot and block number proofs.
testFullWithdrawalFlow: EigenPod.verifyAndProcessWithdrawal(withdrawalProofs, ...) allows full withdrawal with empty slot and block number proofs.
testPartialWithdrawalFlow:
EigenPod.verifyAndProcessWithdrawal(withdrawalProofs, ...) allows partial withdrawal with empty slot and block number proofs.
testProvingMultipleWithdrawalsForSameSlot:
EigenPod.verifyAndProcessWithdrawal(withdrawalProofs, ...) allows partial withdrawal of the same funds twice due to different `slotRoot` in original and modified proof.

The
proofIsForValidBlockNumber(Endian.fromLittleEndianUint64(withdrawalProofs.blockNumberRoot)) modifier is bypassed (see `blockNumberRoot`) in the latter three of the above test cases.

Apply the following *diff* to your `src/test/EigenPod.t.sol` and run the tests with `forge test --match-contract EigenPod`:

```
diff --git a/src/test/EigenPod.t.sol b/src/test/EigenPod.t.sol
index 31e6a58..5242def 100644
--- a/src/test/EigenPod.t.sol
+++ b/src/test/EigenPod.t.sol
@@ -260,7 +260,7 @@ contract EigenPodTests is ProofParsing, Eige

     function testFullWithdrawalProof() public {
         setJSON("./src/test/test-data/fullWithdrawalProof.json'
-        BeaconChainProofs.WithdrawalProofs memory proofs = _get
+        BeaconChainProofs.WithdrawalProofs memory proofs = _get
         withdrawalFields = getWithdrawalFields();
         validatorFields = getValidatorFields();

@@ -281,7 +281,7 @@ contract EigenPodTests is ProofParsing, Eige

         // ./solidityProofGen "WithdrawalFieldsProof" 61336 22
         setJSON("./src/test/test-data/fullWithdrawalProof.json'
-        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
+        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
         bytes memory validatorFieldsProof = abi.encodePacked(ge
         withdrawalFields = getWithdrawalFields();
         validatorFields = getValidatorFields();
@@ -317,7 +317,7 @@ contract EigenPodTests is ProofParsing, Eige
         //generate partialWithdrawalProofs.json with:
         // ./solidityProofGen "WithdrawalFieldsProof" 61068 65
         setJSON("./src/test/test-data/partialWithdrawalProof.js
-        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
+        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
         bytes memory validatorFieldsProof = abi.encodePacked(ge

         withdrawalFields = getWithdrawalFields();
@@ -346,21 +346,22 @@ contract EigenPodTests is ProofParsing, Ei

     /// @notice verifies that multiple partial withdrawals can
     function testProvingMultipleWithdrawalsForSameSlot(/*uint2
-        IEigenPod newPod = testPartialWithdrawalFlow();
+        IEigenPod newPod = testPartialWithdrawalFlow(); // uses

-        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
+        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
         bytes memory validatorFieldsProof = abi.encodePacked(ge
         withdrawalFields = getWithdrawalFields();
         validatorFields = getValidatorFields();

-        cheats.expectRevert(bytes("EigenPod._processPartialWit
```

```
+        // do not expect revert anymore due to different 'slotP
+        //cheats.expectRevert(bytes("EigenPod._processPartialWi
         newPod.verifyAndProcessWithdrawal(withdrawalProofs, val
     }

     /// @notice verifies that multiple full withdrawals for a s
     function testDoubleFullWithdrawal() public {
-        IEigenPod newPod = testFullWithdrawalFlow();
-        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
+        IEigenPod newPod = testFullWithdrawalFlow(); // uses SK
+        BeaconChainProofs.WithdrawalProofs memory withdrawalPro
         bytes memory validatorFieldsProof = abi.encodePacked(ge
         withdrawalFields = getWithdrawalFields();
         validatorFields = getValidatorFields();
@@ -759,8 +760,11 @@ contract EigenPodTests is ProofParsing, Eig
         return proofs;
     }

+    uint256 internal constant FULL_PROOF = 0;
+    uint256 internal constant SKIP_SLOT_BLOCK_PROOF = 1;
+
     /// @notice this function just generates a valid proof so t
-    function _getWithdrawalProof() internal returns(BeaconChain
+    function _getWithdrawalProof(uint256 proofType) internal re
         //make initial deposit
         cheats.startPrank(podOwner);
         eigenPodManager.stake{value: stakeAmount}(pubkey, signa
@@ -773,9 +777,9 @@ contract EigenPodTests is ProofParsing, Eige
             beaconChainOracle.setBeaconChainStateRoot(beaconSta
             bytes32 blockHeaderRoot = getBlockHeaderRoot();
             bytes32 blockBodyRoot = getBlockBodyRoot();
-            bytes32 slotRoot = getSlotRoot();
-            bytes32 blockNumberRoot = getBlockNumberRoot();
+            bytes32 slotRoot = (proofType == FULL_PROOF) ? getS
             bytes32 executionPayloadRoot = getExecutionPayloadR
+            bytes32 blockNumberRoot = (proofType == FULL_PROOF)



@@ -786,9 +790,9 @@ contract EigenPodTests is ProofParsing, Eige
             BeaconChainProofs.WithdrawalProofs memory proofs =
                 abi.encodePacked(getBlockHeaderProof()),
                 abi.encodePacked(getWithdrawalProof()),
-               abi.encodePacked(getSlotProof()),
+               (proofType == FULL_PROOF) ? abi.encodePacked(ge
                abi.encodePacked(getExecutionPayloadProof()),
```

```
-                   abi.encodePacked(getBlockNumberProof()),
+                   (proofType == FULL_PROOF) ? abi.encodePacked(ge
                    uint64(blockHeaderRootIndex),
                    uint64(withdrawalIndex),
                    blockHeaderRoot,
```

We can see that **all** the test cases are still passing, whereby the following ones are confirming the aforementioned outcomes:

```
[PASS] testFullWithdrawalFlow():(address) (gas: 28517915)
[PASS] testFullWithdrawalProof() (gas: 13185538)
[PASS] testPartialWithdrawalFlow():(address) (gas: 28679149)
[PASS] testProvingMultipleWithdrawalsForSameSlot() (gas: 4550228
```

Tools Used

VS Code, Foundry

Recommended Mitigation Steps

Require a minimum length (tree height) for the slot and block number proofs in [BeaconChainProofs.verifyWithdrawalProofs(beaconStateRoot, proofs, withdrawalFields)].

At least require non-empty proofs according to the following *diff*:

```
diff --git a/src/contracts/libraries/BeaconChainProofs.sol b/src
index b4129bf..119baf2 100644
--- a/src/contracts/libraries/BeaconChainProofs.sol
+++ b/src/contracts/libraries/BeaconChainProofs.sol
@@ -259,6 +259,10 @@ library BeaconChainProofs {
             "BeaconChainProofs.verifyWithdrawalProofs: withdraw
         require(proofs.executionPayloadProof.length == 32 * (BE
             "BeaconChainProofs.verifyWithdrawalProofs: executio
+        require(proofs.slotProof.length >= 32,
+            "BeaconChainProofs.verifyWithdrawalProofs: slotProo
+        require(proofs.blockNumberProof.length >= 32,
+            "BeaconChainProofs.verifyWithdrawalProofs: blockNum

         /**
```

```
        * Computes the block_header_index relative to the beac
```

Alternative: Non-empty proofs can also be required in the `Merkle` library.

Assessed type

Invalid Validation

sorrynotsorry (lookout) commented:

> Well demonstrated with referrable code snippets, hyperlinks, and coded POC. Marking as HQ.

Sidu28 (EigenLayer) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to a lack of length check, an empty proof could be provided; which would pass validation.

> This is an example of how a lack of a check can be chained into a proper exploit, and because the proof will pass, funds can be stolen.

> For these reasons I agree with High Severity.

[H-02] It is impossible to slash queued withdrawals that contain a malicious strategy due to a misplacement of the ++i increment

Submitted by juancito, *also found by* yjrwkk, pontifex, evmboi32, bin2chen, sashik_eth, Ruhum, MiloTruck, SpicyMeatball, *and* volodya.

`StrategyManager::slashQueuedWithdrawal()` contains an `indicesToSkip` parameter to skip malicious strategies, as documented in the function definition:

> so that, e.g., if the slashed `QueuedWithdrawal` contains a malicious strategy in the `strategies` array which always reverts on calls to its 'withdraw' function,

> then the malicious strategy can be skipped (with the shares in effect “burned”), while the non-malicious strategies are still called as normal.

The problem is, the function does not work as expected, and `indicesToSkip` is ignored. If the queued withdrawal contains a malicious strategy, it will make the slash always revert.

Owners won’t be able to slash queued withdrawals that contain a malicious strategy.

An adversary can take advantage of this and create withdrawal queues that won’t be able to be slashed, completely defeating the slash system. The adversary can later complete the withdrawal.

Proof of Concept

The `++i;` statement in `StrategyManager::slashQueuedWithdrawal()` is misplaced. It is only executed on the `else` statement:

```
        // keeps track of the index in the `indicesToSkip` array
        uint256 indicesToSkipIndex = 0;

        uint256 strategiesLength = queuedWithdrawal.strategies.lengt
        for (uint256 i = 0; i < strategiesLength;) {
            // check if the index i matches one of the indices speci
            if (indicesToSkipIndex < indicesToSkip.length && indices
                unchecked {
                    ++indicesToSkipIndex;
                }
            } else {
                if (queuedWithdrawal.strategies[i] == beaconChainETH
                        //withdraw the beaconChainETH to the recipie
                    _withdrawBeaconChainETH(queuedWithdrawal.deposit
                } else {
                    // tell the strategy to send the appropriate amo
                    queuedWithdrawal.strategies[i].withdraw(recipien
                }
                unchecked {
                    ++i; // @audit
                }
            }
        }
```

Link to code

Let's suppose that the owner tries to slash a queued withdrawal, and wants to skip the first strategy (index `0`) because it is malicious and makes the whole transaction revert.

1 . It defines `indicesToSkipIndex = 0` .
2 . It enters the `for` loop starting at `i = 0` .
3 . `if (indicesToSkipIndex < indicesToSkip.length && indicesToSkip[indicesToSkipIndex] == i)` will be true: `0 < 1 && 0 == 0` .
4 . It increments `++indicesToSkipIndex;` to "skip" the malicious strategy, so now `indicesToSkipIndex = 1` .
5 . It goes back to the `for` loop. But `i` hasn't been modified, so still `i = 0` .
6 . `if (indicesToSkipIndex < indicesToSkip.length && indicesToSkip[indicesToSkipIndex] == i)` will be false now: `1 < 1 && 0 == 0` .
7 . It will enter the `else` statement and attempt to slash the strategy anyway.
8 . If the strategy is malicious, it will revert, making it impossible to slash.
9 . The adversary can later complete the withdrawal.

POC Test

This test shows how the `indicesToSkip` parameter is completely ignored.

For the sake of simplicity of the test, it uses a normal strategy; which will be slashed, proving that it ignores the `indicesToSkip` parameter and it indeed calls `queuedWithdrawal.strategies[i].withdraw()` .

A malicious strategy that makes `withdraw()` revert, would be to make the whole transaction revert (not shown on this test but easily checkable as the function won't catch it).

Add this test to `src/tests/StrategyManagerUnit.t.sol` and run `forge test -m "testSlashQueuedWithdrawal_IgnoresIndicesToSkip"` .

```
    function testSlashQueuedWithdrawal_IgnoresIndicesToSkip() ex
        address recipient = address(this);
```

```
    uint256 depositAmount = 1e18;
    uint256 withdrawalAmount = depositAmount;
    bool undelegateIfPossible = false;

    // Deposit into strategy and queue a withdrawal
    (IStrategyManager.QueuedWithdrawal memory queuedWithdraw
        testQueueWithdrawal_ToSelf_NotBeaconChainETH(deposit

    // Slash the delegatedOperator
    slasherMock.freezeOperator(queuedWithdrawal.delegatedAdd

    // Keep track of the balance before the slash attempt
    uint256 balanceBefore = dummyToken.balanceOf(address(rec

    // Assert that the strategies array only has one element
    assertEq(queuedWithdrawal.strategies.length, 1);

    // Set `indicesToSkip` so that it should ignore the only
    // As it's the only element, its index is `0`
    uint256[] memory indicesToSkip = new uint256[](1);
    indicesToSkip[0] = 0;

    // Call `slashQueuedWithdrawal()`
    // This should not try to slash the only strategy the qu
    // But in fact it ignores `indicesToSkip` and attempts t
    cheats.startPrank(strategyManager.owner());
    strategyManager.slashQueuedWithdrawal(recipient, queuedW
    cheats.stopPrank();

    uint256 balanceAfter = dummyToken.balanceOf(address(reci

    // The `indicesToSkip` was completely ignored, and the f
    // It can be asserted due to the fact that it increased
    require(balanceAfter == balanceBefore + withdrawalAmount
}
```

Recommended Mitigation Steps

Place the `++i` outside of the if/else statement. This way, it will increment each time the loop runs.

```
for (uint256 i = 0; i < strategiesLength;) {
    // check if the index i matches one of the indices speci
    if (indicesToSkipIndex < indicesToSkip.length && indices
```

```diff
                unchecked {
                    ++indicesToSkipIndex;
                }
            } else {
                if (queuedWithdrawal.strategies[i] == beaconChainETH
                        //withdraw the beaconChainETH to the recipie
                    _withdrawBeaconChainETH(queuedWithdrawal.deposit
                } else {
                    // tell the strategy to send the appropriate amo
                    queuedWithdrawal.strategies[i].withdraw(recipier
                }
-               unchecked {
-                   ++i;
-               }
            }
+           unchecked {
+               ++i;
+           }
        }
```

Assessed type

Loop

sorrynotsorry (lookout) commented:

> The issue is well demonstrated, properly formatted, and contains a coded POC. Marking as HQ.

Sidu28 (EigenLayer) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to incorrect placement of the loop increment, malicious strategies cannot be skipped when slashing queued withdrawals.

> Because this breaks a core functionality of the contracts, which will also cause a loss of funds, I agree with High Severity.

> Mitigation is straightforward.

Medium Risk Findings (2)

[M-01] A staker with verified over-commitment can potentially bypass slashing completely

Submitted by Cyfrin, also found by Josiah, QiuhaoLi, and RaymondFam

https://github.com/code-423n4/2023-04-eigenlayer/blob/5e4872358cd2bda1936c29f460ece2308af4def6/src/contracts/core/StrategyManager.sol#L197
https://github.com/code-423n4/2023-04-eigenlayer/blob/5e4872358cd2bda1936c29f460ece2308af4def6/src/contracts/core/StrategyManager.sol#L513

In EigenLayer, watchers submit over-commitment proof in the event a staker's balance on the Beacon chain falls below the minimum restaked amount per validator. In such a scenario, stakers' shares are decreased by the restaked amount. Note, that when a full withdrawal is processed, stakers' deducted shares are credited back to allow for a planned withdrawal on EigenLayer

If such a staker has delegated to an operator who gets slashed on EigenLayer, there is a possibility that this staker completely bypasses any slashing penalties. If overcommitment reduced the shares in the stakers account to 0, there is nothing available for governance to slash.

It is reasonable to assume that governance calls `StrategyManager::slashShares` to reduce a percentage of shares (penalty) from all stakers who delegated to a slashed operator and then resets the frozen status of operator by calling `ISlasher::resetFrozenStatus`.

By simply unstaking on the beacon chain AFTER slashing is complete (and operator is unfrozen), an over-committed staker can simply unstake on beacon chain and get back their shares that were deducted when over-commitment was recorded (refer to PoC). Note: these shares have not faced any slashing penalties.

An over-committed staker can avoid being slashed in a scenario, in which their stake should be subject to slashing and so we evaluate the severity to **MEDIUM**.

Proof of Concept

Consider the following scenario with a chain of events in this order:

1. Alice stakes 32 ETH and gets corresponding shares in `BeaconEthStrategy`.
2. After some time, Alice gets slashed on the Beacon Chain and her current balance on the Beacon Chain is now less than what she restaked on EigenLayer.
3. An observer will submit a proof via `EigenPod::verifyOverCommittedStake` and Alice's shares are now decreased to 0 (Note: this will be credited back to Alice when she withdraws from the Beacon Chain using `EigenPod::verifyAndProcessWithdrawal`).
4. Next, Alice's operator gets slashed and her account gets frozen.
5. Governance slashes Alice, along with all stakers who delegated to slashed operator (there is nothing to slash since Alice's shares are currently 0).
6. After slashing everyone, governance resets frozen status of operator by calling `ISlasher::resetFrozenStatus`.
7. Alice now unstakes on the Beacon Chain and gets a credit of shares that were earlier deducted while recording over-commitment.
8. Alice queues a withdrawal request and completes the withdrawal, without facing any slashing penalty.

Recommended Mitigation Steps

Over-commitment needs to be accounted for when slashing, such that a staker is being slashed; not just shares in their account, but also the amount that is temporarily debited while recording over-commitment.

Consider adding a mapping to `StrategyManager` that keeps track of over-commitment for each staker and take that into consideration while slashing in `StrategyManager::slashShares`.

[Alex the Entreprenerd (judge) commented]:

> The Warden has shown how, due to an incorrect assumption, it's possible for an over-commited staker to avoid a slashing.

> The finding requires multiple external requirements:

- Being over-commited (perhaps inactivity leak or ETH2 slashing).
- Having delegated to an operator that will be slashed.

> For these reasons I agree with Medium Severity.

Sidu28 (EigenLayer) confirmed via duplicate issue 210

[M-02] A malicious strategy can permanently DoS all currently pending withdrawals that contain it

Submitted by ABA, *also found by* juancito, ToonVH, 0xWaitress, ABA, MiloTruck, rvierdiiev, 8olidity, bughunter007, *and* bytes032.

In order to withdraw funds from the project a user has to:

1. queue a withdrawal (via `queueWithdrawal`).
2. complete a withdrawal (via `completeQueuedWithdrawal(s)`).

Queuing a withdrawal, via `queueWithdrawal` modifies all internal accounting to reflect this:

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L345-L346

```
            // modify delegated shares accordingly, if applicable
            delegation.decreaseDelegatedShares(msg.sender, strategie
```

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L370

```
        if (_removeShares(msg.sender, strategyIndexes[strategyIndex]
```

and saves the withdrawal hash in order to be used in `completeQueuedWithdrawal(s)`.

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L400-L415

```
            queuedWithdrawal = QueuedWithdrawal({
                strategies: strategies,
                shares: shares,
                depositor: msg.sender,
                withdrawerAndNonce: withdrawerAndNonce,
                withdrawalStartBlock: uint32(block.number),
                delegatedAddress: delegatedAddress
            });

        }

        // calculate the withdrawal root
        bytes32 withdrawalRoot = calculateWithdrawalRoot(queuedW

        // mark withdrawal as pending
        withdrawalRootPending[withdrawalRoot] = true;
```

In other words, it is final (as there is no `cancelWithdrawl` mechanism implemented). When executing `completeQueuedWithdrawal(s)`, the `withdraw` function of the strategy is called (if `receiveAsTokens` is set to true).

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L786-L789

```
    // tell the strategy to send the appropriate amount of funds
    queuedWithdrawal.strategies[i].withdraw(
        msg.sender, tokens[i], queuedWithdrawal.shares[i]
    );
```

In this case, a malicious strategy can always revert, blocking the user from retrieving his tokens. If a user sets `receiveAsTokens` to `false`, the other case, then the tokens will be added as shares to the delegation contract https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L797-L800

```
        for (uint256 i = 0; i < strategiesLength;) {
            _addShares(msg.sender, queuedWithdrawal.strategies[i], c
            unchecked {
                ++i;
```

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L646-L647

```
            // if applicable, increase delegated shares accordingly
            delegation.increaseDelegatedShares(depositor, strategy,
```

This still poses a problem because the `increaseDelegatedShares` function's counterpart, `decreaseDelegatedShares`, is only callable by the strategy manager (for users to indirectly get their rewards worth back, via this workaround).

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/DelegationManager.sol#L168-L179

```
        /**
         * @notice Decreases the `staker`'s delegated shares in each
         * @dev Callable only by the StrategyManager
         */
        function decreaseDelegatedShares(
            address staker,
            IStrategy[] calldata strategies,
            uint256[] calldata shares
        )
            external
            onlyStrategyManager
        {
```

But in `StrategyManager`, there are only 3 cases where this is called, none of which is beneficial for the user:

- `recordOvercommittedBeaconChainETH` - slashes user rewards in certain conditions
- `queueWithdrawal` - accounting side effects have already been done, will fail in `_removeShares`
- `slashShares` - slashes user rewards in certain conditions

In other words, the workaround (of setting `receiveAsTokens` to `false` in `completeQueuedWithdrawal(s)`) just leaves the funds accounted and stuck in `DelegationManager`.

In both cases, all shares/tokens associated with other strategies in the pending withdrawal are permanently blocked.

Proof of Concept

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L786-L789

Recommended Mitigation Steps

Add a `cancelQueuedWithdrawl` function that will undo the accounting and cancel out any dependency on the malicious strategy. Although this solution may create a front-running opportunity for when their withdrawal will be slashed via `slashQueuedWithdrawal`, there may exist workarounds to this.

Another possibility is to implement a similar mechanism to how `slashQueuedWithdrawal` treats malicious strategies: adding a list of strategy indices to skip.

https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L532-L535
https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L560-L566

```
     * @param indicesToSkip Optional input parameter -- indices
     * so that, e.g., if the slashed QueuedWithdrawal contains a
     * then the malicious strategy can be skipped (with the shar
     */
```

```
        ...

            for (uint256 i = 0; i < strategiesLength;) {
                // check if the index i matches one of the indices s
                if (indicesToSkipIndex < indicesToSkip.length && inc
                    unchecked {
                        ++indicesToSkipIndex;
                    }
                } else {
```

Assessed type

DoS

[Sidu28 (EigenLayer) disagreed with severity and commented]:

> The description of lack of check here is accurate. There is zero actual impact; we think this is informational severity.

[Alex the Entreprenerd (judge) commented]:

> @Sidu28 - if we assumed the Strategy not to be malicious, but to revert for some reason (e.g. lack of liquidity, smart contract bug, etc...)

> Would you consider the finding as valid since the user cannot withdrawal from all others due to having one withdraw, or is there some other reason why you believe the finding to be Informational?

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]:

> The Warden has shown how, any revert in a strategy (for example it being paused), will make queued withdrawals revert, even if the withdrawal should work for other strategies.

> This falls into the category of DOS and I believe is more appropriately judged as Medium.

[ChaoticWalrus (EigenLayer) commented]:

> @Alex the Entreprenerd - Even if a malicious strategy exists, there is no permanent delay here.

> Users can mark the `receiveAsTokens` input to `completeQueuedWithdrawal` as false here and then the shares will be transferred to the 'withdrawing' user here who can then *queue a new withdrawal not containing the malicious strategy*.

> The description provided by the warden of the `DelegationManager` 's behavior is inaccurate — the funds do not end up 'stuck in the `DelegationManager` ' — they will be withdrawable as part of a new queued withdrawal, which excludes the malicious strategy.

ABA (warden) commented:

> Hey, https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L795-L803 does not do:

> shares will be transferred to the 'withdrawing' user

> It simply increments internal accounting and delegated shares associated to the user in the delegation contract. But these are not withdrawn without a `queueWithdrawal` call.

> https://github.com/code-423n4/2023-04-eigenlayer/blob/main/src/contracts/core/StrategyManager.sol#L647

> *queue a new withdrawal not containing the malicious strategy.*

> A new queue can not reuse the already used balance, as it was already accounted for here.

> Maybe I am missing something of course. Regardless, awaiting judge feedback.

ChaoticWalrus (EigenLayer) commented:

> It simply increments internal accounting and delegated shares associated to the user in the delegation contract. But these are not withdrawn without a

> `queueWithdrawal` call
> Yes, this is what I would describe as transferring the shares, similar to how the 'transfer in' portion of an ERC20 transfer works. But this is a semantic argument; I agree with the substance of your description.

> A new queue can not reuse the already used balance as it was already accounted for [here](#)

> Maybe I am missing something of course. Regardless, awaiting judge feedback.

> I think perhaps you are missing that the internal `_addShares` function calls the `DelegationManager` as well. See [here](#)

> If the user marks `receiveAsTokens` as false, then [this line](#) will be triggered, which calls into the internal `_addShares` function and 're-adds' these delegated shares, so they can indeed be queued in a new withdrawal.

> I can provide an example if it will help.

[ChaoticWalrus (EigenLayer) commented](#):

> Basically, the 're-adding' reverses the accounting taken in the initial `queueWithdrawal` action, to clarify. This then allows the accounting done in the `queueWithdrawal` action to be performed once again when a new withdrawal is queued.

[Alex the Entreprenerd (judge) commented](#):

> @ChaoticWalrus - It seems like the worst case scenario would be having to re-queue without the missing strategy, which would require waiting for the withdrawal to unlock.

> So the question left to answer is whether an additional wait period is acceptable.

> NOTE: I edited this comment because as you pointed out, the shares accounting is internal and doesn't trigger an interaction with the strategy.

[ChaoticWalrus (EigenLayer) commented](#):

> It seems like the worst case scenario would be having to re-queue without the missing strategy which would require waiting for the withdrawal to unlock

> Yes, agreed this is the impact + worst-case scenario. The existing functionality was designed, in part at least, to address concerns about malicious Strategies.

> So the question left to answer is whether an additional wait period is acceptable

> I suppose so, yes. We have deemed this acceptable ourselves, but I could see it being viewed differently. Regardless, the impact here is orders of magnitude less than the original claimed impact.

[Alex the Entreprenerd (judge) commented]():

> The Warden has shown how, due to the possibility of queueing a withdrawal with multiple strategies, in the case in which one of the strategies stops working (reverts, paused, malicious), the withdrawal would be denied.

> As the sponsor said, in those scenarios, the withdrawer would have to perform a second withdrawal, which would have to be re-queued.

> Intuitively, a withdrawer that always withdraws a separate strategy would also never see their withdrawal denied (except for the malicious strategy).

> As we can see, there are plenty of side steps to the risk shown. However, the functionality of the function is denied, even if temporarily, leading to it not working as intended, and for this reason I believe Medium Severity to be the most appropriate.

> As shown above, the finding could be fixed. However, it seems to me like most users would want to plan around the scenarios described in this finding and its duplicates.

Low Risk and Non-Critical Issues

For this audit, 15 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below]() by **volodya** received the top score from the judge.

The following wardens also submitted reports: libratus, Cyfrin, niser93, juancito, btk, QiuhaoLi, Aymen0909, 0xnev, ABA, RaymondFam, sashik_eth, 0xWaitress, ihtishamsudo and bughunter007.

[L-01] `computePhase0Eth1DataRoot` always returns an incorrect Merkle tree

The Merkle tree creation inside the `computePhase0Eth1DataRoot` function is incorrect.

Proof of Concept

Provide direct links to all referenced code in GitHub. Add screenshots, logs, or any other relevant proof that illustrates the concept. Not all fields of `eth1DataFields` are being used in an array due to the usage of `i < ETH1_DATA_FIELD_TREE_HEIGHT` instead of `i<NUM_ETH1_DATA_FIELDS`. Check other similar functions.

src/contracts/libraries/BeaconChainProofs.sol#L160

```
    function computePhase0Eth1DataRoot(bytes32[NUM_ETH1_DATA_FIE
        bytes32[] memory paddedEth1DataFields = new bytes32[](2*

        for (uint256 i = 0; i < ETH1_DATA_FIELD_TREE_HEIGHT; ++i
            paddedEth1DataFields[i] = eth1DataFields[i];
        }

        return Merkle.merkleizeSha256(paddedEth1DataFields);
    }
```

Recommended Mitigation Steps

```
    function computePhase0Eth1DataRoot(bytes32[NUM_ETH1_DATA_FIE
        bytes32[] memory paddedEth1DataFields = new bytes32[](2*

-        for (uint256 i = 0; i < ETH1_DATA_FIELD_TREE_HEIGHT; ++
+        for (uint256 i = 0; i < NUM_ETH1_DATA_FIELDS; ++i) {
            paddedEth1DataFields[i] = eth1DataFields[i];
```

```
        }

        return Merkle.merkleizeSha256(paddedEth1DataFields);
    }
```

Assessed type

Math

[Sidu28 (EigenLayer) disagreed with severity and commented]:

> We believe this is low severity. The code is unused and informally deprecated, but it is indeed technically incorrect.

[Alex the Entreprenerd (judge) decreased severity to QA and commented]:

> Agree with the Sponsor, because the code is unused.

[Alex the Entreprenerd (judge) commented]:

> Consistently high quality submissions. After grading the QAs I believe the Warden deserves the best place.

[L-02] `processInclusionProofKeccak` does not work as expected

Proof of Concept

The function `verifyInclusionKeccak` is not used anywhere but its in the scope of this audit. There is no validation that proof is a tree and a valid tree like it described in the comments. E.x. if proof is less than 32 length, that function will just return a leaf without reverting. In my opinion, function doesn't work as expected and can be exploited. I've submitted the same issue with `processInclusionProofSha256` function that lead to loss a funds for validator due the same issue.

```
    function processInclusionProofKeccak(bytes memory proof, byt
        bytes32 computedHash = leaf;
```

```
        for (uint256 i = 32; i <= proof.length; i+=32) {
            if(index % 2 == 0) {
                // if ith bit of index is 0, then computedHash i
                assembly {
                    mstore(0x00, computedHash)
                    mstore(0x20, mload(add(proof, i)))
                    computedHash := keccak256(0x00, 0x40)
                    index := div(index, 2)
                }
            } else {
                // if ith bit of index is 1, then computedHash i
                assembly {
                    mstore(0x00, mload(add(proof, i)))
                    mstore(0x20, computedHash)
                    computedHash := keccak256(0x00, 0x40)
                    index := div(index, 2)
                }
            }
        }
        return computedHash;
    }
```

src/contracts/libraries/Merkle.sol#L49

Recommended Mitigation Steps

I think its important to add security to that function like this:

```
    function processInclusionProofKeccak(bytes memory proof, byt
+        require(proof.length % 32 == 0 && proof.length > 0, "Ir

        bytes32 computedHash = leaf;
        for (uint256 i = 32; i <= proof.length; i+=32) {
            if(index % 2 == 0) {
                // if ith bit of index is 0, then computedHash i
                assembly {
                    mstore(0x00, computedHash)
                    mstore(0x20, mload(add(proof, i)))
                    computedHash := keccak256(0x00, 0x40)
                    index := div(index, 2)
                }
            } else {
                // if ith bit of index is 1, then computedHash i
```

```
                assembly {
                    mstore(0x00, mload(add(proof, i)))
                    mstore(0x20, computedHash)
                    computedHash := keccak256(0x00, 0x40)
                    index := div(index, 2)
                }
            }
        }
        return computedHash;
    }
```

[Sidu28 (EigenLayer) confirmed]

[Alex the Entreprenerd (judge) decreased severity to QA and commented]:

> https://github.com/code-423n4/2023-04-eigenlayer/blob/398cc428541b91948f717482ec973583c9e76232/src/contracts/operators/MerkleDelegationTerms.sol#L97-L107

```
        // check inclusion of the leafHash in the tree correspor
        require(
            Merkle.verifyInclusionKeccak(
                proof,
                merkleRoots[rootIndex].root,
                leafHash,
                nodeIndex
            ),
            "MerkleDelegationTerms.proveEarningsAndWithdraw: prc
        );
```

> Which calls `processInclusionProofKeccak`

> For this reason, I believe the finding to be a Refactoring. Adding the check in the function is a good idea, but the code in scope is safe.

[L-03] `merkleizeSha256` doesn't work as expected

https://github.com/code-423n4/2023-04-eigenlayer/blob/398cc428541b91948f717482ec973583c9e76232/src/contracts/li

[braries/Merkle.sol#L129]

Proof of Concept

Whenever `merkleizeSha256` is being used in the code, there is always a check that array length is power of 2. E.x.:

```
bytes32[] memory paddedHeaderFields = new bytes32[](2**BEACON_BI
```

[contracts/libraries/BeaconChainProofs.sol#L131]

But inside the function `merkleizeSha256`, there is no check that incoming array is power of 2.

```
    /**
      @notice this function returns the merkle root of a tree cre
      @param leaves the leaves of the merkle tree

      @notice requires the leaves.length is a power of 2
      */
     function merkleizeSha256(
         bytes32[] memory leaves
     ) internal pure returns (bytes32) {
         //there are half as many nodes in the layer above the le
         uint256 numNodesInLayer = leaves.length / 2;
         //create a layer to store the internal nodes
         bytes32[] memory layer = new bytes32[](numNodesInLayer);
         //fill the layer with the pairwise hashes of the leaves
         for (uint i = 0; i < numNodesInLayer; i++) {
             layer[i] = sha256(abi.encodePacked(leaves[2*i], leav
         }
         //the next layer above has half as many nodes
         numNodesInLayer /= 2;
         //while we haven't computed the root
         while (numNodesInLayer != 0) {
             //overwrite the first numNodesInLayer nodes in layer
             for (uint i = 0; i < numNodesInLayer; i++) {
                 layer[i] = sha256(abi.encodePacked(layer[2*i], l
             }
             //the next layer above has half as many nodes
             numNodesInLayer /= 2;
```

```
        }
        //the first node in the layer is the root
        return layer[0];
    }
```

There is a `@notice` that doesn't hold.

> @notice requires the leaves.length is a power of 2

But whenever there is a `require` in natspec inside the project, it always holds. E.x.:

```
    /**
     * @notice Delegates from `staker` to `operator`.
     * @dev requires that:
     * 1) if `staker` is an EOA, then `signature` is valid ECSDA
     * 2) if `staker` is a contract, then `signature` must will
     */
```

[src/contracts/core/DelegationManager.sol#L89]()

```
     * WARNING: In order to mitigate against inflation/donation
     *          minimum amount of shares be either 0 or 1e9. A c
     *          be able to withdraw for 1e9-1 or less shares.
     *
```

[/src/contracts/strategies/StrategyBase.sol#L72]()

Tools Used

You can insert this into remix to check:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

import "hardhat/console.sol";

contract Owner {
```

```
    mapping(address => bool) internal frozenStatus;
    constructor() {
    }

    function dod() external returns (bytes32){
        bytes32[] memory leaves = new bytes32[](7);
        for (uint256 i = 0; i < 7; ++i) {
            leaves[i] = bytes32(i);
        }
        return merkleizeSha256(leaves);
    }

    function merkleizeSha256(
        bytes32[] memory leaves
    ) internal pure returns (bytes32) {
        //there are half as many nodes in the layer above the le
        uint256 numNodesInLayer = leaves.length / 2;
        //create a layer to store the internal nodes
        bytes32[] memory layer = new bytes32[](numNodesInLayer);
        //fill the layer with the pairwise hashes of the leaves
        for (uint i = 0; i < numNodesInLayer; i++) {
            layer[i] = sha256(abi.encodePacked(leaves[2*i], leav
        }
        //the next layer above has half as many nodes
        numNodesInLayer /= 2;
        //while we haven't computed the root
        while (numNodesInLayer != 0) {
            //overwrite the first numNodesInLayer nodes in layer
            for (uint i = 0; i < numNodesInLayer; i++) {
                layer[i] = sha256(abi.encodePacked(layer[2*i], l
            }
            //the next layer above has half as many nodes
            numNodesInLayer /= 2;
        }
        //the first node in the layer is the root
        return layer[0];
    }
}
```

Recommended Mitigation Steps

Either remove `@notice` or add this code for more security because sometimes you can just forget to check array size before calling that function:

```
    function merkleizeSha256(
        bytes32[] memory leaves
    ) internal pure returns (bytes32) {
+        uint256 len = leaves.length;
+        while (len > 1 && len % 2 == 0) {
+            len /= 2;
+        }
+        require(len==1, "requires the leaves.length is a power
        //there are half as many nodes in the layer above the le
        uint256 numNodesInLayer = leaves.length / 2;
        //create a layer to store the internal nodes
        bytes32[] memory layer = new bytes32[](numNodesInLayer);
        //fill the layer with the pairwise hashes of the leaves
        for (uint i = 0; i < numNodesInLayer; i++) {
            layer[i] = sha256(abi.encodePacked(leaves[2*i], leav
        }
        //the next layer above has half as many nodes
        numNodesInLayer /= 2;
        //while we haven't computed the root
        while (numNodesInLayer != 0) {
            //overwrite the first numNodesInLayer nodes in layer
            for (uint i = 0; i < numNodesInLayer; i++) {
                layer[i] = sha256(abi.encodePacked(layer[2*i], l
            }
            //the next layer above has half as many nodes
            numNodesInLayer /= 2;
        }
        //the first node in the layer is the root
        return layer[0];
    }
```

Remix:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

import "hardhat/console.sol";

contract Owner {

    mapping(address => bool) internal frozenStatus;
    constructor() {
```

```
    }

    function dod(uint len) external returns (bytes32){
        bytes32[] memory leaves = new bytes32[](len);
        for (uint256 i = 0; i < len; ++i) {
            leaves[i] = bytes32(i);
        }
        return merkleizeSha256(leaves);
    }
    function merkleizeSha256(
        bytes32[] memory leaves
    ) internal pure returns (bytes32) {
        uint256 len = leaves.length;
        while (len > 1 && len % 2 == 0) {
            len /= 2;
        }
        require(len==1, "requires the leaves.length is a power c
        //there are half as many nodes in the layer above the le
        uint256 numNodesInLayer = leaves.length / 2;
        //create a layer to store the internal nodes
        bytes32[] memory layer = new bytes32[](numNodesInLayer);
        //fill the layer with the pairwise hashes of the leaves
        for (uint i = 0; i < numNodesInLayer; i++) {
            layer[i] = sha256(abi.encodePacked(leaves[2*i], leav
        }
        //the next layer above has half as many nodes
        numNodesInLayer /= 2;
        //while we haven't computed the root
        while (numNodesInLayer != 0) {
            //overwrite the first numNodesInLayer nodes in layer
            for (uint i = 0; i < numNodesInLayer; i++) {
                layer[i] = sha256(abi.encodePacked(layer[2*i], l
            }
            //the next layer above has half as many nodes
            numNodesInLayer /= 2;
        }
        //the first node in the layer is the root
        return layer[0];
    }


}
```

Sidu28 (EigenLayer) disputed, disagreed with severity and commented:

> The comment is ambiguous, but is intended to actually state a precondition on the input. The comment will be changed.

Alex the Entreprenerd (judge) decreased severity to QA and commented:

> Every instance in the in-scope codebase does check, meaning that the finding cannot be considered a vulnerability.

> I can agree with the Warden that a valid refactoring would bring the check in the function to simplify the code.

> For this reason, am downgrading to QA - Refactoring (R)

[L-04] `claimableUserDelayedWithdrawals` sometimes returns unclaimable `DelayedWithdrawals`, so users will see incorrect data

Proof of Concept

The `canClaimDelayedWithdrawal` function will return false for a withdrawal, which the block duration has not passed. The same restriction will be checked whenever an actual withdrawal is triggered, but the `claimableUserDelayedWithdrawals` function does not take into account block duration validation.

```
        function claimableUserDelayedWithdrawals(address user) exter
            uint256 delayedWithdrawalsCompleted = _userWithdrawals[u
            uint256 delayedWithdrawalsLength = _userWithdrawals[user
            uint256 claimableDelayedWithdrawalsLength = delayedWithdo
            DelayedWithdrawal[] memory claimableDelayedWithdrawals =
            for (uint256 i = 0; i < claimableDelayedWithdrawalsLengt
                claimableDelayedWithdrawals[i] = _userWithdrawals[us
            }
            return claimableDelayedWithdrawals;
        }
    ...
        function canClaimDelayedWithdrawal(address user, uint256 inc
            return ((index >= _userWithdrawals[user].delayedWithdraw
        }
```

src/contracts/pods/DelayedWithdrawalRouter.sol#L110

Recommended Mitigation Steps

```
    function claimableUserDelayedWithdrawals(address user) exter
        uint256 delayedWithdrawalsCompleted = _userWithdrawals[u
        uint256 delayedWithdrawalsLength = _userWithdrawals[user
        uint256 claimableDelayedWithdrawalsLength = delayedWithd
        DelayedWithdrawal[] memory claimableDelayedWithdrawals;
        for (uint256 i = 0; i < claimableDelayedWithdrawalsLengt
            if (block.number < _userWithdrawals[user].delayedWit
                break;
            }
            claimableDelayedWithdrawals.push(_userWithdrawals[us
        }
        return claimableDelayedWithdrawals;
    }
```

[Alex the Entreprenerd (judge) decreased severity to QA]()

[L-05] The condition for full withdrawals in the code is different from that in the documentation

Proof of Concept

The condition in [docs]() for full withdrawal is `validator.withdrawableEpoch < executionPayload.slot/SLOTS_PER_EPOCH` while in the code its `validator.withdrawableEpoch <= executionPayload.slot/SLOTS_PER_EPOCH`.

```
    function verifyAndProcessWithdrawal(
        BeaconChainProofs.WithdrawalProofs calldata withdrawalPr
        bytes calldata validatorFieldsProof,
        bytes32[] calldata validatorFields,
        bytes32[] calldata withdrawalFields,
        uint256 beaconChainETHStrategyIndex,
        uint64 oracleBlockNumber
    )
  ...
```

```
            // reference: uint64 withdrawableEpoch = Endian.fromLitt
            if (Endian.fromLittleEndianUint64(validatorFields[Beacor
                _processFullWithdrawal(withdrawalAmountGwei, validat
            } else {
                _processPartialWithdrawal(slot, withdrawalAmountGwei
            }
        }
```

[src/contracts/pods/EigenPod.sol#L354]()

Recommended Mitigation Steps

Synchronize them with each other.

[Sidu28 (EigenLayer) disputed, disagreed with severity and commented]():

> We believe this is an informational-level issue. We failed to update this statement in the higher-level documentation. The code is correct.

[Alex the Entreprenerd (judge) decreased severity to QA and commented]():

> Great catch, but in lack of an impact am downgrading to QA.

> Will award extra points. L + 3.

[L-06] Missing validation to a threshold value on full withdrawal

Proof of Concept

According to the [docs]() there's supposed to be a validation against a const on full withdrawal, but its missing, which can lead to the system not working as expected.

> In this second case, in order to withdraw their balance from the EigenPod, stakers must provide a valid proof of their full withdrawal (differentiated from partial withdrawals through a simple comparison of the amount to a threshold value named MIN*FULL*WITHDRAWAL*AMOUNT*GWEI) against a beacon state root.

```
function _processFullWithdrawal(
    uint64 withdrawalAmountGwei,
    uint40 validatorIndex,
    uint256 beaconChainETHStrategyIndex,
    address recipient,
    VALIDATOR_STATUS status
) internal {
    uint256 amountToSend;

    // if the validator has not previously been proven to be
    if (status == VALIDATOR_STATUS.ACTIVE) {
        // if the withdrawal amount is greater than the REQU
        if (withdrawalAmountGwei >= REQUIRED_BALANCE_GWEI) {
            // then the excess is immediately withdrawable
            amountToSend = uint256(withdrawalAmountGwei - RE
            // and the extra execution layer ETH in the cont
            restakedExecutionLayerGwei += REQUIRED_BALANCE_G
        } else {
            // otherwise, just use the full withdrawal amoun
            restakedExecutionLayerGwei += withdrawalAmountGw
            // remove and undelegate 'extra' (i.e. "overcomm
            eigenPodManager.recordOvercommittedBeaconChainET
        }
    // if the validator *has* previously been proven to be '
    } else if (status == VALIDATOR_STATUS.OVERCOMMITTED) {
        // if the withdrawal amount is greater than the REQU
        if (withdrawalAmountGwei >= REQUIRED_BALANCE_GWEI) {
            // then the excess is immediately withdrawable
            amountToSend = uint256(withdrawalAmountGwei - RE
            // and the extra execution layer ETH in the cont
            restakedExecutionLayerGwei += REQUIRED_BALANCE_G
            /**
             * since in `verifyOvercommittedStake` the podOw
             * in order to allow the podOwner to complete th
             */
            eigenPodManager.restakeBeaconChainETH(podOwner,
        } else {
            // otherwise, just use the full withdrawal amoun
            restakedExecutionLayerGwei += withdrawalAmountGw
            /**
             * since in `verifyOvercommittedStake` the podOw
             * in order to allow the podOwner to complete th
             */
            eigenPodManager.restakeBeaconChainETH(podOwner,
        }
```

```
        // If the validator status is withdrawn, they have alrea
        }  else {
            revert("EigenPod.verifyBeaconChainFullWithdrawal: VA
        }

        // set the ETH validator status to withdrawn
        validatorStatus[validatorIndex] = VALIDATOR_STATUS.WITHD

        emit FullWithdrawalRedeemed(validatorIndex, recipient, w

        // send ETH to the `recipient`, if applicable
        if (amountToSend != 0) {
            _sendETH(recipient, amountToSend);
        }
    }
```

src/contracts/pods/EigenPod.sol#L364

Recommended Mitigation Steps

```
    function _processFullWithdrawal(
        uint64 withdrawalAmountGwei,
        uint40 validatorIndex,
        uint256 beaconChainETHStrategyIndex,
        address recipient,
        VALIDATOR_STATUS status
    ) internal {
+           require(withdrawalAmountGwei >= MIN_FULL_WITHDRAWAL
+           "stakers must provide a valid proof of their full w

        uint256 amountToSend;

        // if the validator has not previously been proven to be
        if (status == VALIDATOR_STATUS.ACTIVE) {
            // if the withdrawal amount is greater than the REQU
            if (withdrawalAmountGwei >= REQUIRED_BALANCE_GWEI) {
                // then the excess is immediately withdrawable
                amountToSend = uint256(withdrawalAmountGwei - RE
                // and the extra execution layer ETH in the cont
                restakedExecutionLayerGwei += REQUIRED_BALANCE_G
            } else {
                // otherwise, just use the full withdrawal amour
                restakedExecutionLayerGwei += withdrawalAmountGw
```

```
                // remove and undelegate 'extra' (i.e. "overcomm
                eigenPodManager.recordOvercommittedBeaconChainET
            }
        // if the validator *has* previously been proven to be '
        } else if (status == VALIDATOR_STATUS.OVERCOMMITTED) {
            // if the withdrawal amount is greater than the REQU
            if (withdrawalAmountGwei >= REQUIRED_BALANCE_GWEI) {
                // then the excess is immediately withdrawable
                amountToSend = uint256(withdrawalAmountGwei - RE
                // and the extra execution layer ETH in the cont
                restakedExecutionLayerGwei += REQUIRED_BALANCE_G
                /**
                 * since in `verifyOvercommittedStake` the podOw
                 * in order to allow the podOwner to complete th
                 */
                eigenPodManager.restakeBeaconChainETH(podOwner,
            } else {
                // otherwise, just use the full withdrawal amour
                restakedExecutionLayerGwei += withdrawalAmountGw
                /**
                 * since in `verifyOvercommittedStake` the podOw
                 * in order to allow the podOwner to complete th
                 */
                eigenPodManager.restakeBeaconChainETH(podOwner,
            }
        // If the validator status is withdrawn, they have alrea
        }  else {
            revert("EigenPod.verifyBeaconChainFullWithdrawal: VA
        }

        // set the ETH validator status to withdrawn
        validatorStatus[validatorIndex] = VALIDATOR_STATUS.WITHD

        emit FullWithdrawalRedeemed(validatorIndex, recipient, w

        // send ETH to the `recipient`, if applicable
        if (amountToSend != 0) {
            _sendETH(recipient, amountToSend);
        }
    }
```

Sidu28 (EigenLayer) disagreed with severity and commented:

> This is an informational-level issue. We failed to update this statement in the higher-level documentation. This check is not necessary.

Alex the Entreprenerd (judge) decreased severity to QA and commented:

> The 4 logical paths seem to cover the possible scenarios.

> In lack of further info, am downgrading to QA.

[L-07] User can stake twice on beacon chain from same eipod, thus losing funds due to same withdrawal credentials

Proof of Concept

There are no restriction to how many times user can stake on beacon with `EigenPodManager` on `EigenPod`, thus all of them will have the same `_podWithdrawalCredentials()` and I think first deposit will be lost.

```
    function stake(bytes calldata pubkey, bytes calldata signatu
        // stake on ethpos
        require(msg.value == 32 ether, "EigenPod.stake: must ini
        ethPOS.deposit{value : 32 ether}(pubkey, _podWithdrawalC
        emit EigenPodStaked(pubkey);
    }
```

src/contracts/pods/EigenPod.sol#L159

There are some ways users can make a mistake by calling it twice or they would like to create another one. I've looked into rocketpool contracts; they are not allowing users to stake twice with the same pubkeys, so I think its important to implement the same security issue.

```
    function preStake(bytes calldata _validatorPubkey, bytes cal
  ...
        require(rocketMinipoolManager.getMinipoolByPubkey(_valic
        // Set minipool pubkey
        rocketMinipoolManager.setMinipoolPubkey(_validatorPubkey
```

```
        // Get withdrawal credentials
        bytes memory withdrawalCredentials = rocketMinipoolManag
        // Send staking deposit to casper
        casperDeposit.deposit{value : prelaunchAmount}(_validato
        // Emit event
        emit MinipoolPrestaked(_validatorPubkey, _validatorSigna
    }
```

[contracts/contract/minipool/RocketMinipoolDelegate.sol#L235]()

Same safe thing done in frax finance:

```
    function depositEther(uint256 max_deposits) external nonReer
...
            // Deposit the ether in the ETH 2.0 deposit contract
            depositContract.deposit{value: DEPOSIT_SIZE}(
                pubKey,
                withdrawalCredential,
                signature,
                depositDataRoot
            );

            // Set the validator as used so it won't get an extr
            activeValidators[pubKey] = true;
    ...
}
```

[src/frxETHMinter.sol#L156]()

Tools Used

POC

```
    function testWithdrawFromPod() public {
        cheats.startPrank(podOwner);
        eigenPodManager.stake{value: stakeAmount}(pubkey, signat
+        eigenPodManager.stake{value: stakeAmount}(pubkey, signa
        cheats.stopPrank();

        IEigenPod pod = eigenPodManager.getPod(podOwner);
        uint256 balance = address(pod).balance;
```

```
        cheats.deal(address(pod), stakeAmount);

        cheats.startPrank(podOwner);
        cheats.expectEmit(true, false, false, false);
        emit DelayedWithdrawalCreated(podOwner, podOwner, balanc
        pod.withdrawBeforeRestaking();
        cheats.stopPrank();
        require(address(pod).balance == 0, "Pod balance should b
    }
```

Recommended Mitigation Steps

You can look at rocketpool contracts and borrow their logic:

```
    function stake(bytes calldata pubkey, bytes calldata signatu
+         require(EigenPodManager.getEigenPodByPubkey(_validatorP
+         EigenPodManager.setEigenPodByPubkey(_validatorPubkey);

        require(msg.value == 32 ether, "EigenPod.stake: must ini
        ethPOS.deposit{value : 32 ether}(pubkey, _podWithdrawalC
        emit EigenPodStaked(pubkey);
    }
```

ChaoticWalrus (EigenLayer) disagreed with severity and commented:

> We believe this is purely informational. People can already stake multiple times through the `ETH2Deposit` contract.

Alex the Entreprenerd (judge) decreased severity to QA and commented:

> Downgrading to QA for now. @volodya - please do send me proof that a new withdrawal would be bricked (I believe it would be releasable via the normal flow).

Alex the Entreprenerd (judge) commented:

> Have had a informal confirmation that excess ETH is refunded as rewards.

> In lack of additional info, am maintaining the judgment.

Gas Optimizations

For this audit, 13 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **neutiyoo** received the top score from the judge.

The following wardens also submitted reports: 0xSmartContract, turvy_fuzz, pontifex, niser93, 0xnev, QiuhaoLi, Aymen0909, ihtishamsudo, tonisives, clayj, ReyAdmirado and naman1778.

[G-01] Optimize `merkleizeSha256` function for gas-efficiency

Although the current implementation of the `merkleizeSha256` function in the `Merkle` contract is correct, it can be more gas-efficient by making use of the following optimizations:

1. In-place Computation

The `merkleizeSha256` function can be optimized by using in-place computation to store intermediate hashes at each level of the Merkle tree. This approach eliminates the need to create new arrays, reducing memory usage and gas costs.

Note: this optimization requires the `leaves` array not to be used again after it is modified. Based on the current implementation, this optimization is safe because the `leaves` array is not used again after it is modified.

2. Assembly

The use of assembly code to load the left and right siblings into memory is more gas-efficient than using the `abi.encodePacked` function.

3. Unchecked Arithmetic

The use of unchecked arithmetic for `uint i` is more gas-efficient as it skips checks for overflow or underflow. This optimization is safe because `i` is always less than `numNodesInLayer`, meaning that overflow is not possible.

Proof of Concept

The function `merkleizeSha256Optimized` provided below is an optimized version of the `merkleizeSha256` function.

[src/contracts/libraries/MerkleOptimized.sol]()

```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.12;

library MerkleOptimized {
    /**
     * @notice Returns the Merkle root of a tree created fro
     * @param leaves The leaves of the Merkle tree. This par
     * @return The Merkle root of the tree.
     *
     * @notice Requires the leaves.length is a power of 2.
     * @dev This is adapted from https:
     */
    function merkleizeSha256Optimized(
        bytes32[] memory leaves
    ) internal pure returns (bytes32) {
        // Reserve memory space for our hashes.
        bytes memory buf = new bytes(64);

        // We'll need to keep track of left and right siblin
        bytes32 leftSibling;
        bytes32 rightSibling;

        // Number of non-empty nodes at the current depth.
        uint256 rowSize = leaves.length;

        // Common sub-expressions
        uint256 halfRowSize; // rowSize / 2

        while (rowSize > 1) {
            halfRowSize = rowSize / 2;

            for (uint256 i = 0; i < halfRowSize; ) {
                leftSibling = leaves[(2 * i)];
                rightSibling = leaves[(2 * i) + 1];
                assembly {
                    mstore(add(buf, 32), leftSibling)
                    mstore(add(buf, 64), rightSibling)
                }

```

```
                    leaves[i] = sha256(buf);

                    unchecked {
                        ++i;
                    }
                }

                rowSize = halfRowSize;
            }

            return leaves[0];
        }
}
```

Function Name	min	avg	median	max	# calls
merkleizeSha256	2353	274987	62975	1396167	10
merkleizeSha256Optimized	2136	238896	56158	1197190	10

Improvement	
Minimum	9.22%
Average	13.12%
Median	10.82%
Maximum	14.25%

The data shows a significant increase in gas efficiency with the use of `merkleizeSha256Optimized` compared to `merkleizeSha256`. It's worth emphasizing, that these results are influenced by the input data and execution environment, so the actual improvement may differ in other contexts. Nonetheless, the results provide valuable insight into the potential gas cost savings that can be achieved by leveraging the optimized version of the function.

The test codes are the following:

src/test/unit/Merkle.t.sol

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity =0.8.12;
```

```
import {Test} from "forge-std/Test.sol";
import {Merkle} from "../../contracts/libraries/Merkle.sol";
import {MerkleOptimized} from "../../contracts/libraries/MerkleC

contract MerkleMock {
    function merkleizeSha256(
        bytes32[] calldata leaves
    ) external pure returns (bytes32) {
        return Merkle.merkleizeSha256(leaves);
    }

    function merkleizeSha256Optimized(
        bytes32[] calldata leaves
    ) external pure returns (bytes32) {
        return MerkleOptimized.merkleizeSha256Optimized(leaves);
    }
}

contract MerkleTest is Test {
    MerkleMock public c;

    function setUp() external {
        c = new MerkleMock();
    }

    function gen(uint256 length) internal pure returns (bytes32[
        bytes32[] memory leaves = new bytes32[](length);
        for (uint i = 0; i < length; i++) {
            leaves[i] = bytes32(i);
        }
        return leaves;
    }

    function testMerkleizeSha256Equivalence() external {
        for (uint i = 2; i <= 1024; i *= 2) {
            assertEq(
                c.merkleizeSha256(gen(i)),
                c.merkleizeSha256Optimized(gen(i)),
                "ok"
            );
        }
    }
}
```

Recommendation

Consider optimizing `merkleizeSha256` by using in-place computation, assembly, and unchecked arithmetic.

[G-02] Use unchecked arithmetic in `processInclusionProofKeccak` and `processInclusionProofSha256` functions

The `processInclusionProofKeccak` and `processInclusionProofSha256` functions in the `Merkle` contract include unnecessary arithmetic checks for incrementing `uint256 i` in a for-loop. By using unchecked arithmetic, the gas cost of executing these functions can be reduced.

1. `processInclusionProofKeccak`

src/contracts/libraries/Merkle.sol#L48-L50

```
function processInclusionProofKeccak(bytes memory proof, byt
    bytes32 computedHash = leaf;
    for (uint256 i = 32; i <= proof.length; i+=32) {
...
```

2. `processInclusionProofSha256`

src/contracts/libraries/Merkle.sol#L99-L101

```
function processInclusionProofSha256(bytes memory proof, byt
    bytes32[1] memory computedHash = [leaf];
    for (uint256 i = 32; i <= proof.length; i+=32) {
...
```

Based on the current implementation, this optimization is safe because overflow is not possible, as the length of `proof` is validated before the function call.

Recommendation

Consider using unchecked arithmetic for `uint256 i`.

```
unchecked {
    i += 32;
}
```

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.