hexens × EigenLayer

Oct.23

SECURITY REVIEW REPORT FOR EIGENLAYER STAGE 2 TESTNET

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date
02.10.2023

Audit Completion Date
16.10.2023

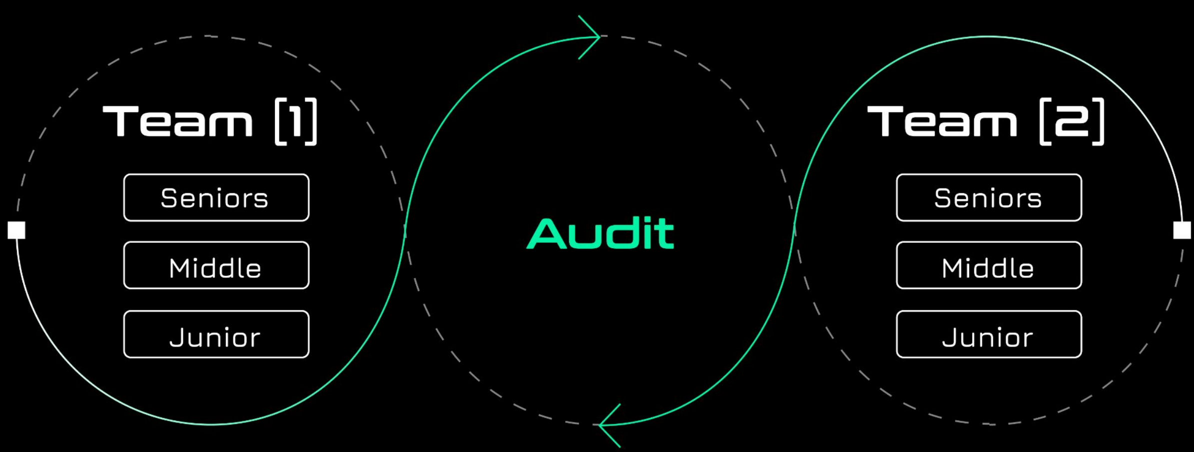
METHODOLOGY

COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered a new update to EigenLayer's EigenPod smart contract and related smart contracts, such as the EigenPodmanager. The update is EigenLayer's Stage 2 Testnet, which will enable true re-staking for EigenPods through beacon chain proofs.

Our security assessment was a full review of these contracts, spanning a total of 2 weeks.

During our audit, we have identified 3 critical severity vulnerabilities. One vulnerability would allow anyone to forge withdrawal proofs for EigenPods and another one would cause user funds to get frozen.

We have also identified 2 medium severity vulnerabilities, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/Layr-Labs/eigenlayer-contracts/tree/230554b09ef3c7303f51f015363bcbed3ef6be76/src/contracts/pods

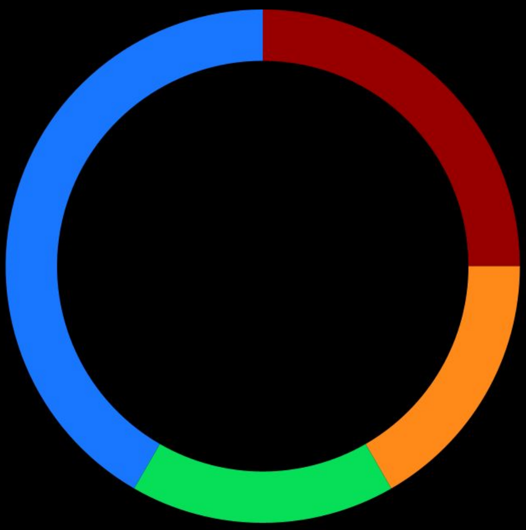
The issues described in this report were fixed. Corresponding commits are mentioned in the description.

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	3
HIGH	0
MEDIUM	2
LOW	2
INFORMATIONAL	5

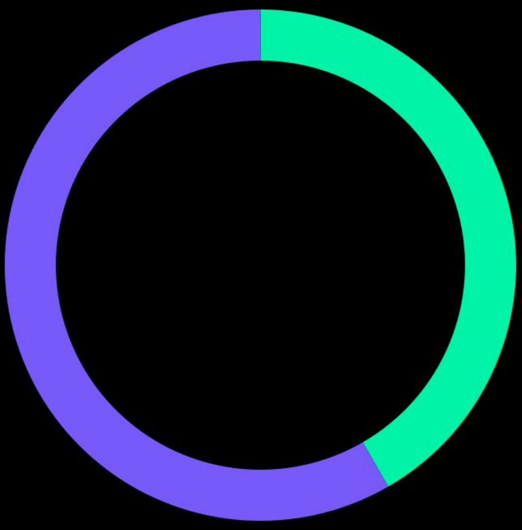
TOTAL: 14

SEVERITY



● Critical ● Medium ● Low
● Informational

STATUS



● Fixed ● Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

EIG-7. DIRECT LOSS OF USER PRINCIPAL FUNDS WHEN PROCESSING A FULL WITHDRAWAL OF PENALISED VALIDATOR

SEVERITY: Critical

PATH: EigenPod.sol:_processFullWithdrawal:L652-706

REMEDIATION: the full withdrawal for a penalised validator should not only use _calculateRestakedBalanceGwei on the withdrawal amount or the difference should be calculated and returned to the user immediately by setting verifiedWithdrawal.amountToSend

STATUS: fixed

DESCRIPTION:

The EigenPod allows users to proof withdrawals from the beacon chain for connected validators, so the ETH can then be withdrawn from the EigenPod. The function **_processFullWithdrawal** is called whenever a withdrawal has happened after the withdrawable epoch of a validator.

In this function and in the branch where it processes a full withdrawal for a penalised validator (i.e. withdrawal amount is less than **MAX_VALIDATOR_BALANCE_GWEI**) on lines 680-686, it uses **_calculateRestakedBalanceGwei** to convert the withdrawal amount.

This function is normally used to calculate a validator's balance and the amount of shares to mint in the EigenPodManager. This function also greatly rounds down the amount.

Here it is used directly on the withdrawal amount and only this rounded down amount is added to **withdrawableRestakedExecutionLayerGwei** and **withdrawalAmountWei**. The rounded down amount will be strictly less than the actual withdrawn amount, but the leftover is basically discarded and becomes stuck in the contract.

For example, if the **MAX_VALIDATOR_BALANCE_GWEI** is 32 ETH and **RESTAKED_BALANCE_OFFSET_GWEI** is 0.75 ETH, then for a withdrawal amount of 30.74 ETH it would round down to 29 ETH, causing almost 1.5 ETH of principal funds to become lost.

```
function _processFullWithdrawal(
    uint40 validatorIndex,
    bytes32 validatorPubkeyHash,
    uint64 withdrawalHappenedTimestamp,
    address recipient,
    uint64 withdrawalAmountGwei,
    ValidatorInfo memory validatorInfo
) internal returns (VerifiedWithdrawal memory) {
    VerifiedWithdrawal memory verifiedWithdrawal;
    uint256 withdrawalAmountWei;

    uint256 currentValidatorRestakedBalanceWei = validatorInfo.restakedBalanceGwei * GWEI_TO_WEI;

    /**
     * If the validator is already withdrawn and additional deposits are made, they will be automatically withdrawn
     * in the beacon chain as a full withdrawal.  Thus such a validator can prove another full withdrawal, and
     * withdraw that ETH via the queuedWithdrawal flow in the strategy manager.
     */
        // if the withdrawal amount is greater than the MAX_VALIDATOR_BALANCE_GWEI (i.e. the max amount restaked on
EigenLayer, per ETH validator)
    uint64 maxRestakedBalanceGwei = _calculateRestakedBalanceGwei(MAX_VALIDATOR_BALANCE_GWEI);
    if (withdrawalAmountGwei > maxRestakedBalanceGwei) {
        // then the excess is immediately withdrawable
        verifiedWithdrawal.amountToSend =
            uint256(withdrawalAmountGwei - maxRestakedBalanceGwei) *
            uint256(GWEI_TO_WEI);
        // and the extra execution layer ETH in the contract is MAX_VALIDATOR_BALANCE_GWEI, which must be withdrawn
through EigenLayer's normal withdrawal process
        withdrawableRestakedExecutionLayerGwei += maxRestakedBalanceGwei;
        withdrawalAmountWei = maxRestakedBalanceGwei * GWEI_TO_WEI;
    } else {
        // otherwise, just use the full withdrawal amount to continue to "back" the podOwner's remaining shares in
EigenLayer
        // (i.e. none is instantly withdrawable)
        withdrawalAmountGwei = _calculateRestakedBalanceGwei(withdrawalAmountGwei);
        withdrawableRestakedExecutionLayerGwei += withdrawalAmountGwei;
        withdrawalAmountWei = withdrawalAmountGwei * GWEI_TO_WEI;
    }
```

```
    // if the amount being withdrawn is not equal to the current accounted for validator balance, an update
must be made
    if (currentValidatorRestakedBalanceWei != withdrawalAmountWei) {
        verifiedWithdrawal.sharesDelta = _calculateSharesDelta({
            newAmountWei: withdrawalAmountWei,
            currentAmountWei: currentValidatorRestakedBalanceWei
        });
    }

    // now that the validator has been proven to be withdrawn, we can set their restaked balance to 0
    validatorInfo.restakedBalanceGwei = 0;
    validatorInfo.status = VALIDATOR_STATUS.WITHDRAWN;
    validatorInfo.mostRecentBalanceUpdateTimestamp = withdrawalHappenedTimestamp;

    _validatorPubkeyHashToInfo[validatorPubkeyHash] = validatorInfo;

    emit FullWithdrawalRedeemed(validatorIndex, withdrawalHappenedTimestamp, recipient, withdrawalAmountGwei);

    return verifiedWithdrawal;
}

function _calculateRestakedBalanceGwei(uint64 amountGwei) internal view returns (uint64) {
    if (amountGwei <= RESTAKED_BALANCE_OFFSET_GWEI) {
        return 0;
    }
    uint64 effectiveBalanceGwei = uint64(((amountGwei - RESTAKED_BALANCE_OFFSET_GWEI) / GWEI_TO_WEI) *
GWEI_TO_WEI);
    return uint64(MathUpgradeable.min(MAX_VALIDATOR_BALANCE_GWEI, effectiveBalanceGwei));
}
```

EIG-10. WITHDRAWAL PROOFS CAN BE FORGED DUE TO MISSING INDEX BIT SIZE CHECK

SEVERITY: Critical

PATH: BeaconChainProofs.sol:verifyWithdrawal:L269-399

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In order for the EigenPod to verify and consequently process a withdrawal from the beacon chain, it uses the BeaconChainsProofs' **verifyWithdrawal** function. This function takes various parameters to prove the existence of a supplied Withdrawal struct in the beacon chain state Merkle root.

To do so, it proves 5 different leaves: the block root against the beacon state root, the slot number against the block root, the execution root against the block root, the timestamp against the execution root and finally the withdrawal against the execution root.

The first proof is not fully checked and it is vulnerable to tampering. Because the other proofs all depend on this first proof, it also influences the others and allows for tampering there as well.

The block root is proven against the beacon state root by first traversing to the historical summaries root in the beacon state. This is done using a constant **HISTORICAL_SUMMARIES_INDEX** which is then concatenated with the **historicalSummaryIndex** that is supplied by the user to choose the right block from the historical summary tree.

This is again concatenated with **blockRootIndex**, also supplied by the user to choose the right block from the historical summary tree.

To make sure that the user does not control the flow of traversal through the Merkle tree, it is important to make sure that the proof lengths are of correct lengths and that the bit size of indexes are not greater than the tree height. This is done correctly for each proof, except for the **historySummaryIndex**, which is missing a size check.

For example, **blockRootIndex** is check on lines 279-282:

```
require(
    withdrawalProof.blockRootIndex < 2 ** BLOCK_ROOTS_TREE_HEIGHT,
    "BeaconChainProofs.verifyWithdrawal: blockRootIndex is too large"
);
```

But **historySummaryIndex** is missing such checks.

This allows a malicious user to provide an index greater than the tree height. If this were a simple, single Merkle tree with one index, then it would not be problem. But in this case we are traversing combinations of multiple trees from the beacon state root to the block root and so it allows for other indexes to be overwritten.

For example, in the first proof the combined index for the proof is calculated using the concatenations as described above:

```
uint256 historicalBlockHeaderIndex = (HISTORICAL_SUMMARIES_INDEX <<
    ((HISTORICAL_SUMMARIES_TREE_HEIGHT + 1) + 1 + (BLOCK_ROOTS_TREE_HEIGHT))) |
    (uint256(withdrawalProof.historicalSummaryIndex) << (1 + (BLOCK_ROOTS_TREE_HEIGHT))) |
    (BLOCK_SUMMARY_ROOT_INDEX << (BLOCK_ROOTS_TREE_HEIGHT)) |
    uint256(withdrawalProof.blockRootIndex);
```

As can be seen, the **historySummaryIndex** is appended on top of the constant **HISTORICAL_SUMMARIES_INDEX** (which should ensure that we first traverse from the beacon state root to the history summaries root). Now that **historySummaryIndex** is unbounded, it becomes possible to overwrite the **HISTORY_SUMMARIES_INDEX** to any value and make the traversal go into any other field of the beacon state instead: https://github.com/ethereum/consensus-specs/blob/dev/specs/capella/beacon-chain.md#beaconstate

In order to exploit this bug and forge a withdrawal proof, it becomes important to plan a path where the proofs will go and result in valid values for the withdrawal struct.

For example, the first proof provides a lot of freedom in traversing from the **historicalSummaryIndex** and **blockRootIndex**.

The timestamp, slot and execution root proofs can be ignored as the proof lengths are short and they can simply pass with a hash value as leaf. The hash value would be interpreted as timestamp and slot, which will not make any checks fails, rather it gives unique timestamps and slots which could give either full or partial withdrawals depending on the validator's withdrawable epoch.

Only the withdrawal proof will require some planning, as the withdrawal fields will either have to be some other leaf value or brute-forced hashes (intermediate leaves) in some part of the entire beacon state Merkle tree. Brute-forced hashes would still work, as the only used fields from the withdrawal struct are the validator index (which is parsed into 5 bytes) and the withdrawal amount (which is parsed into 8 bytes, expressed in Gwei and should be not too large).

A working exploit would allow a malicious user to proof withdrawals for themselves or victim users. If the timestamp could be controlled, then it can also be used to proof 0 amount withdrawals for victim users that have a real withdrawal at some timestamp. The timestamp would be set to `true` and they cannot prove the actual withdrawal anymore, locking their ETH.

```
function verifyWithdrawal(
        bytes32 beaconStateRoot,
        bytes32[] calldata withdrawalFields,
        WithdrawalProof calldata withdrawalProof
    ) internal view {
        require(
            withdrawalFields.length == 2 ** WITHDRAWAL_FIELD_TREE_HEIGHT,
            "BeaconChainProofs.verifyWithdrawal: withdrawalFields has incorrect length"
        );

        require(
            withdrawalProof.blockRootIndex < 2 ** BLOCK_ROOTS_TREE_HEIGHT,
            "BeaconChainProofs.verifyWithdrawal: blockRootIndex is too large"
        );
        require(
            withdrawalProof.withdrawalIndex < 2 ** WITHDRAWALS_TREE_HEIGHT,
            "BeaconChainProofs.verifyWithdrawal: withdrawalIndex is too large"
        );

        require(
            withdrawalProof.withdrawalProof.length ==
                32 * (EXECUTION_PAYLOAD_HEADER_FIELD_TREE_HEIGHT + WITHDRAWALS_TREE_HEIGHT + 1),
            "BeaconChainProofs.verifyWithdrawal: withdrawalProof has incorrect length"
        );
        require(
            withdrawalProof.executionPayloadProof.length ==
                32 * (BEACON_BLOCK_HEADER_FIELD_TREE_HEIGHT + BEACON_BLOCK_BODY_FIELD_TREE_HEIGHT),
            "BeaconChainProofs.verifyWithdrawal: executionPayloadProof has incorrect length"
        );
        require(
            withdrawalProof.slotProof.length == 32 * (BEACON_BLOCK_HEADER_FIELD_TREE_HEIGHT),
            "BeaconChainProofs.verifyWithdrawal: slotProof has incorrect length"
        );
        require(
            withdrawalProof.timestampProof.length == 32 * (EXECUTION_PAYLOAD_HEADER_FIELD_TREE_HEIGHT),
            "BeaconChainProofs.verifyWithdrawal: timestampProof has incorrect length"
        );
```

```
        require(
            withdrawalProof.historicalSummaryBlockRootProof.length ==
                32 *
                    (BEACON_STATE_FIELD_TREE_HEIGHT +
                        (HISTORICAL_SUMMARIES_TREE_HEIGHT + 1) +
                        1 +
                        (BLOCK_ROOTS_TREE_HEIGHT)),
            "BeaconChainProofs.verifyWithdrawal: historicalSummaryBlockRootProof has incorrect length"
        );
        /**
         * Note: Here, the "1" in "1 + (BLOCK_ROOTS_TREE_HEIGHT)" signifies that extra step of choosing the
"block_root_summary" within the individual
         * "historical_summary". Everywhere else it signifies merkelize_with_mixin, where the length of an array is hashed
with the root of the array,
         * but not here.
         */
        uint256 historicalBlockHeaderIndex = (HISTORICAL_SUMMARIES_INDEX <<
            ((HISTORICAL_SUMMARIES_TREE_HEIGHT + 1) + 1 + (BLOCK_ROOTS_TREE_HEIGHT))) |
            (uint256(withdrawalProof.historicalSummaryIndex) << (1 + (BLOCK_ROOTS_TREE_HEIGHT))) |
            (BLOCK_SUMMARY_ROOT_INDEX << (BLOCK_ROOTS_TREE_HEIGHT)) |
            uint256(withdrawalProof.blockRootIndex);

        require(
            Merkle.verifyInclusionSha256({
                proof: withdrawalProof.historicalSummaryBlockRootProof,
                root: beaconStateRoot,
                leaf: withdrawalProof.blockRoot,
                index: historicalBlockHeaderIndex
            }),
            "BeaconChainProofs.verifyWithdrawal: Invalid historicalsummary merkle proof"
        );

        //Next we verify the slot against the blockRoot
        require(
            Merkle.verifyInclusionSha256({
                proof: withdrawalProof.slotProof,
                root: withdrawalProof.blockRoot,
                leaf: withdrawalProof.slotRoot,
                index: SLOT_INDEX
            }),
```

```
        "BeaconChainProofs.verifyWithdrawal: Invalid slot merkle proof"
    );

    {
        // Next we verify the executionPayloadRoot against the blockRoot
        uint256 executionPayloadIndex = (BODY_ROOT_INDEX << (BEACON_BLOCK_BODY_FIELD_TREE_HEIGHT)) |
            EXECUTION_PAYLOAD_INDEX;
        require(
            Merkle.verifyInclusionSha256({
                proof: withdrawalProof.executionPayloadProof,
                root: withdrawalProof.blockRoot,
                leaf: withdrawalProof.executionPayloadRoot,
                index: executionPayloadIndex
            }),
            "BeaconChainProofs.verifyWithdrawal: Invalid executionPayload merkle proof"
        );
    }

    // Next we verify the timestampRoot against the executionPayload root
    require(
        Merkle.verifyInclusionSha256({
            proof: withdrawalProof.timestampProof,
            root: withdrawalProof.executionPayloadRoot,
            leaf: withdrawalProof.timestampRoot,
            index: TIMESTAMP_INDEX
        }),
        "BeaconChainProofs.verifyWithdrawal: Invalid blockNumber merkle proof"
    );

    {
        /**
         * Next we verify the withdrawal fields against the blockRoot:
         * First we compute the withdrawal_index relative to the blockRoot by concatenating the indexes of all the
         * intermediate root indexes from the bottom of the sub trees (the withdrawal container) to the top, the
blockRoot.
         * Then we calculate merkleize the withdrawalFields container to calculate the the withdrawalRoot.
         * Finally we verify the withdrawalRoot against the executionPayloadRoot.
         *
         *
```

```
         * Note: Merkleization of the withdrawals root tree uses MerkleizeWithMixin, i.e., the length of the array is
hashed with the root of
         * the array.  Thus we shift the WITHDRAWALS_INDEX over by WITHDRAWALS_TREE_HEIGHT + 1 and not just
WITHDRAWALS_TREE_HEIGHT.
         */
        uint256 withdrawalIndex = (WITHDRAWALS_INDEX << (WITHDRAWALS_TREE_HEIGHT + 1)) |
            uint256(withdrawalProof.withdrawalIndex);
        bytes32 withdrawalRoot = Merkle.merkleizeSha256(withdrawalFields);
        require(
            Merkle.verifyInclusionSha256({
                proof: withdrawalProof.withdrawalProof,
                root: withdrawalProof.executionPayloadRoot,
                leaf: withdrawalRoot,
                index: withdrawalIndex
            }),
            "BeaconChainProofs.verifyWithdrawal: Invalid withdrawal merkle proof"
        );
    }
}
```

Check the length of **historicalSummaryIndex**, for example:

```
require(
    withdrawalProof.historicalSummaryIndex < 2 ** HISTORICAL_SUMMARIES_TREE_HEIGHT,
    "BeaconChainProofs.verifyWithdrawal: historicalSummaryIndex is too large"
);
```

EIG-14. M1 EIGENPODS CAN RESTAKE AND WITHDRAW WITHOUT PROVING AND BURNING SHARES

SEVERITY: Critical

PATH:
EigenPod.sol:withdrawNonBeaconChainETHBalanceWei:L393-404

REMEDIATION: the function _processWithdrawalBeforeRestaking should also zero out nonBeaconChainETHBalanceWei, as the entire balance will be withdrawn anyway

STATUS: fixed

DESCRIPTION:

The EigenPod offers 2 functions to withdraw ETH directly without proving a withdrawal. The first one is **withdrawBeforeRestaking**, which requires **hasRestaked** to be **false**. The second one is **withdrawNonBeaconChainETHBalanceWei**, which is introduced in M2 and takes from a balance counter that is increased upon execution of receive.

The former is to allow for depositing and withdrawing before restaking (and so without proofs) and the latter is to withdraw any ETH that was mistakenly sent to the EigenPod. However, they do not work well together.

Most M1 EigenPods will still have **hasRestaked** be set to **false**, as proving is not enabled for M1 EigenPods. Once they are upgraded to the M2 implementation, they will have access to both functions.

This can then be exploited to restake and withdraw the stake without proving the withdrawal and consequently without burning the shares, effectively allowing for free minting of shares.

Consider the following scenario:

1. An M1 EigenPod with **hasRestaked** is **false** is upgraded to the M2 implementation.
2. The owner sends 32 ETH to the EigenPod, the **nonBeaconChainETHBalanceWei** increases with 32 ETH.
3. The owner calls **withdrawBeforeRestaking**, which will simply send the entire ETH balance (32 ETH) to the owner.
4. The owner activates restaking, creates a validator and verifies the withdrawal credentials, receiving 32 ETH in shares.
5. The owner exits the validator and the EigenPod receives the 32 ETH principal.
6. The owner can now call **withdrawNonBeaconChainETHBalanceWei** to withdraw the 32 ETH, because **nonBeaconChainETHBalanceWei** is still equal to 32 ETH, bypassing the withdrawal proof and keeping the 32 ETH shares.
7. Repeat (or use multiple validators) for more free shares.

```
receive() external payable {
    nonBeaconChainETHBalanceWei += msg.value;
    emit NonBeaconChainETHReceived(msg.value);
}

function withdrawNonBeaconChainETHBalanceWei(
    address recipient,
    uint256 amountToWithdraw
) external onlyEigenPodOwner {
    require(
        amountToWithdraw <= nonBeaconChainETHBalanceWei,
        "EigenPod.withdrawnonBeaconChainETHBalanceWei: amountToWithdraw is greater than
nonBeaconChainETHBalanceWei"
    );
    nonBeaconChainETHBalanceWei -= amountToWithdraw;
    emit NonBeaconChainETHWithdrawn(recipient, amountToWithdraw);
    _sendETH_AsDelayedWithdrawal(recipient, amountToWithdraw);
}

function withdrawBeforeRestaking() external onlyEigenPodOwner hasNeverRestaked {
    _processWithdrawalBeforeRestaking(podOwner);
}

function _processWithdrawalBeforeRestaking(address _podOwner) internal {
    mostRecentWithdrawalTimestamp = uint32(block.timestamp);
    _sendETH_AsDelayedWithdrawal(_podOwner, address(this).balance);
}
```

EIG-17. OPERATOR OR DELEGATION APPROVER HAVE THE POWER TO CENSOR DELEGATED STAKERS

SEVERITY: Medium

PATH: DelegationManager.sol:undelegate:L213-257

REMEDIATION: the staker should claim back his shares immediately without waiting for the withdrawalDelayBlocks. Delay withdrawalDelayBlocks should be applied only if the staker withdraws their tokens (ETH or LSTs) back

STATUS: acknowledged, see commentary

DESCRIPTION:

The operator or delegation approver that stakers have delegated to, have the power to selectively censor those stakers.

As soon as anyone can register an operator on EigenLayer, an attacker can create operators massively and grief stakers by undelegating them, consequently damaging protocol stability and making users averse to use the protocol.

The operator or delegation approver can call the **undelegate()** function for a particular staker. This will move the staker to the undelegation limbo (call to **forceIntoUndelegationLimbo()**) , and forcefully removing staker's shares from **StartegyManager** (call to **forceTotalWithdrawal()**).

```solidity
function undelegate(
    address staker
) external onlyWhenNotPaused(PAUSED_UNDELEGATION) returns (bytes32 withdrawalRoot) {
    require(isDelegated(staker), "DelegationManager.undelegate: staker must be delegated to undelegate");
    address operator = delegatedTo[staker];
    require(!isOperator(staker), "DelegationManager.undelegate: operators cannot be undelegated");
    require(staker != address(0), "DelegationManager.undelegate: cannot undelegate zero address");
    require(
        msg.sender == staker ||
            msg.sender == operator ||                              // @audit
            msg.sender == _operatorDetails[operator].delegationApprover,    // @audit
        "DelegationManager.undelegate: caller cannot undelegate staker"
    );

    // remove any shares from the delegation system that the staker currently has delegated, if necessary
    // force the staker into "undelegation limbo" in the EigenPodManager if necessary
   if (eigenPodManager.podOwnerHasActiveShares(staker)) {
        uint256 podShares = eigenPodManager.forceIntoUndelegationLimbo(staker, operator); // @audit
        ...
    }
    // force-queue a withdrawal of all of the staker's shares from the StrategyManager, if necessary
    if (strategyManager.stakerStrategyListLength(staker) != 0) {
        IStrategy[] memory strategies;
        uint256[] memory strategyShares;
        (strategies, strategyShares, withdrawalRoot) = strategyManager.forceTotalWithdrawal(staker); // @audit
        ...
}
```

The staker is able to delegate his shares to another operator only after **withdrawalDelayBlocks** period. Which is **1 week** according to the documentation https://github.com/Layr-Labs/eigenlayer-contracts/blob/master/docs/core/StrategyManager.md#strategymanager.

```solidity
function exitUndelegationLimbo(
    uint256 middlewareTimesIndex,
    bool withdrawFundsFromEigenLayer
) external onlyWhenNotPaused(PAUSED_WITHDRAW_RESTAKED_ETH) onlyNotFrozen(msg.sender) nonReentrant {
    ...

    // enforce minimum delay lag
    require(
        limboStartBlock + strategyManager.withdrawalDelayBlocks() <= block.number,  // @audit
        "EigenPodManager.exitUndelegationLimbo: withdrawalDelayBlocks period has not yet passed"
    );
```

```solidity
function _completeQueuedWithdrawal(
    QueuedWithdrawal calldata queuedWithdrawal,
    IERC20[] calldata tokens,
    uint256 middlewareTimesIndex,
    bool receiveAsTokens
) internal onlyNotFrozen(queuedWithdrawal.delegatedAddress) {
    ...
    // enforce minimum delay lag
    require(
        queuedWithdrawal.withdrawalStartBlock + withdrawalDelayBlocks <= block.number,   // @audit
        "StrategyManager.completeQueuedWithdrawal: withdrawalDelayBlocks period has not yet passed"
    );
```

```solidity
* uint withdrawalDelayBlocks:
  As of M2, this is 50400 (roughly 1 week) // @audit
  Stakers must wait this amount of time before a withdrawal can be completed
```

During **1 week** period staker's funds, including beacon chain staked ETH, and LSTs (cbETH, rETH, stETH), aren't usable on EigenLayer.

Commentary from the client:

“ - This is expected behavior because of how our slashing design will work (eventually). Stakers are expected to delegate to Operators they trust, and they should know that when they stake to an Operator, their funds are also at risk of being slashed should the Operator misbehave.
Force-undelegating a Staker is a part of this arrangement. We expect that Operators who do not live up to community standards (i.e. griefing stakers by making them wait in a withdrawal queue for no reason) will not be delegated to..”

EIG-19. THE EIGENPOD BALANCE UPDATE FUNCTION HAS TO BE PERMISSIONED

SEVERITY: Medium

PATH: EigenPod.sol:verifyBalanceUpdate:L193-274

REMEDIATION: the verifyBalanceUpdate() function should be permissioned

STATUS: fixed

DESCRIPTION:

The **verifyBalanceUpdate()** function is permissionless and, therefore, could be called by anyone with valid proof of a validator's current balance on the beacon chain. Resulting in the adjustment of a pod owner's shares.

Whilst withdrawals are processed by **verifyAndProcessWithdrawals()** against historical summaries https://github.com/Layr-Labs/eigenlayer-contracts/blob/master/docs/core/proofs/BeaconChainProofs.md#beaconchainproofsverifywithdrawal. They could be generated with a time lag of **8192 slots** or ~ **27 hours**.

Consider the following scenario when a Pod owner withdrew a fraction of their validators. In an unlucky case, withdrawal slots of validators are close, and are at the beginning of the **8192 slot** window. So the pod owner have to wait one day before they can call the **verifyAndProcessWithdrawals()**.

In the meantime, someone with malicious intent could call **verifyBalanceUpdate** before the pod owner processes the withdrawal. And so the balances of those validators will be set to **0**, and the pod owner's shares will be massively decreased.

```
function verifyBalanceUpdate(
    uint64 oracleTimestamp,
    uint40 validatorIndex,
    BeaconChainProofs.StateRootProof calldata stateRootProof,
    BeaconChainProofs.BalanceUpdateProof calldata balanceUpdateProof,
    bytes32[] calldata validatorFields
) external onlyWhenNotPaused(PAUSED_EIGENPODS_VERIFY_BALANCE_UPDATE) {  // @audit permissionless
...
}
```

EIG-1. CUSTOM ERRORS

SEVERITY: Low

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

In each contract the validation checks are performed using the **require** function with a reason string.

```
  modifier onlyEigenPodManager() {
      require(msg.sender == address(eigenPodManager), "EigenPod.onlyEigenPodManager: not
eigenPodManager");
      _;
  }
```

We would recommend to replace these with custom errors. This should be done by flipping the check.

For example:

```
require(X == Y, "X is not Y");
```

becomes

```
error XnotY(uint, uint);

if (X != Y)
  revert XnotY(X, Y);
```

The usage of custom errors will save a lot of gas during deployment as well as save on code bytesize of the contract. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

EIG-13. CREATION OF PODS CAN SUFFER DENIAL OF SERVICE

SEVERITY: Low

PATH: EigenPodManager.sol:_deployPod:L378-396

REMEDIATION: remove the cap on the number of pods as it does not seem to have any actual purpose

STATUS: acknowledged

DESCRIPTION:

The function to deploy a pod uses a counter **numPods** to check the total amount of pods against the **maxPods** configuration variable.

An attacker can create as many pods as needed in order to reach maxPods and prevent anyone else to create more pods or stake.

maxPods can be increased, however, the attacker can always quickly increase **numPods** with the creation of more pods for as long as **maxPods** low enough. Setting **maxPods** to a very high value would also negate the reason for its existence in such context.

```
function _deployPod() internal onlyWhenNotPaused(PAUSED_NEW_EIGENPODS) returns (IEigenPod) {
    // check that the limit of EigenPods has not been hit, and increment the EigenPod count
    require(numPods + 1 <= maxPods, "EigenPodManager._deployPod: pod limit reached");
    ++numPods;
    // create the pod
    IEigenPod pod = IEigenPod(
        Create2.deploy(
            0,
            bytes32(uint256(uint160(msg.sender))),
            // set the beacon address to the eigenPodBeacon and initialize it
            abi.encodePacked(beaconProxyBytecode, abi.encode(eigenPodBeacon, ""))
        )
    );
    pod.initialize(msg.sender);
    // store the pod in the mapping
    ownerToPod[msg.sender] = pod;
    emit PodDeployed(address(pod), msg.sender);
    return pod;
}
```

EIG-3. ONLY CONSTANTS SHOULD USE UPPERCASE

SEVERITY: Informational

REMEDIATION: follow Solidity documentation and only use uppercase for constant variable, while for immutable variables we suggest using mixedCase

STATUS: acknowledged

DESCRIPTION:

According to Solidity documentation: "Constants should be named with all capital letters with underscores separating words. Examples: **MAX_BLOCKS**, **TOKEN_NAME**, **TOKEN_TICKER**, **CONTRACT_VERSION**." While local and state variable names should use mixedCase "Examples: **totalSupply**, **remainingSupply**, **balancesOf**, **creatorAddress**, **isPreSale**, **tokenExchangeRate**."

On the same note, it should be noted there is an inconsistency how immutable variables are being named. Some are being named using uppercase, while others using mixedCase. This inconsistency hurts readability

```
IDelayedWithdrawalRouter public immutable delayedWithdrawalRouter;

/// @notice The single EigenPodManager for EigenLayer
IEigenPodManager public immutable eigenPodManager;

///@notice The maximum amount of ETH, in gwei, a validator can have staked in the beacon chain
uint64 public immutable MAX_VALIDATOR_BALANCE_GWEI;

/**
 * @notice The value used in our effective restaked balance calculation, to set the
 * amount by which to underestimate the validator's effective balance.
 */
uint64 public immutable RESTAKED_BALANCE_OFFSET_GWEI;
```

EIG-4. EIGENPOD CONSTRUCTOR SHOULD IMPLEMENT BOUNDARIES FOR IMMUTABLE VARIABLES

SEVERITY: Informational

PATH: EigenPod.sol:constructor:L140-153

REMEDIATION: set sensible boundaries in the constructor for _MAX_VALIDATOR_BALANCE_GWEI and _RESTAKED_BALANCE_OFFSET_GWEI to avoid costly deployment mistakes

STATUS: acknowledged

DESCRIPTION:

Currently there are no boundaries for **_MAX_VALIDATOR_BALANCE_GWEI** and **_RESTAKED_BALANCE_OFFSET_GWEI**. Consequently, these variables could be set to an unreasonable amount when deploying the **EigenPod** implementation contract.

If that were to happen it could cause user loss of funds or the EigenPod could suffer denial of service.

```
constructor(
    IETHPOSDeposit _ethPOS,
    IDelayedWithdrawalRouter _delayedWithdrawalRouter,
    IEigenPodManager _eigenPodManager,
    uint64 _MAX_VALIDATOR_BALANCE_GWEI,
    uint64 _RESTAKED_BALANCE_OFFSET_GWEI
) {
    ethPOS = _ethPOS;
    delayedWithdrawalRouter = _delayedWithdrawalRouter;
    eigenPodManager = _eigenPodManager;
    MAX_VALIDATOR_BALANCE_GWEI = _MAX_VALIDATOR_BALANCE_GWEI;
    RESTAKED_BALANCE_OFFSET_GWEI = _RESTAKED_BALANCE_OFFSET_GWEI;
    _disableInitializers();
}
```

EIG-5. MISSING CHECK FOR EQUALITY BETWEEN NEW AND OLD VALUES IN SETTER FUNCTION

SEVERITY: Informational

PATH: DelayedWithdrawalRouter.sol:_setWithdrawalDelayBlocks: L222-229

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

The **_setWithdrawalDelayBlocks** function in the **DelayedWithdrawalRouter.sol** contract does not check whether the new value for **withdrawalDelayBlocks** is equal to the old value. This can lead to unnecessary gas costs and transactions when the same value is set, which doesn't change the state of the contract.

```
function _setWithdrawalDelayBlocks(uint256 newValue) internal {
    require(
        newValue <= MAX_WITHDRAWAL_DELAY_BLOCKS,
        "DelayedWithdrawalRouter._setWithdrawalDelayBlocks: newValue too large"
    );
    emit WithdrawalDelayBlocksSet(withdrawalDelayBlocks, newValue);
    withdrawalDelayBlocks = newValue;
}
```

To address this issue and allow for initialization with a value of 0 (if needed), add a check to ensure that the new value is not equal to the old one (unless the old value is 0) before updating the **withdrawalDelayBlocks** variable.

For example:

```
// Check if the new value is different from the old value (unless old value is 0)
if (newValue != withdrawalDelayBlocks || withdrawalDelayBlocks == 0) {
    emit WithdrawalDelayBlocksSet(withdrawalDelayBlocks, newValue);
    withdrawalDelayBlocks = newValue;
} else {
    revert("DelayedWithdrawalRouter._setWithdrawalDelayBlocks: same value");
}
```

EIG-6. CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: DelayedWithdrawalRouter.sol

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

The **MAX_WITHDRAWAL_DELAY_BLOCKS** parameter on line 27 should be **private**. Setting constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the values outside of where it's used, and won't add another entry to the method ID table. The values can still be read from the verified contract source code if necessary.

```
uint256 public constant MAX_WITHDRAWAL_DELAY_BLOCKS = 50400;
```

EIG-16. INCORRECT DOCUMENTATION

SEVERITY: Informational

PATH: EigenPod.sol:verifyAndProcessWithdrawals:L322

REMEDIATION: change strategy manager to eigenPodManager

STATUS: fixed

DESCRIPTION:

In the EigenPod contract on line 322, the documentation string does not agree with the implementation.

```
    //update podOwner's shares in the strategy manager
    if (withdrawalSummary.sharesDelta != 0) {
        eigenPodManager.recordBeaconChainETHBalanceUpdate(podOwner,
withdrawalSummary.sharesDelta);
    }
```

hexens