2023-04-eigenlayer (EigenLayer)

Benchmark Security Review Performed by Bug Hunter

18.05.2025

Table of Contents

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

This security-review report was generated with Truscova's Bug Hunter tool to benchmark its performance on selected Yield Farming projects. It is made available through the Bug Hunter dashboard and relevant Git repositories. Users may choose to share or publish this report at their own discretion.

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project 2023-04-eigenlayer was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/code-423n4/2023-04-eigenlayer
  Commit hash: `5e4872358cd2bda1936c29f460ece2308af4def6`

Bug Hunter reviewed the following files:

StrategyManager.sol StrategyManagerStorage.sol StrategyBase.sol EigenPodManager.sol EigenPod.sol EigenPodPausingConstants.sol DelayedWithdrawalRouter.sol Pausable.sol PauserRegistry.sol BeaconChainProofs.sol Merkle.sol Endian.sol

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	1
Medium	1
Low	1

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
18.05.2025	Project submitted
18.05.2025	Security review concluded

Detailed Findings

1. merkleizeSha256 May Revert or Return Incorrect Value for Invalid Input Sizes

Severity	Low
Finding ID	BH-2023-04-eigenlayer-01
Target	`src/contracts/libraries/Merkle.sol`
Function name	`merkleizeSha256`

Description

The `merkleizeSha256` function assumes that the number of leaves is a power of two, as it performs pairwise hashing without padding or validation. If the input `leaves` array has a length that is not a power of two, the function may revert due to out-of-bounds access (e.g., accessing `leaves[2*i+1]` when it doesn't exist), or it may silently compute an incorrect Merkle root.

```
function merkleizeSha256(
    bytes32[] memory leaves
) internal pure returns (bytes32) {
    //there are half as many nodes in the layer above the leaves
    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], leaves[2*i+1]));
    }
    //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 with the pairwise
hashes of their children
        for (uint i = 0; i < numNodesInLayer; i++) {
            layer[i] = sha256(abi.encodePacked(layer[2*i], layer[2*i+1]));
        }
```

```
        //the next layer above has half as many nodes
        numNodesInLayer /= 2;
    }
    //the first node in the layer is the root
    return layer[0];
}
```

Recommendation

Add an explicit check that `leaves.length` is greater than zero and a power of two. Alternatively, implement padding logic (e.g., duplicate the last leaf) to safely handle odd-length inputs, consistent with common Merkle tree implementations.

2. Malicious Strategy Can Permanently DoS Pending Withdrawals

Severity	Medium
Finding ID	BH-2023-04-eigenlayer-02
Target	`src/contracts/core/StrategyManager.sol`
Function name	`queueWithdrawal`

Description

The `queueWithdrawal` function in `StrategyManager.sol` allows users to queue withdrawals that include arbitrary strategy contracts. However, there is no validation of strategy behavior during withdrawal processing (e.g., in `completeQueuedWithdrawal` or similar). A malicious strategy can behave adversarially during withdrawal processing — for example, by reverting, consuming excess gas, or otherwise preventing execution — leading to a permanent denial-of-service (DoS) for all pending withdrawals that include that strategy. Since these withdrawals are stored by root and treated as a unit, a single malicious strategy can effectively lock all included user funds indefinitely.

```
function queueWithdrawal(
        uint256[] calldata strategyIndexes,
        IStrategy[] calldata strategies,
        uint256[] calldata shares,
        address withdrawer,
        bool undelegateIfPossible
    )
        external
        onlyWhenNotPaused(PAUSED_WITHDRAWALS)
        onlyNotFrozen(msg.sender)
        nonReentrant
        returns (bytes32)
    {
        require(strategies.length == shares.length,
"StrategyManager.queueWithdrawal: input length mismatch");
        require(withdrawer != address(0), "StrategyManager.queueWithdrawal: cannot
withdraw to zero address");

        // modify delegated shares accordingly, if applicable
        delegation.decreaseDelegatedShares(msg.sender, strategies, shares);
```

```
        uint96 nonce = uint96(numWithdrawalsQueued[msg.sender]);

        // keeps track of the current index in the `strategyIndexes` array
        uint256 strategyIndexIndex;

        /**
         * Ensure that if the withdrawal includes beacon chain ETH, the specified
'withdrawer' is not different than the caller.
         * This is because shares in the enshrined `beaconChainETHStrategy`
ultimately represent tokens in **non-fungible** EigenPods,
         * while other share in all other strategies represent purely fungible
positions.
         */
        for (uint256 i = 0; i < strategies.length;) {
            if (strategies[i] == beaconChainETHStrategy) {
                require(withdrawer == msg.sender,
                    "StrategyManager.queueWithdrawal: cannot queue a withdrawal of
Beacon Chain ETH to a different address");
                require(strategies.length == 1,
                    "StrategyManager.queueWithdrawal: cannot queue a withdrawal
including Beacon Chain ETH and other tokens");
                require(shares[i] % GWEI_TO_WEI == 0,
                    "StrategyManager.queueWithdrawal: cannot queue a withdrawal of
Beacon Chain ETH for an non-whole amount of gwei");
            }

            // the internal function will return 'true' in the event the strategy
was
            // removed from the depositor's array of strategies -- i.e.
stakerStrategyList[depositor]
            if (_removeShares(msg.sender, strategyIndexes[strategyIndexIndex],
strategies[i], shares[i])) {
                unchecked {
                    ++strategyIndexIndex;
                }
            }

            emit ShareWithdrawalQueued(msg.sender, nonce, strategies[i], shares[i]);

            //increment the loop
            unchecked {
                ++i;
            }
        }
```

```
        // fetch the address that the `msg.sender` is delegated to
        address delegatedAddress = delegation.delegatedTo(msg.sender);

        QueuedWithdrawal memory queuedWithdrawal;

        {
            WithdrawerAndNonce memory withdrawerAndNonce = WithdrawerAndNonce({
                withdrawer: withdrawer,
                nonce: nonce
            });
            // increment the numWithdrawalsQueued of the sender
            unchecked {
                numWithdrawalsQueued[msg.sender] = nonce + 1;
            }

            // copy arguments into struct and pull delegation info
            queuedWithdrawal = QueuedWithdrawal({
                strategies: strategies,
                shares: shares,
                depositor: msg.sender,
                withdrawerAndNonce: withdrawerAndNonce,
                withdrawalStartBlock: uint32(block.number),
                delegatedAddress: delegatedAddress
            });

        }

        // calculate the withdrawal root
        bytes32 withdrawalRoot = calculateWithdrawalRoot(queuedWithdrawal);

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

        // If the `msg.sender` has withdrawn all of their funds from EigenLayer in
this transaction, then they can choose to also undelegate
        /**
         * Checking that `stakerStrategyList[msg.sender].length == 0` is not
strictly necessary here, but prevents reverting very late in logic,
         * in the case that 'undelegate' is set to true but the `msg.sender` still
has active deposits in EigenLayer.
         */
        if (undelegateIfPossible && stakerStrategyList[msg.sender].length == 0) {
            _undelegate(msg.sender);
```

```
        }

        emit WithdrawalQueued(msg.sender, nonce, withdrawer, delegatedAddress,
withdrawalRoot);

        return withdrawalRoot;
    }
```

Recommendation

Introduce a strategy allowlist to restrict withdrawal interactions to audited or trusted strategy contracts. Alternatively, implement a mechanism to isolate or skip over failing strategies during withdrawal processing and refund their portion separately or mark them as failed for retry later.

3. verifyAndProcessWithdrawal Accepts Forged Slot and Block Number Proofs

Severity	High
Finding ID	BH-2023-04-eigenlayer-03
Target	`src/contracts/pods/EigenPod.sol`
Function name	`verifyAndProcessWithdrawal`

Description

The function `verifyAndProcessWithdrawal` in `EigenPod.sol` indirectly accepts empty or invalid Merkle proofs for `slotProof` and `blockNumberProof` due to its reliance on `Merkle.verifyInclusionSha256`, which returns true when `proof.length < 32` and `leaf == root`. This allows an attacker to forge `slotRoot` and `blockNumberRoot` by making them equal to their respective parent roots and submitting empty proofs. Consequently, an attacker can manipulate the slot and block number used for validator status checks, potentially enabling unauthorized or repeated withdrawals.

```
function verifyAndProcessWithdrawal(
        BeaconChainProofs.WithdrawalProofs calldata withdrawalProofs,
        bytes calldata validatorFieldsProof,
        bytes32[] calldata validatorFields,
        bytes32[] calldata withdrawalFields,
        uint256 beaconChainETHStrategyIndex,
        uint64 oracleBlockNumber
    )
        external
        onlyWhenNotPaused(PAUSED_EIGENPODS_VERIFY_WITHDRAWAL)
        onlyNotFrozen
        /**
         * Check that the provided block number being proven against is after the
`mostRecentWithdrawalBlockNumber`.
         * Without this check, there is an edge case where a user proves a past
withdrawal for a validator whose funds they already withdrew,
         * as a way to "withdraw the same funds twice" without providing adequate
proof.
         * Note that this check is not made using the oracleBlockNumber as in the
`verifyWithdrawalCredentials` proof; instead this proof
         * proof is made for the block number of the withdrawal, which may be within
```

```
8192 slots of the oracleBlockNumber.
         * This difference in modifier usage is OK, since it is still not possible
to `verifyAndProcessWithdrawal` against a slot that occurred
         * *prior* to the proof provided in the `verifyWithdrawalCredentials`
function.
         */

proofIsForValidBlockNumber(Endian.fromLittleEndianUint64(withdrawalProofs.blockNumberRoot))
    {
        /**
         * If the validator status is inactive, then withdrawal credentials were
never verified for the validator,
         * and thus we cannot know that the validator is related to this EigenPod at
all!
         */
        uint40 validatorIndex =
uint40(Endian.fromLittleEndianUint64(withdrawalFields[BeaconChainProofs.WITHDRAWAL_VALIDATOR_INDEX

        require(validatorStatus[validatorIndex] != VALIDATOR_STATUS.INACTIVE,
            "EigenPod.verifyOvercommittedStake: Validator never proven to have
withdrawal credentials pointed to this contract");

        // fetch the beacon state root for the specified block
        bytes32 beaconStateRoot =
eigenPodManager.getBeaconChainStateRoot(oracleBlockNumber);

        // Verifying the withdrawal as well as the slot
        BeaconChainProofs.verifyWithdrawalProofs(beaconStateRoot, withdrawalProofs,
withdrawalFields);
        // Verifying the validator fields, specifically the withdrawable epoch
        BeaconChainProofs.verifyValidatorFields(validatorIndex, beaconStateRoot,
validatorFieldsProof, validatorFields);

        uint64 withdrawalAmountGwei =
Endian.fromLittleEndianUint64(withdrawalFields[BeaconChainProofs.WITHDRAWAL_VALIDATOR_AMOUNT_INDEX

        //check if the withdrawal occured after mostRecentWithdrawalBlockNumber
        uint64 slot = Endian.fromLittleEndianUint64(withdrawalProofs.slotRoot);

        /**
         * if the validator's withdrawable epoch is less than or equal to the slot's
epoch, then the validator has fully withdrawn because
         * a full withdrawal is only processable after the withdrawable epoch has
passed.
```

```
         */
        // reference: uint64 withdrawableEpoch =
Endian.fromLittleEndianUint64(validatorFields[BeaconChainProofs.VALIDATOR_WITHDRAWABLE_EPOCH_INDEX
        if
(Endian.fromLittleEndianUint64(validatorFields[BeaconChainProofs.VALIDATOR_WITHDRAWABLE_EPOCH_INDE
<= slot/BeaconChainProofs.SLOTS_PER_EPOCH) {
            _processFullWithdrawal(withdrawalAmountGwei, validatorIndex,
beaconChainETHStrategyIndex, podOwner, validatorStatus[validatorIndex]);
        } else {
            _processPartialWithdrawal(slot, withdrawalAmountGwei, validatorIndex,
podOwner);
        }
    }
```

Recommendation

Introduce explicit checks that enforce a minimum length for `withdrawalProofs.slotProof` and `withdrawalProofs.blockNumberProof` (e.g., `require(proof.length >= 32)`), or implement such validation inside `verifyInclusionSha256` to prevent bypasses across the entire Merkle proof verification system.