CANTINA

Eigenlayer contracts

Competition

April 5, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

EigenLayer is a protocol built on Ethereum that introduces restaking, a new primitive in cryptoeconomic security.

From Feb 27th to Mar 18th Cantina hosted a competition based on eigenlayer-contracts. The participants identified a total of **71** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 2
- Low Risk: 14
- Gas Optimizations: 1
- Informational: 53

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Beacon chain withdrawals that occur at `lastwithdrawaltimestamp` will be lost

Submitted by hash

Severity: High Risk

Context: EigenPod.sol#L119-L126, EigenPod.sol#L381-L391, EigenPod.sol#L566-L583, EigenPod.sol#L733-L737

Description: ETH withdrawn from Beacon Chain is withdrawn from the EigenPod by calling the `withdrawBeforeRestaking` function before restaking is activated. To activate restaking, the `activateRestaking` function is called, which internally calls `_processWithdrawalBeforeRestaking` with the idea that all withdrawals until this timestamp will be processed. Hence, the `verifyAndProcessWithdrawals` function need only be called on timestamps greater than `mostRecentWithdrawalTimestamp` i.e. the timestamp in which the last withdrawal via `_processWithdrawalBeforeRestaking` occurred.

```
function _verifyAndProcessWithdrawal(
    bytes32 beaconStateRoot,
    BeaconChainProofs.WithdrawalProof calldata withdrawalProof,
    bytes calldata validatorFieldsProof,
    bytes32[] calldata validatorFields,
    bytes32[] calldata withdrawalFields
)
    internal
    proofIsForValidTimestamp(withdrawalProof.getWithdrawalTimestamp())
    returns (VerifiedWithdrawal memory)
{

modifier proofIsForValidTimestamp(uint64 timestamp) {
    require(
        timestamp > mostRecentWithdrawalTimestamp,
        "EigenPod.proofIsForValidTimestamp: beacon chain proof must be for timestamp after
↪  mostRecentWithdrawalTimestamp"
    );
    _;
}
```

```
function activateRestaking()
    external
    onlyWhenNotPaused(PAUSED_EIGENPODS_VERIFY_CREDENTIALS)
    onlyEigenPodOwner
    hasNeverRestaked
{
    hasRestaked = true;
    _processWithdrawalBeforeRestaking(podOwner);


    emit RestakingActivated(podOwner);
}

function _processWithdrawalBeforeRestaking(address _podOwner) internal {
    mostRecentWithdrawalTimestamp = uint32(block.timestamp);
    nonBeaconChainETHBalanceWei = 0;
    _sendETH_AsDelayedWithdrawal(_podOwner, address(this).balance);
}
```

In case there is a withdrawal from the beacon chain that occurred in `mostRecentWithdrawalTimestamp`, this amount will be lost since the withdrawals from the beacon chain are executed after all the user transactions according to EIP-4895. Hence, when the user executes `activateRestaking` and consequently `_processWithdrawalBeforeRestaking`, the pod will not be having the ETH withdrawn from beacon chain in that block. After this block, the user will not be able to withdraw this ETH via `verifyAndProcessWithdrawals` since `verifyAndProcessWithdrawals` can only be called for timestamps greater than `mostRecentWithdrawalTimestamp`.

Proof of concept:

1. User has a withdrawal of 32 ETH from the beacon chain at time `t`.

2. User calls `activateRestaking` in the timestamp `t`.
3. This is supposed to move out all ETH from the contract. But since the withdrawal execution only happens after all the user's transactions, `address(this)` will not factor this ETH.
4. These 32 ETH cannot be withdrawn using `verifyAndProcessWithdrawals` since the proof timestamp will be equal to the `mostRecentWithdrawalTimestamp` (timestamp in which `acitvateRestaking` is called) and the condition for calling `verifyAndProcessWithdrawals` is `timestamp > mostRecentWithdrawalTimestamp`

Impact: Beacon chain withdrawals will be lost in case any user activates restaking in the block which also contains one of their withdrawal

Recommendation: Use `>= mostRecentWithdrawalTimestamp` instead.

3.2 Medium Risk

3.2.1 External LST compromission leads to avs compromission due to a lack of operators' staked amount cooldown

Submitted by zigtur

Severity: Medium Risk

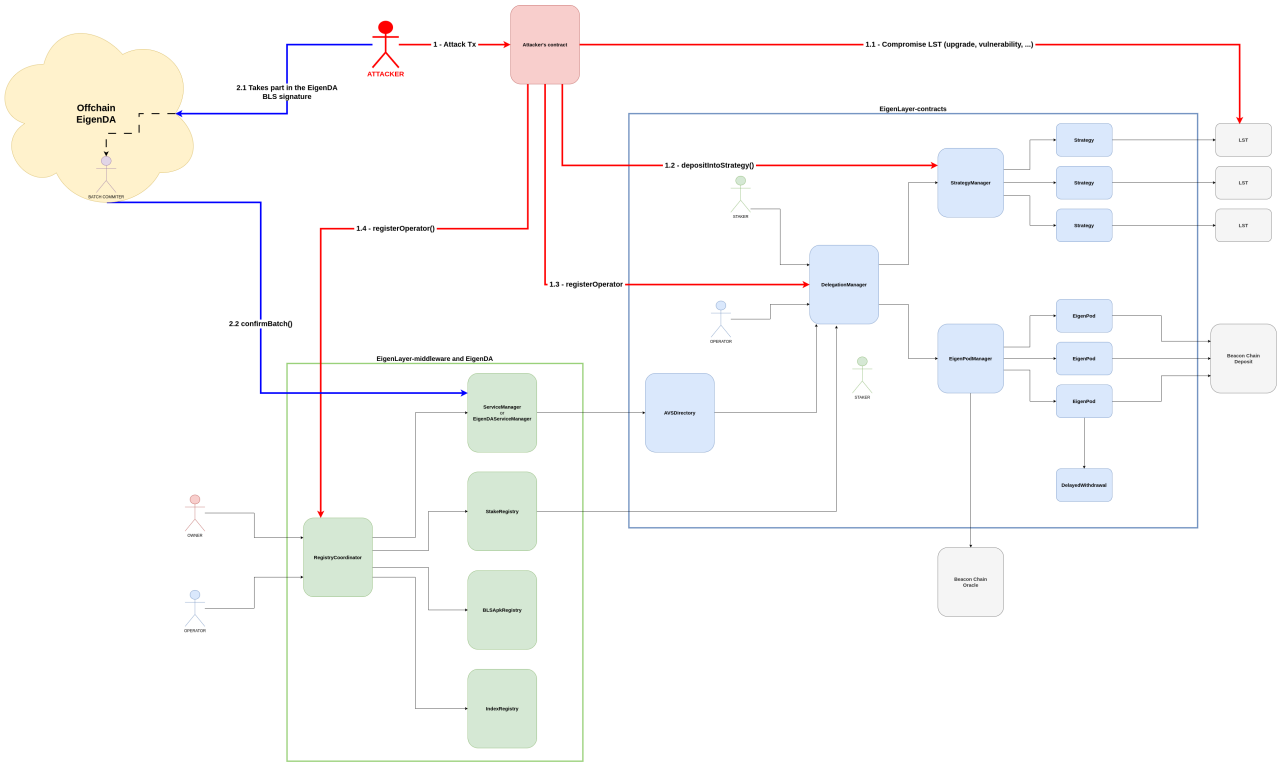
Context: DelegationManager.sol#L97-L112, EigenDAServiceManager.sol#L75, RegistryCoordinator.sol#L128-L151, StrategyManager.sol#L105-L111, StrategyManager.sol#L323-L342

Description: A compromised LST can instantly lead the attacker to break EigenDA and every other AVS that use this LST.

The protocol relies on external LST. Some of these LST may be compromised through a vulnerability (low likelihood) or through a malicious update (medium likelihood).

Through a LST compromission, the attacker is able to compromise any AVS that use this LST. This is due to a lack of cooldown period in the registering process of an operator.

Proof of concept:



When a LST is compromised, **within a single block** the attacker will be able to:

- Mint LST (external party).
- Deposit LST through `StrategyManager.depositIntoStrategy()`.

- Register as operator through `DelegationManager.registerOperator()`(optional: if attacker isn't already operator).
- Register as AVS operator through `RegistryCoordinator.registerOperator()`, with a quorum that uses the LST (optional: if attacker isn't already operator).

Then, they can take part in the offchain AVS process. By looking at `EigenDAServiceManager.confirmBatch`, we can see that the block number previous from the current one (`block.number - 1`) can be taken as reference to confirm the batch.

This means that within two blocks an attacker that compromised an external LST is able to increase their shares, register as operator if needed and take part in the EigenDA offchain process with the increased shares amount (and higher power on the protocol).

This two block period is way too tight for any defensive operation from the protocol (such as pausing).

Attached to this report, the different code location involved in the attack scenario. All these functions have no cooldown mechanism.

Recommendation: The goal of the mitigation is to let time for EigenLayer's team to respond to the emergency by rate-limiting the amount of LST depositable per block.

If an attacker compromised an LST, they still need to call `StrategyBase.deposit` to get their shares increased in EigenLayer. A rate limit at this code location will require more time (several blocks) for the attacker to achieve a high amount of shares.

The following patch is an example of such a fix by using the `_beforeDeposit` hooking function. Please note that the `rateLimitAmountPerBlock` must be well chosen to protect the protocol while being convenient for operators and users.

```
diff --git a/src/contracts/strategies/StrategyBase.sol b/src/contracts/strategies/StrategyBase.sol
index cf31a30f..ad74f9b2 100644
--- a/src/contracts/strategies/StrategyBase.sol
+++ b/src/contracts/strategies/StrategyBase.sol
@@ -55,6 +55,10 @@ contract StrategyBase is Initializable, Pausable, IStrategy {
     /// @notice The total number of extant shares in this Strategy
     uint256 public totalShares;

+    uint256 public lastUpdateBlockNumber;
+    uint256 public lastUpdateAmount;
+    uint256 public rateLimitAmountPerBlock;
+
     /// @notice Simply checks that the `msg.sender` is the `strategyManager`, which is an address stored
↪  immutably at construction.
     modifier onlyStrategyManager() {
         require(msg.sender == address(strategyManager), "StrategyBase.onlyStrategyManager");
@@ -67,8 +71,9 @@ contract StrategyBase is Initializable, Pausable, IStrategy {
         _disableInitializers();
     }

-    function initialize(IERC20 _underlyingToken, IPauserRegistry _pauserRegistry) public virtual initializer {
+    function initialize(IERC20 _underlyingToken, IPauserRegistry _pauserRegistry, uint256 _rateLimitAmount)
↪  public virtual initializer {
         _initializeStrategyBase(_underlyingToken, _pauserRegistry);
+        rateLimitAmountPerBlock = _rateLimitAmount;
     }

     /// @notice Sets the `underlyingToken` and `pauserRegistry` for the strategy.
@@ -170,6 +175,16 @@ contract StrategyBase is Initializable, Pausable, IStrategy {
      */
     function _beforeDeposit(IERC20 token, uint256 amount) internal virtual {
         require(token == underlyingToken, "StrategyBase.deposit: Can only deposit underlyingToken");
+        uint256 _blockAmount; // used as local cache, for gas optimizations
+        if (block.number == lastUpdateBlockNumber) {
+            _blockAmount = lastUpdateAmount + amount;
+        } else {
+            lastUpdateBlockNumber = block.number;
+            _blockAmount = amount;
+        }
+        // store amount
+        lastUpdateAmount = _blockAmount;
+        require(_blockAmount <= rateLimitAmountPerBlock, "StrategyBase.deposit: Can only deposit rate-limited
↪  amount");
     }
```

```
     /**
diff --git a/src/contracts/strategies/StrategyBaseTVLLimits.sol
↪  b/src/contracts/strategies/StrategyBaseTVLLimits.sol
index a395bf08..73baf57f 100644
--- a/src/contracts/strategies/StrategyBaseTVLLimits.sol
+++ b/src/contracts/strategies/StrategyBaseTVLLimits.sol
@@ -30,10 +30,11 @@ contract StrategyBaseTVLLimits is StrategyBase {
         uint256 _maxPerDeposit,
         uint256 _maxTotalDeposits,
         IERC20 _underlyingToken,
-        IPauserRegistry _pauserRegistry
+        IPauserRegistry _pauserRegistry,
+        uint256 _rateLimitAmount
     ) public virtual initializer {
         _setTVLLimits(_maxPerDeposit, _maxTotalDeposits);
-        _initializeStrategyBase(_underlyingToken, _pauserRegistry);
+        _initializeStrategyBase(_underlyingToken, _pauserRegistry, _rateLimitAmount);
     }

     /**
```

Note: The patch can be applied with `git apply`. An admin function to update the rate limit could also be implemented for future needs.

3.2.2 Paused deregistering is bypassed via `registeroperatorwithchurn`

Submitted by hash

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Deregistering of operators is supposed to be pausable as indicated by the `onlyWhenNotPaused(PAUSED_REGISTER_OPERATOR)` modifier on the `deregister` function:

```
function deregisterOperator(
    bytes calldata quorumNumbers
) external onlyWhenNotPaused(PAUSED_DEREGISTER_OPERATOR) {
```

But users can be deregistered even when `PAUSED_REGISTER_OPERATOR` bit is set via the `registerOperatorWithChurn` function when the necessary conditions (the removing operator's stake, new operator's stake and the total operator count) are met:

```
function registerOperatorWithChurn(
    bytes calldata quorumNumbers,
    string calldata socket,
    IBLSApkRegistry.PubkeyRegistrationParams calldata params,
    OperatorKickParam[] calldata operatorKickParams,
    SignatureWithSaltAndExpiry memory churnApproverSignature,
    SignatureWithSaltAndExpiry memory operatorSignature
) external onlyWhenNotPaused(PAUSED_REGISTER_OPERATOR)   // ...
    // ...
    _deregisterOperator(operatorKickParams[i].operator, quorumNumbers[i:i+1]);
    // ...
}
```

Recommendation: If this behavior is unintended, check for `onlyWhenNotPaused(PAUSED_REGISTER_OPERATOR)` in the internal `_deregister` function instead.