EigenLayer

EigenLayer

Smart Contract Security Review #2

Version: 1.0

May, 2023

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the EigenLayer smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the EigenLayer smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the EigenLayer smart contracts.

Overview

EigenLayer is a restaking service on the Ethereum mainnet that utilizes already staked assets as collateral to secure new services. Assets which have already been staked, such as an Ethereum validator's ETH, can be placed under the control of the EigenLayer smart contracts to act as stake securing additional services such as rollups, bridges or Dapps.

The purpose of Eigenlayer is to provide a generalised permissionless platform by which these stakers can be connected to middlewares in need of securing their services. In return for securing a system, stakers are paid additional yield on top of their existing staking rewards.

Security Assessment Summary

This review was conducted on the files hosted on the EigenLayer repository and were assessed at commit 2500148f.

A previous review of the codebase was undertaken in Q1 of 2023.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 6 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Low: 1 issue.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the EigenLayer smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
EGN2-01	Middleware can Deny Withdrawls by Revoking Slashing Prior to Queueing Withdrawal	High	Open
EGN2-02	Domain Seperator not Recalculated in Case of a Hard Fork	Low	Open
EGN2-03	Delayed Withdrawals can be Created During Paused State	Informational	Open
EGN2-04	Funds can be Lost Due to SelfDestructed Staker Contract	Informational	Open
EGN2-05	Novel Staking Risks Posed by Middleware	Informational	Open
EGN2-06	Miscellaneous General Comments	Informational	Open

EGN2-01	Middleware can Deny Withdrawls by Revoking Slashing Prior to Queueing Withdrawal		
Asset	`Slasher.sol, StrategyManager.sol`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `slasher.sol` contract manages middleware rights and capabilities for a given operator, whilst the `StrategyManager.sol` relies on this contract to determine if withdrawals are allowed. If a given slasher (middleware) revokes their own rights to slash for an `operator` using `Slasher.recordLastStakeUpdateAndRevokeSlashingAbility()`, a staker will be unable to queue withdrawals. A condition for this edge case is that a staker maintains only one middleware.

The `StrategyManager.queueWithdrawal()` leverages the function modifier `onlyNotFrozen(msg.sender)` to determine if a user is able to initiate a withdrawal. Rejecting withdrawals is managed by updating the `frozenStatus[operator]` and can only be done by an approved middleware that the operator has opted into slashing. Once a withdrawal is queued, a staker can complete the withdrawal by calling `StrategyManager.completeQueuedWithdrawl()`.

The `StrategyManager._completeQueuedWithdrawal()` checks the `Slasher.canWithdraw()` which requires the following conditions to be met:

1. `withdrawalStartBlock < update.stalestUpdateBlock`: requiring the withdrawal to be initiated prior the latest update from the slasher.
2. `uint32(block.timestamp) > update.latestServeUntil`: requiring the withdrawal to be completed after the slasher's serving period has elapsed.

These two conditions are valid and required for the system to function under normal circumstances. However, in the event that a middleware operates maliciously, or accidentally revokes their slashing ability by calling `Slasher.recordLastStakeUpdateAndRevokeSlashingAbility()` prior to a withdrawal being queued, condition (1) above will fail. Effectively preventing the completion of queued withdrawals.

Recommendations

The resolution discussed with the development team involves only validating condition (2) when there is a single approved middleware for a given operator.

EGN2-02	Domain Seperator not Recalculated in Case of a Hard Fork		
Asset	`DelegationManager.sol, StrategyManager.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `DOMAIN_SEPARATOR` in `DelegationManager` and `StrategyManager` is set during initialisation and will not change after the contract is initialised. This feature is used to prevent replay attacks from impacting users providing SECP256k1 signatures across different EVM compatible chains.

However, if a hard fork occurs after the contract deployment, the `DOMAIN_SEPARATOR` would become invalid on one of the forked chains as the `block.chainid` will be different to the one used in the initialiser.

The user may try to course correct by providing the old `block.chainid` in their `digestHash` when message signing. The submitted signature can then be reused on both chains provided the staker's `nonce` has not changed in the parent chain. This could lead to a user being compromised with unexpected or unwanted delegation or strategy decisions being made in the parent chain on their behalf.

To summarise any message signed on one fork will be valid on both forks.

Recommendations

A solution is to add a `getDomainSeperator()` function which will calculate and return the correct domain separator dynamically in case of a hardfork. That is if the `block.chainid` is different to that used in `DOMAIN_SEPARATOR` then the `DOMAIN_SEPARATOR` needs to be recalculated with the current `block.chainid`.

A possible implementation of a solution can be found in here.

EGN2-03	Delayed Withdrawals can be Created During Paused State
Asset	`DelayedWithdrawalRouter.sol`
Status	**Open**
Rating	Informational

Description

The `claimDelayedWithdrawals()` function, which is used to claim delayed withdrawals, can be paused using the `onlyWhenNotPaused(PAUSED_DELAYED_WITHDRAWAL_CLAIMS)` modifier. However, this modifier is not present on the `createDelayedWithdrawal()` function.

This can lead to a scenario in which, if the `DelayedWithdrawalRouter` contract is paused, `EigenPods` can still create delayed withdrawals that are recorded using the `block.time` . Therefore, users can wait out the withdrawal delay period during the paused state, potentially allowing them to withdraw funds immediately after the contract is unpaused, rather than waiting for the entire withdrawal delay period.

The impact is a user may create a pending withdrawal, if `PAUSED_NEW_FREEZING` is also set, they would not be slashable until the contracts are unpaused. This is related to the off-chain pausing logic determining which functions may be simultaneously paused.

Recommendations

Consider adding the `onlyWhenNotPaused` modifier to the `createDelayedWithdrawal()` function to prevent the creation of delayed withdrawals while the contract is in a paused state.

Additionally, it is recommended to have a predetermined logic table off-chain which states the functions that may be paused simultaneously.

EGN2-04	Funds can be Lost Due to SelfDestructed Staker Contract
Asset	`StrategyManager.sol`
Status	Open
Rating	Informational

Description

A contract can be passed into `StrategyManager.depositIntoStrategyWithSignature()` as the `staker` address, this contract will then be responsible for verifying signatures using ERC1271. However an edge case exists that may deny withdrawal transactions from being created, due to the the requirement of `msg.sender` in `StrategyManager.queueWithdrawal()`.

Checks performed on line [277] rely on the OpenZeppelin's `Address` library to validate if the staker is a contract. In the edge case, involving a single transaction, where the staker's contract calls the `SELFDESTRUCT` opcode prior to calling the function `StrategyManager.depositIntoStrategyWithSignature()`, the `Address.isContract()` will return `true`.

This would result in the check on line [277] passing, and since the contract's destruction is only acted on at the end of the transaction, the call to `IERC1271(staker).isValidSignature(digestHash, signature)` should pass. This will lead to a valid deposit being made, with no way to extract the funds back out (as the contract bytecode is destroyed after the transaction is finalised).

Recommendations

Several approaches for solving this issue may be applicable. Providing a backup withdrawal address when using signature deposits, could allow a user to exit if assigned staker contract proves incapable or unwilling to cooperate (as in the aforementioned edge case). Alternatively, documentation that clearly directs implementation of a staker contract to avoid the use of any `SELFDESTRUCT` opcodes.

EGN2-05	Novel Staking Risks Posed by Middleware
Asset	`Slasher.sol`
Status	**Open**
Rating	Informational

Description

Middleware (as named by EigenLayer) poses a unique value proposition within the liquid staking derivative ecosystem. Their premise is that middlewares can leverage existing stake assigned to a single operator, allowing multiple different protocols to leverage the same stake to secure their network, protocol or application. With novel value propositions comes the reasonably novel risk to staking protocols of 'cross-protocol security implications'. Previously calculated staking risks, and therefore relevant rewards have only accounted for risks to the platform that staking is occuring on. With EigenLayer's middleware contracts, situations may arise that may previously have been unprofitable that are now profitable. Essentially, protocol risk becomes entangled and compounded within subsets of other protocols.

For instance, knowing the slashing penalties involved, it may previously have been unprofitable to commit an attestation or proposal violation on ethereum. However, as we enter a state where multiple protocols and applications (all providing their own rewards) leverage the same stake, an operator can potentially violate all protocols and risk being slashed if the rewards potentially outweigh the slashing penalty risk.

Stakers, Operators and Middleware alike are all impacted in various ways through compouding risks of operators adding multiple 'unsafe' or even 'malicious' middlewares to their whitelist. One middleware could hold a group of other middlewares ransom, by threatening or inacting slashing of its operators if compromising the security of other middlewares can yield more reward than the cost of ruining their own network.

Recommendations

The testing team acknowledges that EigenLayer does not aim to enforce the use of secure middleware on any Operator. However, the testing team recommends the following actions;

1. Documentation should clearly state the risks of using middleware for operators
2. Documentation should clearly state the risks of too many middlewares using the same subset of operators
3. Potentially adding functionality that allows middlewares to dictate which operators match their risk profile. For instance it could be deemed unsafe for one protocol to use a potentially high risk set of operators using an extremely large set of middlewares.

EGN2-06	Miscellaneous General Comments
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Use of Magic Numbers** Constants should be defined rather than using magic numbers. The number `50400` is used for `MAX_WITHDRAWAL_DELAY_BLOCKS`.
   - StrategyManagerStorage line [**46**]
   - DelayedWithdrawRouter line [**24**]

   `7 days / 12 seconds` should be used instead.
2. **Lack of zero-address Check** The `recipient` address in `DelayedWithdrawRouter.createDelayedWithdrawal()` is missing a zero-address check.
3. **Redundant Code** The `if` statement, in `EigenPod.sol` on line [**218**] is redundant. `if (!hasRestaked)`
4. **Project Files with Outdated Naming Conventions** The `package.json` file in the project root directory still references `eigenlayr-contracts` instead of the new branding of `eigenlayer-contracts`

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
Running 9 tests for test/Slasher.t.sol:SlasherTests
[PASS] testFreezeOperator() (gas: 172388)
[PASS] testOnlyCanSlash(address,uint32) (runs: 256, : 127341, ~: 127341)
[PASS] testOptIn(address,address) (runs: 256, : 163425, ~: 163433)
[PASS] testOrderingRecordStakeUpdateVuln() (gas: 299149)
[PASS] testRecordFirstStakeUpdate() (gas: 385598)
[PASS] testRecordLastStakeUpdateAndRevokeSlashingAbility() (gas: 252770)
[PASS] testRecordStakeUpdate() (gas: 396826)
[PASS] testRecursiveCallRevert() (gas: 470736)
[PASS] testResetFrozenOperator(address) (runs: 256, : 166095, ~: 166095)
Test result: ok. 9 passed; 0 failed; finished in 286.99ms

Running 2 tests for test/DelayedWithdrawalRouterUnit.t.sol:DelayedWithdrawalRouterUnitTests
[PASS] testClaimableUserDelayedWithdrawals(uint8,uint8,uint256,address,bool) (runs: 256, : 536006, ~: 340670)
[PASS] testCreateDelayedWithdrawalNonzeroAmount(uint224,address,address) (runs: 256, : 91780, ~: 91780)
Test result: ok. 2 passed; 0 failed; finished in 480.56ms

Running 9 tests for test/InvestmentManager.t.sol:InvestmentTests
[PASS] testDepositEigen(uint96) (runs: 256, : 1653331, ~: 1653331)
[PASS] testDepositNonexistentStrategy(address) (runs: 256, : 764944, ~: 764944)
[PASS] testDepositStrategies(uint8) (runs: 256, : 4871168, ~: 2981402)
[PASS] testDepositUnsupportedToken() (gas: 834995)
[PASS] testFrontrunFirstDepositor() (gas: 449810)
[PASS] testPreventSlashing() (gas: 2026089)
[PASS] testRevertOnZeroDeposit() (gas: 140749)
[PASS] testWethDeposit(uint256):(uint256) (runs: 256, : 1654967, ~: 1654967)
[PASS] testWithdrawalSequences() (gas: 2294671)
Test result: ok. 9 passed; 0 failed; finished in 1.87s

Running 15 tests for test/Delegation.t.sol:DelegationTests
[PASS] testCannotInitMultipleTimesDelegation(address) (runs: 256, : 26459, ~: 26459)
[PASS] testCannotRegisterAsOperatorTwice(address,address) (runs: 256, : 104595, ~: 104209)
[PASS] testCannotSetDelegationTermsZeroAddress() (gas: 23505)
[PASS] testDelegateToByInvalidSignature(address,uint96,uint96,uint8,bytes32,bytes32) (runs: 256, : 608186, ~: 608223)
[PASS] testDelegateToBySignature(address,uint96,uint96,uint256) (runs: 256, : 693891, ~: 702043)
[PASS] testDelegateToInvalidOperator(address,address) (runs: 256, : 43275, ~: 43275)
[PASS] testDelegation(address,address,uint96,uint96) (runs: 256, : 746180, ~: 746180)
[PASS] testDelegationMultipleStrategies(uint8,address,address) (runs: 256, : 4913771, ~: 2977934)
[PASS] testDelegationReceived(address,address) (runs: 256, : 1026338, ~: 1026356)
[PASS] testDelegationToUnregisteredDelegate(address) (runs: 256, : 2735155, ~: 2735155)
[PASS] testRegisterAsOperatorMultipleTimes(address) (runs: 256, : 115218, ~: 115212)
[PASS] testSelfOperatorDelegate(address) (runs: 256, : 105381, ~: 105381)
[PASS] testSelfOperatorRegister() (gas: 2021835)
[PASS] testTwoSelfOperatorsRegister() (gas: 2559719)
[PASS] testUndelegate(address,address,address) (runs: 256, : 148356, ~: 148275)
Test result: ok. 15 passed; 0 failed; finished in 3.15s

Running 24 tests for test/EigenPods.t.sol:EigenPodTests
[PASS] testAttemptedWithdrawalAfterVerifyingWithdrawalCredentials() (gas: 13924205)
[PASS] testDependancyChangesGetPod() (gas: 1906960)
[PASS] testDeployAndVerifyNewEigenPod():(address) (gas: 13917973)
[PASS] testDeployNewEigenPodWithActiveValidator() (gas: 24054549)
[PASS] testDeployNewEigenPodWithWrongWithdrawalCreds(address) (runs: 256, : 13478840, ~: 13478840)
[PASS] testDeployingEigenPodRevertsWhenPaused() (gas: 80530)
[PASS] testDoubleFullWithdrawal() (gas: 45905645)
[PASS] testFullWithdrawalFlow():(address) (gas: 29372221)
[PASS] testFullWithdrawalProof() (gas: 13943800)
[PASS] testPartialWithdrawalFlow():(address) (gas: 29533489)
[PASS] testProveOverComittedStakeOnWithdrawnValidator() (gas: 24163728)
[PASS] testProveOverCommittedBalance() (gas: 24453704)
[PASS] testProveSingleWithdrawalCredential() (gas: 14015502)
```

```
[PASS] testProvingMultipleWithdrawalsForSameSlot() (gas: 46099625)
[PASS] testStake(bytes,bytes,bytes32) (runs: 256, : 1035294, ~: 1035141)
[PASS] testStaking() (gas: 382058)
[PASS] testUpdateSlashedBeaconBalance() (gas: 24327586)
[PASS] testVerifyCorrectWithdrawalCredentialsRevertsWhenPaused() (gas: 13705093)
[PASS] testVerifyOvercommittedStakeRevertsWhenPaused() (gas: 24379170)
[PASS] testVerifyWithdrawalCredentialsWithInadequateBalance() (gas: 13674534)
[PASS] testWithdrawBeforeRestaking() (gas: 466612)
[PASS] testWithdrawBeforeRestakingAfterRestaking() (gas: 13902048)
[PASS] testWithdrawFromPod() (gas: 459881)
[PASS] testWithdrawRestakedBeaconChainETHRevertsWhenPaused() (gas: 59772)
Test result: ok. 24 passed; 0 failed; finished in 5.39s
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].