# Security Review Report
# NM-0231 SWELL

NETHERMIND SECURITY

(Jun 4, 2024)

# Contents

# 1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the Swell contracts. **Swell** is a non-custodial staking protocol, allowing holders of Ether or Liquid Staking Tokens (LSTs) to earn yield by staking and restaking.

Users can deposit their Ether or Liquid Staking Tokens into the Swell contracts in exchange for either swETH or the restaking token `rswETH`. These deposited tokens are restaked within the EigenLayer protocol facilitated by the `EigenPodManager` contract for native ETH and the `StrategyManager` for LSTs, generating rewards.

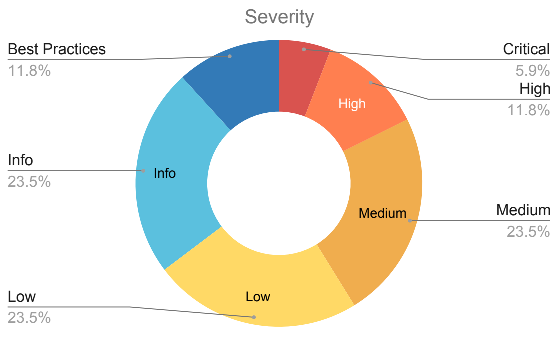
Upon receipt of rewards, node operators and the Swell treasury are compensated according to predetermined percentages. Subsequently, the price of `rswETH` is adjusted to reflect the claimed rewards. This mechanism ensures that `rswETH` holders benefit from an augmented rate, thereby receiving increased funds upon burning their tokens.

This audit focuses on the latest protocol upgrade designed to accommodate the M2 version of EigenLayer contracts. This upgrade entails introducing `StakerProxy` contracts, managed by the `EigenLayerManager` contract. Each `StakerProxy` contract represents a distinct staker and is associated with an EigenPod, enabling the protocol to allocate funds among various stakers and EigenLayer operators efficiently. Additionally, it preserves the V1 EigenPod, which is managed by the `DepositManager`. **Note that the mechanism for redeeming `rswETH`, implemented within the `RswExit` contract, falls outside the scope of this audit. Furthermore, the review of the repricing mechanism assumes the accuracy of the data reported by the Repricing Oracle**.

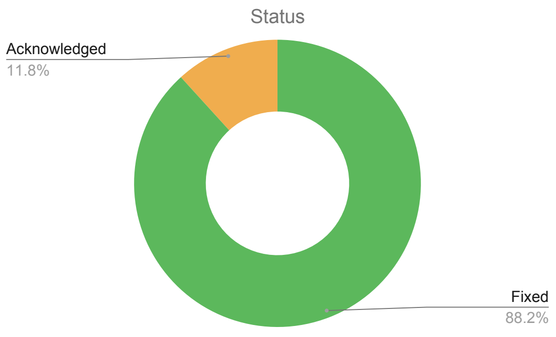
**The audited code comprises** 1492 lines of code in Solidity. The **Swell** team has provided documentation that explains Swell's architecture and the detailed properties of some of the components. Furthermore, the documentation details the economics behind Swell's tokens.

**The audit was performed using**: (a) manual analysis of the codebase, (b) simulation of the smart contracts. **Along this document, we report** 17 points of attention, where one is classified as `Critical`, two are classified as `High`, four are classified as `Medium`, four are classified as `Low`, and six are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

**This document is organized as follows.** Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

**Fig 1: (a) Distribution of issues: Critical** (1), **High** (2), **Medium** (4), **Low** (4), **Undetermined** (0), **Informational** (4), **Best Practices** (2). **(b) Distribution of status: Fixed** (15), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

## Summary of the Audit

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | May 31, 2024 |
| **Final Report** | Jun 4, 2024 |
| **Methods** | Manual Review, Automated analysis |
| **Repository** | v3-contracts-lrt |
| **Commit Hash** | e0d35943929a8bd3d8e6945eec5d20347afc5e84 |
| **Final Commit Hash** | efda60f8bc0510cfad8aee10e5786ab511b473ab |
| **Documentation** | README & Documentation |
| **Documentation Assessment** | High |
| **Test Suite Assessment** | Medium |

## 2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | implementations/RswETH.sol | 298 | 37 | 12.4% | 86 | 421 |
| 2 | implementations/EigenLayerManager.sol | 453 | 20 | 4.4% | 59 | 532 |
| 3 | implementations/DepositManager.sol | 232 | 27 | 11.6% | 58 | 317 |
| 4 | implementations/StakerProxy.sol | 181 | 14 | 7.7% | 33 | 228 |
| 5 | implementations/RateProviders/SfrxETHRateProvider.sol | 29 | 8 | 27.6% | 9 | 46 |
| 6 | implementations/RateProviders/OETHRateProvider.sol | 20 | 6 | 30.0% | 8 | 34 |
| 7 | implementations/RateProviders/OsETHRateProvider.sol | 29 | 7 | 24.1% | 9 | 45 |
| 8 | implementations/RateProviders/CbETHRateProvider.sol | 29 | 7 | 24.1% | 9 | 45 |
| 9 | implementations/RateProviders/StETHRateProvider.sol | 22 | 6 | 27.3% | 8 | 36 |
| 10 | implementations/RateProviders/METHRateProvider.sol | 29 | 7 | 24.1% | 9 | 45 |
| 11 | implementations/RateProviders/ETHxRateProvider.sol | 30 | 7 | 23.3% | 9 | 46 |
| 12 | implementations/RateProviders/RETHRateProvider.sol | 29 | 7 | 24.1% | 9 | 45 |
| 13 | implementations/RateProviders/WbETHRateProvider.sol | 25 | 7 | 28.0% | 9 | 41 |
| 14 | implementations/RateProviders/WstETHRateProvider.sol | 29 | 7 | 24.1% | 9 | 45 |
| 15 | implementations/RateProviders/AnkrETHRateProvider.sol | 30 | 10 | 33.3% | 10 | 50 |
| 16 | implementations/RateProviders/SwETHRateProvider.sol | 27 | 7 | 25.9% | 9 | 43 |
| | **Total** | **1492** | **184** | **12.3%** | **343** | **2019** |

## 3 Summary of Issues

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Missing `initializer` modifier leads to reinitialization of `StakerProxy` . | Critical | Fixed |
| 2 | Incorrect assignment of `upgradableBeacon` results in breaking the beacon proxy pattern | High | Fixed |
| 3 | LST Tokens withdrawn from `EigenStrategy` are locked in the `StakerProxy` contract | High | Fixed |
| 4 | BOT can invoke functions when paused | Medium | Fixed |
| 5 | Incorrect loop termination in `_undelegateStakerFromOperator(...)` function | Medium | Fixed |
| 6 | User can receive fewer `rswETH` tokens due to the rate change while depositing LST tokens | Medium | Fixed |
| 7 | `depositLST(...)` should be pausable | Medium | Fixed |
| 8 | Possible locked Ether within `EigenLayerManager` contract | Low | Fixed |
| 9 | Possible unminted rewards leading to discrepancies in `rswETH` rate | Low | Fixed |
| 10 | Unbounded loop in the `reprice(...)` function | Low | Acknowledged |
| 11 | `depositLST(...)` ignores `RswETH` contract whitelist | Low | Fixed |
| 12 | Inconsistent retrieving of `stakerProxy` address | Info | Fixed |
| 13 | The `operatorToStakers` mapping may contain outdated data | Info | Acknowledged |
| 14 | Unnecessary assignment during element removal from the `delegatedStakers` array | Info | Fixed |
| 15 | Unnecessary writing into storage in the `operatorToStakers` mapping | Info | Fixed |
| 16 | Inconsistency of `SafeERC20` functions usage | Best Practices | Fixed |
| 17 | The `owner` in `StakerProxy` could be misleading | Best Practices | Fixed |

# 4 System Overview

**Swell** is a non-custodial staking protocol, allowing users to engage in staking and restocking activities by depositing ETH or LSTs, thereby earning yield without having their funds locked up or running their own validators.

The **Swell** protocol compromises several contracts. We list below the main contracts in scope for the present code review:

– `DepositManager`: This contract serves as the custodian of all funds deposited by users. It facilitates the transfer of ETH and LSTs for staking on `EigenLayer`. Additionally, it manages and owns the `EigenPod` created in the previous version of Swell contracts. Consequently, it was initially responsible for setting up new validators for the ETH staking process. However, this functionality has been transferred to the `EigenLayerManager` contract for the new EigenPods.

– `EigenLayerManager`: This contract handles all interactions with `EigenLayer` contracts in the updated version of the protocol. It manages the `StakerProxy` contracts and initiates all staking and withdrawal processes on the `EigenLayer` contracts.

– `StakerProxy`: This contract represents a staker on `EigenLayer`, linked to a single `EigenPod` created during its initialization. As an owner of `EigenPod`, it provides functions to stake, initiate LST deposits to EigenLayer's StrategyManager, and manage withdrawal requests.

– `RswETH`: Serving as Swell's restaking token. Users are enabled to deposit ETH or LSTs via the `DepositManager` in exchange for `rswETH` based on the current price. The `rswETH-ETH` rate is updated via the `reprice(...)` function.

– `Rate Providers contracts`: Each LST has its own `RateProvider` contract, providing the current LST-ETH rate. This rate is used in the deposit process within the `DepositManager` contract.

The following sections provide an overview of the system components, highlighting the distinct roles within the system, the repricing mechanism for the `rswETH` token, as well as the user deposit and withdrawal processes. Furthermore, it explains the stakers management within the system and the staking and withdrawal processes on `EigenLayer`.

## 4.1 Access control roles

`Swell` contracts use a single `AccessControlManager` contract, which defines the different roles within the system:

– `PLATFORM_ADMIN`: As the main admin of `Swell` contracts, this actor controls important protocol parameters. They manage operators within the `NodeOperatorRegistry`, control the whitelist for deposits, and have the capability to pause functionalities within other contracts.

– `BOT`: This actor serves to initiate the staking process on EigenLayer. Additionally, they can transfer Ether from the `StakerProxy` contract back to the `DepositManager`.

– `EIGENLAYER_DELEGATOR`: Manages delegations of EigenLayer stakers to operators. It is allowed to call functions that delegate a staker to an operator or undelegated it.

– `EIGENLAYER_WITHDRAWALS`: Responsible for withdrawal processes on EigenLayer, allowed to call functions to initiate or complete withdrawal requests on EigenLayer.

– `REPRICER`: This actor is responsible for updating the `rswETH` token price via the `reprice(...)` function.

## 4.2 rswETH repricing mechanism

The repricing mechanism for the `rswETH` token, managed exclusively by the `REPRICER` role through the `reprice(...)` function, is important for maintaining an accurate `rswETH` to ETH exchange rate. This function, designed for frequent invocation, adjusts the exchange rate based on the reported `totalSupply` of tokens, total ETH reserves, and the accumulated rewards at a specific snapshot.

The function signature is as follows:

```
function reprice(uint256 _preRewardETHReserves, uint256 _newETHRewards, uint256 _rswETHTotalSupply)
```

Where:

– _preRewardETHReserves: Represents the total ETH reserves at the snapshot time, excluding the accumulated rewards.

– _newETHRewards: Amount of new rewards in ETH. A percentage of these rewards will be allocated among the Swell Treasury and various node operators through minting equivalent `rswETH` amounts.

– _rswETHTotalSupply: Total supply of `rswETH` token at the snapshot time.

The exchange rate is updated according to the following formula; it is computed as the ratio between the total ETH reserves and the corresponding total supply of `rswETH`, with the distributed rewards `rewardsInRswETH` accounted as minted.

$$\text{updatedRswETHToETHRate} = \frac{\text{FixedtotalReserves}}{\text{rswETHTotalSupply} + \text{rewardsInRswETH}}$$

Where:

- `FixedtotalReserves` represents the total ETH reserves, obtained as the sum of the pre-rewards ETH reserves and the new ETH rewards.

$$\text{FixedtotalReserves} = \text{\_preRewardETHReserves} + \text{\_newETHRewards}$$

- `rswETHTotalSupply` is the `rswETH` total supply, provided as an input to the function.
- `rewardsInRswETH` is the amount of rewards that are distributed among node operations and Swell Treasury, converted into an `rswETH` amount.

Moreover, the repricing process is controlled using multiple thresholds:

- **Repricing Period**: The `minimumRepriceTime` parameter sets the minimum time interval between two consecutive calls to update the rate.
- **Rate Deviation**: If the newly computed rate deviates from the previous one by an amount exceeding the defined threshold `maximumRepriceDiff`, the repricing process reverts, avoiding drastic fluctuations in the exchange rate.
- **Total Supply Changes**: The `maximumRswETHDiff` parameter establishes the maximum allowable difference between the reported total token supply and the actual current supply, serving as a guard against manipulation.

## 4.3 Users deposits and withdrawals

Users have the option to mint `rswETH` tokens by depositing either native Ether or Liquid Staking Tokens (LSTs).

- The `depositLST(...)` function within the `DepositManager` contract facilitates the minting of `rswETH` tokens by depositing LSTs. This process involves first converting the deposited LST amount into an equivalent ETH amount, utilizing the corresponding `RateProvider` contract. Subsequently, the obtained ETH amount is further converted into `rswETH` tokens based on the current exchange rate.
- Alternatively, users can interact with the `RswETH` contract through its `deposit(...)` and `depositWithReferral(...)` functions, enabling the deposit of ETH to mint corresponding `rswETH` tokens, based on the current exchange rate.

All funds deposited are directed to the `DepositManager` contract, where they are utilized for staking on EigenLayer.

A user can redeem his `rswETH` token for ETH in the `RswEXIT` contract. The withdrawals are split into three stages. Firstly, a user initiates a withdrawal request, the current rswETH-ETH rate is saved, and the user receives an NFT, which works as a receipt to receive his ETH once the request is processed. Secondly, the `PROCESS_WITHDRAWALS` actor processes the withdrawal request, and finally, the user calls `finalizeWithdrawal(...)` to burn their NFT and obtain the ETH amount.

## 4.4 Stakers on EigenLayer

The `EigenLayerManager` contract manages the deployment of `StakerProxy` contracts, each identified by a unique ID. These contracts represent individual stakers on EigenLayer, each associated with a single `EigenPod`, created upon staker deployment.

The `StakerProxy` contracts, implemented under a beacon proxy pattern, share a unified implementation and are managed by the same admin signer.

The `EIGENLAYER_DELEGATOR` actor can delegate a staker to a specific operator on EigenLayer via the `delegateToWithSignature(...)` function, which requires parameters including the staker's ID, the operator's address, and their respective signatures.

```
function delegateToWithSignature(uint256 _stakerId, address _operator,
IDelegationManager.SignatureWithExpiry calldata _stakerSignatureAndExpiry,
IDelegationManager.SignatureWithExpiry calldata _approverSignatureAndExpiry, bytes32 _approverSalt)
```

The contract also offers functions to revoke delegation, allowing stakers to undelegate themselves from operators. This can be achieved via the `undelegateStakerFromOperator(...)` function:

```
function undelegateStakerFromOperator(uint256 _stakerId)
```

Or in a batched way via the `batchUndelegateStakerFromOperator(...)` function:

```
function batchUndelegateStakerFromOperator(uint256[] calldata _stakerIdArray)
```

## 4.5 Deposits on EigenLayer

The EigenLayerManager manages all the interactions with EigenLayer contracts, encompassing staking, depositing into strategies, and withdrawals.

### 4.5.1 Stake on EigenLayer Pods

The stakeOnEigenLayer(...) function enables the BOT to stake on EigenLayer. It requires providing a list of stakers identified by _stakersIds, alongside public keys of Swell registered operators denoted by _pubKeys, and the deposit data root _depositDataRoot. This process involves depositing 32ETH for each staker via the stake(...) function of EigenPodManager. Funds are transferred from the DepositManager, excluding ETH reserved for exits.

```
function stakeOnEigenLayer(uint256[] calldata _stakerIds, bytes[] calldata _pubKeys, bytes32 _depositDataRoot)
```

### 4.5.2 Deposits into EigenLayer Strategy

Similarly, the depositIntoEigenLayerStrategy(...) function allows the BOT actor to deposit liquid staking tokens into an EigenLayer Strategy. It requires providing the staker ID, the amount of LST to deposit, and the token address. This action involves transferring the specified token amount from the DepositManager contract to the StakerProxy identified by the ID. Subsequently, the StakerProxy executes the deposit into the EigenLayer strategy through the depositIntoStrategy(...) function of EigenLayer's StrategyManager contract.

```
function depositIntoEigenLayerStrategy(uint256 _stakerId, uint256 _amount, address _token)
```

## 4.6 Withdrawals on EigenLayer

Withdrawals from EigenLayer are managed by the EIGENLAYER_WITHDRAWALS role. For that, EigenLayerManager contract offers two main functions:

- queueWithdrawals(...): This function initiates a withdrawal on EigenLayer, invoking the queueWithdrawals(...) function on the DelegationManager contract.
- completeQueuedWithdrawal(...): Used to finalize withdrawals on EigenLayer for a specific StakerProxy.

These functions necessitate the staker proxy ID on which the withdrawal is executed and the withdrawal parameters required by EigenLayer's DelegationManager contract.

Furthermore, the contract includes two other withdrawal functions:

- claimDelayedWithdrawals(): This function enables the claiming of withdrawals previously sent as delayed withdrawals on the DelayedWithdrawalRouter contract. Claimed funds are promptly transferred to the designated recipient.
- verifyAndProcessWithdrawals(...): This function is used for withdrawing beacon chain rewards as partial withdrawals. It requires the staker ID as an input to identify the EigenPod on which the function is invoked.

Note that the DepositManager contract features the eigenPodWithdrawBeforeRestaking(...) function. This function invokes the withdrawBeforeRestaking() function on the inherited EigenPod from the previous Swell version, facilitating the withdrawal of the pod's balance before restaking is activated.

# 5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

**Likelihood** measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

**Impact** is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

| | | **Severity Risk** | | |
|---|---|---|---|---|
| **Impact** | **High** | Medium | High | Critical |
| | **Medium** | Low | Medium | High |
| | **Low** | Info/Best Practices | Low | Medium |
| | **Undetermined** | Undetermined | Undetermined | Undetermined |
| | | **Low** | **Medium** | **High** |
| | | **Likelihood** | | |

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

# 6 Issues

## 6.1 [Critical] Missing `initializer` modifier leads to reinitialization of `StakerProxy`.

**File(s)**: `StakerProxy.sol`

**Description**: The `initialize()` function in the `StakerProxy` contract serves to initialize the core variables of the contract.

```
// @audit Missing `initializer` modifier
function initialize(
    IAccessControlManager _accessControlManager,
    address _delegationManager,
    address _eigenPodManager,
    address _depositManager,
    address _eigenLayerManager,
    address _owner
) public checkZeroAddress(address(_accessControlManager))
    checkZeroAddress(_delegationManager)
    checkZeroAddress(_depositManager)
    checkZeroAddress(_eigenLayerManager)
    checkZeroAddress(_owner) {
    AccessControlManager = _accessControlManager;
    DelegationManager = IDelegationManager(_delegationManager);
    depositManager = _depositManager;
    eigenLayerManager = _eigenLayerManager;

    owner = _owner;
    EigenPodManager = IEigenPodManager(_eigenPodManager);
    eigenPod = EigenPodManager.createPod();
}
```

However, the function is missing the `initializer` modifier to prevent double initialization of the contract. A malicious actor can call the `initialize(...)` function, manipulating the contract variables by pointing them to malicious implementations. This can result in stealing ETH and tokens sent to the contract.

**Recommendation(s)**: Consider adding the `initializer` modifier to prevent the function from being called multiple times. Additionally, call `_disableInitializers()` within the contract constructor.

**Status**: Fixed

**Update from the client**: Resolved in commit b16b83fbd09ccf797d42c0fb8aaefa0362c388df

## 6.2 [High] Incorrect assignment of `upgradableBeacon` results in breaking the beacon proxy pattern

**File(s)**: `EigenLayerManager.sol`

**Description**: The `upgradableBeacon` variable within the `EigenLayerManager` contract stores the address of the beacon proxy contract. This contract holds the implementation address of different Staker proxies and provides the `updateTo(...)` function to update its address.

In the current flow, each `StakerProxy` contract fetches the implementation address from the `upgradableBeacon` contract before each function invocation by calling the `implementation()` function. Subsequently, the function call is delegated to the implementation.

Within the `EigenLayerManager` contract, the `upgradeStakerProxy(...)` function allows the admin to upgrade the implementation within the beacon contract. However, after calling the `upgradeTo(...)` function, there's an additional step where the `upgradableBeacon` variable is replaced by the new implementation address.

```
function upgradeStakerProxy(address _newImplementation) {
   external
   checkRole(SwellLib.PLATFORM_ADMIN)
   checkZeroAddress(_newImplementation) {
   //@audit Upgrade of the `StakerProxy` implementation on beacon proxy
   UpgradeableBeacon(upgradableBeacon).upgradeTo(address(_newImplementation));

   //@audit Beacon proxy is replaced by its implementation address
   upgradableBeacon = _newImplementation;
   emit StakerProxyUpgraded(_newImplementation);
}
```

This results in the `upgradableBeacon` variable pointing to the staker implementation instead of the beacon proxy. Consequently, the newly deployed `StakerProxy` contracts become inoperational as the call to `implementation()` on the implementation contract will always revert. This error cannot be fixed without a protocol upgrade, as `upgradeStakerProxy(...)` will revert when invoking the `upgradeTo()` function, and the `registerStakerProxyImplementation(...)` function doesn't allow resetting the `upgradableBeacon` variable.

**Recommendation(s)**: Remove the assignment of the `upgradableBeacon` address.

**Status**: Fixed

**Update from the client**: Resolved in commit bf69e5c715538c734fcf12cf15fa0280e8438f68

## 6.3 [High] LST Tokens withdrawn from `EigenStrategy` are locked in the `StakerProxy` contract

**File(s)**: `StakerProxy.sol`

**Description**: The withdrawal process within `EigenLayer` involves two distinct stages. Initially, the owner of deposited assets initiates a withdrawal request. Subsequently, after a predefined period, the withdrawer can claim their assets. Upon completion of the withdrawal process via `completeQueuedWithdrawal(...)`, the assets are transferred to the withdrawer's address, which, in the context of this protocol, corresponds to the address of the `StakerProxy` contract.

However, an issue arises when the `StakerProxy` completes a withdrawal from LST strategies. While LSTs are sent to the `StakerProxy`, there is no functionality to transfer ERC20 tokens from the contract.

**Recommendation(s)**: Implement functionality that allows the admin to transfer ERC20 tokens sent to `StakerProxy` contract.

**Status**: Fixed

**Update from the client**: Resolved in commit bab7bf81611b9f65e96f5843e18e97e9f1dfa8f1

## 6.4 [Medium] BOT can invoke functions when paused

**File(s)**: `EigenLayerManager.sol`

**Description**: The `AccessControlManager` implements a `botMethodsPaused()` function to ensure bot methods are disallowed when paused. However, the `depositIntoEigenLayerStrategy(...)` function within `EigenLayerManager`, designed as a BOT function to perform deposits into the `EigenLayer` strategies, lacks the necessary check of the paused status.

```
function depositIntoEigenLayerStrategy(
    uint256 _stakerId,
    uint256 _amount,
    address _token
) external checkRole(SwellLib.BOT) checkZeroAddress(_token) nonReentrant {
   // @audit function can be called when Bot methods are paused
   // ...
}
```

Similarly, the `sendFundsToDepositManager()` function lacks the paused check.

This allows these functions to be invoked even when bot methods are paused, contrary to the intended behavior.

**Recommendation(s)**: Consider implementing a check for `botMethodsPaused()` within the mentioned functions, disallowing their invocation if BOT methods are paused.

**Status**: Fixed

**Update from the client**: Resolved in commit 8b7b9520d0407148ecbec7deb41ecc6bf7476951. Second revision resolved in commit efda60f8bc0510cfad8aee10e5786ab511b473ab

## 6.5 [Medium] Incorrect loop termination in `_undelegateStakerFromOperator(...)` function

**File(s)**: `EigenLayerManager.sol`

**Description**: The `_undelegateStakerFromOperator(...)` function is designed to undelegate a staker from its associated operator. It achieves this by iterating through the list of stakers for the specified operator. When the targeted staker is found, it is removed from the array.

However, there's an issue with the loop implementation. The loop's termination condition relies on a counter variable `i`, which is only incremented within the `if` block when the provided staker is found. Consequently, the function enters an infinite loop when it encounters an element different from the provided staker, leading to an `out-of-gas` error and preventing stakers from undelegating themselves from operators.

```
function _undelegateStakerFromOperator(uint256 _stakerId) internal {
  // ...
  uint256[] storage delegatedStakers = operatorToStakers[operator];
  uint256 temp;
  for (uint256 i; i < delegatedStakers.length; ) {
    if (delegatedStakers[i] == _stakerId) {
      // ...
      //@audit counter incremented only when the staker is found
      unchecked {
        ++i;
      }
    }
  }
  StakerProxy(payable(staker)).undelegateFromOperator();
}
```

**Recommendation(s)**: Ensure that the loop counter `i` is incremented only when the staker is not found.

**Status**: Fixed

**Update from the client**: Resolved in commit 36a00f4313fe73215553ae2f27fa0d0bfe68b102

## 6.6 [Medium] User can receive fewer `rswETH` tokens due to the rate change while depositing LST tokens

**File(s)**: `DepositManager.sol`

**Description**: The `depositLST(...)` function allows the deposit of the various LST tokens in exchange for the `rswETH` token. The amount of `rswETH` is calculated based on the current `LST-ETH` and `ETH-rswETH` rates.

```
function depositLST(
    address _token,
    uint256 _amount
  ) external checkZeroAddress(_token) {
  // ...
  //@audit `ETHAmount` computed using the current LST rate
  uint256 rate = ILstRateProvider(exchangeRateProviders[_token]).getRate();
  uint256 ETHAmount = (_amount * rate) / 1e18;
  // ...
  //@audit Mint of `rswETH` depends on current `rswETH` rate
  rswETH.depositViaDepositManager(ETHAmount, msg.sender);
  // ...
}
```

The current implementation lacks protection against slippage resulting from LST token price change, potentially causing users to receive fewer `rswETH` tokens than expected.

**Recommendation(s)**: Consider introducing a slippage protection mechanism within the deposit flow. This could involve adding a parameter `minRswETH` to the `depositLST(...)` function, representing the minimum expected amount of `rswETH` to be minted for the user.

**Status**: Fixed

**Update from the client**: Resolved in commit cff2dff111de1ac6a3ed13fb77622ff15b036a0f and 16ff9a39e35b48876cd814cd29eb04a860557804

## 6.7 [Medium] `depositLST(...)` should be pausable

**File(s)**: `DepositManager.sol`

**Description**: According to the provided `README.md`, user functionality for deposits and withdrawals is a core method that should be pausable. However, the function `depositLST(...)` is not pausable as it lacks a call to the `AccessControlManager.coreMethodsPaused()`.

**Recommendation(s)**: Implement the call to `AccessControlManager.coreMethodsPaused()` within the `depositViaDepositManager(...)` function of `RswETH` contract to check if the deposit functionality is paused.

**Status**: Fixed

**Update from the client**: Resolved in commit 22a9155432e4481d5bb997f5992f0e99b68c563a

## 6.8 [Low] Possible locked Ether within `EigenLayerManager` contract

**File(s)**: `EigenLayerManager.sol`

**Description**: The `receive()` function allows Ether to be received from any account. Within the `EigenLayerManager`, this function is specifically implemented to manage the transfer of Ether from the `DepositManager` for the Ethereum staking process. This transfer is initiated during the execution of the `DepositManager().transferETHForEigenLayerDeposits(...)` method, where a specified amount of funds is transferred and subsequently staked directly through the `EigenPod`.

However, Ether sent directly to the contract, whether as a donation or due to a user error, cannot be withdrawn as the `EigenLayerManager` lacks withdrawal functionality for Ether. Consequently, the Ether becomes locked within the contract.

_Note:_ _This problem was also found in the `RswExit` contract, which is out of scope for this audit._

**Recommendation(s)**: Consider implementing a withdrawal functionality for the received `ETH`. Alternatively, restrict calls to the `receive()` function, allowing transfers only from the `DepositManager` contract.

**Status**: Fixed

**Update from the client**: Addressed in commit afa39217dbb51a901908dd9bab3460437b9f134e

## 6.9 [Low] Possible unminted rewards leading to discrepancies in `rswETH` rate

**File(s)**: `RswETH.sol`

**Description**: The `reprice(...)` function is designed to update the `rswETH` exchange rate based on the provided inputs. This involves computing rewards in `rswETH` ( `rewardsInRswETH`), distributing them among node operators ( `nodeOperatorRewards`) and Swell treasury ( `swellTreasuryRewards`), based on the predefined percentages.

When computing the new rate `updatedRswETHToETHRateFixed`, the function assumes that all rewards will be fully minted within the function as `rewardsInRswETH` is included in the denominator. However, this assumption may not hold due to rounding errors when distributing rewards to node operators. Specifically, the sum of the minted shares ( `operatorsRewardShare`) for each operator might be slightly less than the total rewards allocated to node operators ( `nodeOperatorRewards`).

At the end of the function, the remaining reward tokens are minted to the treasury by subtracting `nodeOperatorRewards` from the total rewards `rewardsInRswETH`, assuming that `nodeOperatorRewards` was fully minted. This results in some rewards being accounted for within the new rate but remaining unused within the contract, leading to a slightly lower new rate compared to the actual one.

```solidity
function reprice(...) external override checkRole(SwellLib.REPRICER) {
  // ...
  //@audit total rewards to be distributed
  UD60x18 rewardsInRswETH = wrap(_rswETHTotalSupply).mul(rewardsInETH).div(
    wrap(totalReserves - rewardsInETH.unwrap())
  );

  // @audit new rate accounts for `rewardsInRswETH` as minted shares
  uint256 updatedRswETHToETHRateFixed = wrap(totalReserves)
    .div(wrap(_rswETHTotalSupply + rewardsInRswETH.unwrap()))
    .unwrap();

  // ...

  uint256 nodeOperatorRewards;
  uint256 swellTreasuryRewards;

  if (rewardsInRswETH.unwrap() != 0) {
    // ...
    if (totalActiveValidators == 0) {
      nodeOperatorRewards = 0;
    } else if (nodeOperatorRewards != 0) {
      uint128 totalOperators = nodeOperatorRegistry.numOperators();
      UD60x18 rewardsPerValidator = wrap(nodeOperatorRewards).div(
        wrap(totalActiveValidators)
      );

      for (uint128 i = 1; i <= totalOperators; ) {
        // ...
        if (operatorActiveValidators != 0) {
        // @audit Possible precision loss in `operatorsRewardShare`
          uint256 operatorsRewardShare = rewardsPerValidator
            .mul(wrap(operatorActiveValidators))
            .unwrap();

          _mint(rewardAddress, operatorsRewardShare);
        }
        // ...
      }
    }

    //@audit The sum of `operatorsRewardShare` minted shares can be less than `nodeOperatorRewards`
    swellTreasuryRewards = rewardsInRswETH.unwrap() - nodeOperatorRewards;
    // ...
  }
  // ...
}
```

**Recommendation(s)**: To address this issue, compute the actual minted amount for validators by summing up `operatorsRewardShare` and use it when computing the remainder for the Swell Treasury. This ensures that the total `rswETH` rewards `rewardsInRswETH` are fully minted and accurately reflected in the new rate.

**Status**: Fixed

**Update from the client**: Resolved in commit 40a6d7e18deee2abd653729b59595224f619fcec

## 6.10 [Low] Unbounded loop in the `reprice(...)` function

**File(s)**: `RswETH.sol`

**Description**: In the `RswETH` contract, the `reprice(...)` function iterates through the list of operators to distribute reported rewards among them. However, as the number of operators may increase over time, the loop within this function could consume excessive gas, leading to the function consistently reverting with an `out-of-gas` error.

**Recommendation(s)**: Consider revisiting the logic within the `reprice(...)` function to handle a scalable number of operators.

**Status**: Acknowledged

**Update from the client**: We acknowledge this issue. As the number of operators will be very small in the short to medium term, this issue will not likely be encountered. As we onboard new operators, we will continue to monitor the gas consumption effects and ensure there is sufficient headroom to avoid any out-of-gas error. In the long term, this issue will be addressed through a new protocol upgrade.

## 6.11 [Low] `depositLST(...)` ignores `RswETH` contract whitelist

**File(s)**: `RswETH.sol`

**Description**: A user can obtain `rswETH` by depositing ETH directly to the `RswETH` contract or by depositing LST tokens through the `DepositManager.depositLST(...)` function. Deposits are restricted to a list of whitelisted users if the `whitelistEnabled` storage variable is set to `true`.

```
function _deposit(address referral) internal checkWhitelist(msg.sender) {
    if (AccessControlManager.coreMethodsPaused()) {
      revert SwellLib.CoreMethodsPaused();
    }
    // ...
}
```

The `_deposit(...)` function is invoked during ETH deposits in the `RswETH` contract. This function incorporates the `checkWhitelist` modifier to verify if the whitelist is activated. If enabled, the `msg.sender` must be whitelisted.

However, the `depositLST(...)` function lacks implementation of the whitelisting check, allowing users to deposit LST and obtain `rswETH` even if they are not whitelisted.

**Recommendation(s)**: Consider adding the `checkWhitelist(msg.sender)` modifier to the `depositLST(...)` function.

**Status**: Fixed

**Update from the client**: Resolved in commit e507d04f1f45faba9909a638c494d1b7adcac74d

## 6.12 [Info] Inconsistent retrieving of `stakerProxy` address

**File(s)**: `EigenLayerManager.sol`

**Description**: The `StakerProxy` address within `EigenLayerManager` is always retrieved through the internal function `_isValidStaker(uint256 id)`. However, in the function `batchWithdrawERC20(...)`, the address is directly retrieved from the `stakerProxyAddresses` mapping.

```
function batchWithdrawERC20(
  uint256 _stakeId,
  IERC20[] memory _tokens,
  uint256[] memory _amounts,
  address _recipient
) external checkRole(SwellLib.PLATFORM_ADMIN) {
  // ...
  // @audit `_isValidStaker(...)` should be used
  address staker = stakerProxyAddresses[_stakeId];
  // ..
}
```

The `_isValidStaker(...)` function ensures the existence of an address for a given `_stakeId` and provides a clear error message if not found.

**Recommendation(s)**: Consider using `_isValidStaker(...)` function to retrieve the `stakerProxy` address in `batchWithdrawERC20(...)` function.

**Status**: Fixed

**Update from the client**: Resolved in commit 2cbaa8c87ec3dfca7577b664af434981c44c2675

### 6.13 [Info] The `operatorToStakers` mapping may contain outdated data

**File(s)**: `EigenLayerManager.sol`

**Description**: The `operatorToStakers` mapping maintains a record of stakers delegated to each operator. It is updated during delegation through the `delegateToWithSignature(...)` or `batchDelegateToWithSignature(...)` functions, as well as during undelegation via `undelegateStakerFromOperator(...)` or `batchUndelegateStakerFromOperator(...)` functions.

However, the mapping may retain outdated information due to the following reasons:

- Delegation with a signature can be executed by anyone, potentially leading to front-running calls to `delegateToWithSignature(...)` and `batchDelegateToWithSignature(...)` functions and performing it directly on `EigenLayer` contracts. Consequently, stakers may be delegated to operators without the mapping being updated and without interaction with the `EigenLayerManager` contract;
- Similarly, undelegation on `EigenLayer` can be initiated by the staker, the operator, or an address authorized by the operator. In the latter two cases, the staker may be undelegated from the operator without involving the `EigenLayerManager` contract, resulting in an un-updated mapping array;

**Recommendation(s)**: The current impact is limited due to the absence of mapping usage within the code in scope. However, it's advisable to revisit the implementation and avoid relying on the mapping data in sensitive operations.

**Status**: Acknowledged

**Update from the client**: The issue is acknowledged and internal processes will be developed to ensure outdated data will not be used for sensitive operations.

### 6.14 [Info] Unnecessary assignment during element removal from the `delegatedStakers` array

**File(s)**: `EigenLayerManager.sol`

**Description**: The `_undelegateStakerFromOperator(...)` function is responsible for undelegating a staker from its associated operator. To perform the deletion, it iterates through the stakers array to find the matching staker ID and replace it with the last element, which will be removed immediately after. However, an unnecessary step is performed during this process where the function swaps the provided staker ID with the last element in the array before removing it. This additional step incurs extra gas overhead due to unnecessary storage write for each loop iteration.

```solidity
function _undelegateStakerFromOperator(uint256 _stakerId) internal {
   // ...
    uint256[] storage delegatedStakers = operatorToStakers[operator];
    uint256 temp;
    for (uint256 i; i < delegatedStakers.length; ) {
      if (delegatedStakers[i] == _stakerId) {
        temp = delegatedStakers[i];
        delegatedStakers[i] = delegatedStakers[delegatedStakers.length - 1];
        //@audit Unnecessary assignment as the last element will be removed
        delegatedStakers[delegatedStakers.length - 1] = temp;
        delegatedStakers.pop();
        unchecked {
          ++i;
        }
      }
    }
    StakerProxy(payable(staker)).undelegateFromOperator();
}
```

**Recommendation(s)**: Consider removing the unnecessary step of moving the `stakerId` to the last element before its removal.

**Status**: Fixed

**Update from the client**: Resolved in commit 5d90bd4308b30ca6ea076aab5f8ec075dee441f2

## 6.15 [Info] Unnecessary writing into storage in the `operatorToStakers` mapping

**File(s)**: `EigenLayerManager.sol`

**Description**: Within the `delegateToWithSignature(...)` and `batchDelegateToWithSignature(...)` functions, a new staker ID is added to the `operatorToStakers` mapping. Initially, the array of stakers associated with a particular operator is copied into storage ( `delegatedStakers`). Subsequently, the new ID is pushed into this array. However, an unnecessary step follows where the array is assigned back into the mapping. This last assignment is redundant because updating the array in storage automatically updates the corresponding mapping.

```
function delegateToWithSignature(...) external checkRole(SwellLib.EIGENLAYER_DELEGATOR) {
    address _staker = _isValidStaker(_stakerId);
    uint256[] storage delegatedStakers = operatorToStakers[_operator];
    // ...
    delegatedStakers.push(_stakerId);
    //@audit Unnecessary assignment as `delegatedStakers` is already in storage
    operatorToStakers[_operator] = delegatedStakers;
}
```

**Recommendation(s)**: Consider removing the unnecessary assignment or using the `memory` keyword instead of `storage` for `delegatedStakers`.

**Status**: Fixed

**Update from the client**: Resolved in commit a976f756db0c7edbf758221b0d56ba31ae052d18

## 6.16 [Best Practices] Inconsistency of `SafeERC20` functions usage

**File(s)**: `DepositManager.sol`, `StakerProxy.sol`

**Description**: Most of the contract functions effectively use the `SafeERC20` library for ERC20 token transfers and approvals. However, there are two instances where direct ERC20 interactions are performed, lacking the safety checks provided by the `SafeERC20` library.

The first occurrence is within the `depositLST(...)` function:

```
//@audit `safeTransferFrom` should be used
IERC20(_token).transferFrom(msg.sender, address(this), _amount);
```

The second instance is found in the `depositIntoStrategy(...)` function:

```
//@audit `safeApprove` should be used
token.approve(strategyManagerAddress, _amount);
```

Utilizing `SafeERC20` functions is recommended to ensure proper handling of `ERC20` operations, including edge cases and non-standard tokens.

**Recommendation(s)**: Consider utilizing `SafeERC20` functions for all ERC20 transfers and approvals within the contract.

**Status**: Fixed

**Update from the client**: Resolved in commit cb093f06e11082391dd129cd1e05875beec9e0f7

## 6.17 [Best Practices] The `owner` in `StakerProxy` could be misleading

**File(s)**: `StakerProxy.sol`

**Description**: The `StakerProxy` defines an `owner` variable initialized to the `adminSigner` of `EigenLayerManager` during contract deployment. However, this variable remains unused throughout the contract's logic. Furthermore, updates to the `adminSigner` in `EigenLayerManager` are not propagated to the `StakerProxy` contract, potentially causing inconsistencies.

While the current implementation correctly retrieves the latest `adminSigner` during signature validation, the presence of the local `owner` variable could introduce errors during development if used instead of the current `adminSigner`.

**Recommendation(s)**: If the intended behavior is to maintain a single `adminSigner` across all deployed `StakerProxy` contracts, consider eliminating the `owner` variable from `StakerProxy` to avoid potential discrepancies.

**Status**: Fixed

**Update from the client**: Resolved in commit 4ff70e227fca3adbb522265153a83bcc4e5a4f95

# 7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

**Remarks about Swell documentation**

The `Swell` team has provided comprehensive documentation about their protocol based on the provided README and documentation website. Additionally, the Swell team was available to address any questions or concerns from the Nethermind Security team.

# 8 Test Suite Evaluation

## 8.1 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 209 files with 0.8.16
[] Solc 0.8.16 finished in 34.69s
Compiler run successful with warnings:
// Warnings
```

## 8.2 Tests Output

```
> forge test
Ran 2 tests for test/Foundry/RateProviders/OETHRateProvider.t.sol:OETHRateProviderTest
[PASS] testGetRateSuccess() (gas: 15640)
[PASS] testOETHRateProviderInitialize() (gas: 17128)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 995.94ms (201.73µs CPU time)

Ran 2 tests for test/Foundry/RateProviders/StETHRateProvider.t.sol:StETHRateProviderTest
[PASS] testGetRateSuccess() (gas: 12740)
[PASS] testStETHRateProviderInitialize() (gas: 17136)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 996.14ms (113.44µs CPU time)

Ran 1 test for test/Foundry/StakerProxyUpgrades.t.sol:StakerProxyUpgrades
[PASS] test_upgradeStakerProxy() (gas: 1687525)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 997.54ms (1.06ms CPU time)

Ran 1 test for test/Foundry/EigenLayerManager.t.sol:EigenLayerManagerTest
[PASS] testOnlyDepositManagerCanSendETHToEigenLayerManager() (gas: 77213)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 998.29ms (158.38µs CPU time)

Ran 25 tests for test/Foundry/StakerProxy.t.sol:StakerProxyTest
[PASS] testCompleteQueuedWithdrawalFailsWithIncorrectCaller() (gas: 43197)
[PASS] testCompleteQueuedWithdrawalSuccess() (gas: 836694)
[PASS] testDepositIntoStrategySuccess() (gas: 460997)
[PASS] testQueueWithdrawalsFailsOnInvalidCaller() (gas: 29386)
[PASS] testQueueWithdrawalsSuccess() (gas: 400239)
[PASS] testSendFundsToDepositManagerFailsOnIncorrectCaller() (gas: 95380)
[PASS] testSendFundsToDepositManagerSuccess() (gas: 59081)
[PASS] testSendTokenBalanceToDepositManagerFailsOnInvalidCaller() (gas: 97537)
[PASS] testSendTokenBalanceToDepositManagerFailsOnNoTokensToWithdraw() (gas: 43843)
[PASS] testSendTokenBalanceToDepositManagerSuccess() (gas: 88235)
[PASS] testStakeOnEigenLayerFailsWithIncorrectCaller() (gas: 1263878)
[PASS] testStakerProxyInit() (gas: 38910)
[PASS] testStakerProxyRevertIfAlreadyInitialized() (gas: 32184)
[PASS] testStakerProxyStakeOnEigenLayerSuccess() (gas: 1360746)
[PASS] testUndelegateFailsOnInvalidCaller() (gas: 296161)
[PASS] testUndelegateSuccess() (gas: 295931)
[PASS] testVerifyAndProcessWithdrawalsFailsOnIncorrectCaller() (gas: 274914)
[PASS] testVerifyAndProcessWithdrawalsSuccess() (gas: 754136)
[PASS] testVerifyPodWithdrawalCredentialsFailsOnIncorrectCaller() (gas: 169828)
[PASS] testVerifyPodWithdrawalCredentialsSuccess() (gas: 350309)
[PASS] testWithdrawERC20FromPodFailsOnArrayLengthMismatch() (gas: 66717)
[PASS] testWithdrawERC20FromPodFailsOnIvalidCaller() (gas: 66462)
[PASS] testWithdrawERC20FromPodSuccess() (gas: 83925)
[PASS] testWithdrawNonStakedBeaconChainEthFailsWithIncorrectCaller() (gas: 143191)
[PASS] testWithdrawNonStakedBeaconChainEthSuccess() (gas: 172870)
Suite result: ok. 25 passed; 0 failed; 0 skipped; finished in 1.01s (16.52ms CPU time)

Ran 4 tests for test/Foundry/Scenarios.t.sol:ScenarioTests
[PASS] testScenario1() (gas: 4285882)
[PASS] testScenario2() (gas: 6435826)
[PASS] testScenario3() (gas: 5855274)
[PASS] testStakeDelegateStakeUpdatesOperatorShares() (gas: 3560150)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.02s (23.79ms CPU time)
```

```
Ran 2 tests for test/Foundry/RateProviders/SfrxETHRateProvider.t.sol:SfrxETHRateProviderTest
[PASS] testGetRateMainnetForkSuccess() (gas: 785210)
[PASS] testSfrxETHRateProviderInitialize() (gas: 17224)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.03s (4.04s CPU time)

Ran 2 tests for test/Foundry/RateProviders/OsETHRateProvider.t.sol:OsETHRateProviderTest
[PASS] testOsETHGetRateMainnetForkSuccess() (gas: 790239)
[PASS] testOsETHRateProviderInitialize() (gas: 17137)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.03s (4.04s CPU time)

Ran 2 tests for test/Foundry/RateProviders/CbETHRateProvider.t.sol:CbETHRateProviderTest
[PASS] testCbETHDepositInitialize() (gas: 17128)
[PASS] testGetRateMainnetForkSuccess() (gas: 787468)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.03s (4.04s CPU time)

Ran 2 tests for test/Foundry/RateProviders/ETHxRateProvider.t.sol:ETHxrateProviderTest
[PASS] testETHxDepositInitialize() (gas: 17137)
[PASS] testGetETHxRateMainnetForkSuccess() (gas: 821210)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.23s (4.23s CPU time)

Ran 2 tests for test/Foundry/RateProviders/AnkrETHRateProvider.t.sol:AnkrETHRateProviderTest
[PASS] testAnkrETHrateProviderInitialize() (gas: 17138)
[PASS] testGetRateAnkrMainnetForkSuccess() (gas: 801881)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.98s (4.98s CPU time)

Ran 2 tests for test/Foundry/RateProviders/RETHRateProvider.t.sol:RETHRateProviderTest
[PASS] testGetRateMainnetForkSuccess() (gas: 798476)
[PASS] testRETHRateProviderInitialize() (gas: 17183)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 6.27s (5.27s CPU time)

Ran 67 tests for test/Foundry/DepositManager.t.sol:DepositManagerTest
[PASS] testBatchDelegateToWithSignatureFailsWithArrayLengthMismatch() (gas: 1431722)
[PASS] testBatchDelegateToWithSignatureFailsWithInvalidCaller() (gas: 1488996)
[PASS] testBatchDelegateToWithSignatureSuccess() (gas: 2767166)
[PASS] testBatchUndelegateStakerFromOperatorFailsOnInvalidCaller() (gas: 93566)
[PASS] testBatchUndelegateStakerFromOperatorFailsOnInvalidStakerId() (gas: 36552)
[PASS] testBatchUndelegateStakerFromOperatorSuccess() (gas: 2748087)
[PASS] testBatchWithdrawERC20sFailsWithArrayMismatch() (gas: 73846)
[PASS] testBatchWithdrawERC20sFailsWithIncorrectCaller() (gas: 137212)
[PASS] testBatchWithdrawERC20sFailsWithInvalidRecipient() (gas: 73416)
[PASS] testBatchWithdrawERC20sSuccess() (gas: 712583)
[PASS] testCompleteQueuedWithdrawal() (gas: 1850168)
[PASS] testCompleteQueuedWithdrawalFailsOnInvalidId() (gas: 858386)
[PASS] testCreateStakerAndPodFailsAsExpected() (gas: 118982)
[PASS] testCreateStakerAndPodSuccess() (gas: 2455242)
[PASS] testDelegateToWithSignatureChangeAdminSignerSuccess() (gas: 1836353)
[PASS] testDelegateToWithSignatureFailsWithAddressZero() (gas: 65597)
[PASS] testDelegateToWithSignatureFailsWithIncorrectCaller() (gas: 153094)
[PASS] testDelegateToWithSignatureFailsWithInvalidStaker() (gas: 97253)
[PASS] testDelegateToWithSignatureSuccess() (gas: 1221775)
[PASS] testDepositIntoEigenLayerStrategyFailsAsExpected() (gas: 462124)
[PASS] testDepositIntoEigenLayerStrategyPaused() (gas: 57383)
[PASS] testDepositIntoEigenLayerStrategySuccess() (gas: 2294873)
[PASS] testDepositLSTEmitsEvent() (gas: 365769)
[PASS] testDepositLSTFailsCorrectly() (gas: 303359)
[PASS] testDepositLSTFailsDueToWhitelist() (gas: 224503)
[PASS] testDepositLSTFailsWhenPaused() (gas: 341437)
[PASS] testDepositLSTRevertsWhenReceivedRswETHLessThanMinimum() (gas: 335433)
[PASS] testDepositLSTWhenWhitelisted() (gas: 421819)
[PASS] testDepositManagerInit() (gas: 52063)
[PASS] testFallback() (gas: 19941)
[PASS] testFunctionsFailWhenNotCalledByAdmin() (gas: 718071)
[PASS] testPartialWithdraw() (gas: 2035424)
[PASS] testQueueWithdrawals() (gas: 1360098)
[PASS] testQueueWithdrawalsFailsOnInvalidStakerId() (gas: 41880)
[PASS] testReceive() (gas: 28081)
[PASS] testSetAdminSignerFailsOnAddressZero() (gas: 33628)
[PASS] testSetAdminSignerFailsOnInvalidCaller() (gas: 93292)
[PASS] testSetAdminSignerSuccess() (gas: 45992)
[PASS] testSetEigenLayerStrategyFailsWithAddressZero() (gas: 62633)
```

```
[PASS] testSetEigenLayerStrategyFailsWithInvalidCaller() (gas: 97229)
[PASS] testSetEigenLayerStrategyFailsWithInvalidToken() (gas: 50771)
[PASS] testSetEigenLayerStrategySuccess() (gas: 80653)
[PASS] testSetExchangeRateProviderContractSuccess() (gas: 67822)
[PASS] testSetExchangeRateProviderFailsOnInvaliCaller() (gas: 97120)
[PASS] testSetExchangeRateProviderFailsOnZeroAddress() (gas: 62459)
[PASS] testStakeOnEigenLayerFailsWhenBotMethodsPaused() (gas: 444093)
[PASS] testStakeOnEigenLayerFailsWhenNotEnoughETH() (gas: 472597)
[PASS] testStakeOnEigenLayerFailsWithIncorrectCaller() (gas: 485854)
[PASS] testStakeOnEigenLayerFailsWithInvalidDepositDataRoot() (gas: 448501)
[PASS] testStakeOnEigenLayerFailsWithInvalidStakerId() (gas: 577675)
[PASS] testStakeOnEigenLayerSuccess() (gas: 2307593)
[PASS] testStakeOnEigenLayerWithArrayMismatch() (gas: 437970)
[PASS] testStakeOnEigenLayerWithMultipleStakerIds() (gas: 4415258)
[PASS] testStakeOnEigenLayerWithZeroStakerIds() (gas: 437608)
[PASS] testUndelegateFailsWithInvalidCaller() (gas: 92742)
[PASS] testUndelegateFailsWithInvalidStaker() (gas: 35748)
[PASS] testUndelegateSuccess() (gas: 1192144)
[PASS] testUndelegateSuccessWithMultipleDelegatedStakers() (gas: 2697045)
[PASS] testVerifyAndProcessWithdrawals() (gas: 1795537)
[PASS] testVerifyAndProcessWithdrawalsFailsOnInvalidStakerId() (gas: 637259)
[PASS] testVerifyWithdrawalCredentials() (gas: 1304169)
[PASS] testVerifyWithdrawalCredentialsFailsWithNonExistantStaker() (gas: 182394)
[PASS] testWithdrawERC20FailsWhenNoTokens() (gas: 41171)
[PASS] testWithdrawERC20FailsWithIncorrectCaller() (gas: 131680)
[PASS] testWithdrawERC20Success() (gas: 59032)
[PASS] testclaimDelayedWithdrawalsFailsOnAddressZero() (gas: 33771)
[PASS] testclaimDelayedWithdrawalsFailsOnInvalidCaller() (gas: 95112)
Suite result: ok. 67 passed; 0 failed; 0 skipped; finished in 7.07s (50.48ms CPU time)

Ran 2 tests for test/Foundry/RateProviders/WbETHRateProvider.t.sol:WbETHRateProviderTest
[PASS] testGetRateMainnetForkSuccess() (gas: 787580)
[PASS] testWbETHRateProviderInitialize() (gas: 17127)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 7.03s (6.11s CPU time)

Ran 2 tests for test/Foundry/RateProviders/WstETHRateProvider.t.sol:WstETHRateProviderTest
[PASS] testGetRateMainnetForkSuccess() (gas: 814824)
[PASS] testWstETHRateProviderInitialize() (gas: 17182)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 7.01s (6.11s CPU time)

Ran 53 tests for test/Foundry/RswEXIT.t.sol:RswEXITTest
[PASS] testCreateWithdrawalRequestFailsWhenInvalidBalance() (gas: 256370)
[PASS] testCreateWithdrawalRequestFailsWhenTokenNotApproved() (gas: 112615)
[PASS] testCreateWithdrawalRequestFailsWhenWithdrawalsArePaused() (gas: 195625)
[PASS] testCreateWithdrawalRequestFailsWithInvalidWIthdrawalAmounts() (gas: 250143)
[PASS] testCreateWithdrawalRequestFailsWithUnwhitelistedCaller() (gas: 212418)
[PASS] testCreateWithdrawalRequestSuccess() (gas: 608632)
[PASS] testCreateWithdrawalRequestUsesCorrectRateAfterReprice() (gas: 603104)
[PASS] testCreateWithdrawalRequestWhitelistSuccess() (gas: 533318)
[PASS] testFinalizeWithdrawalsDoesNotOverwrightRequestsWhenTotalSupplyDecreased() (gas: 1113961)
[PASS] testFinalizeWithdrawalsEmitsEvent() (gas: 556032)
[PASS] testFinalizeWithdrawalsFailsRequestNotProccessed() (gas: 483800)
[PASS] testFinalizeWithdrawalsFailsWhenWithdrawalDoesNotExist() (gas: 35294)
[PASS] testFinalizeWithdrawalsFailsWhenWithdrawalsPaused() (gas: 43842)
[PASS] testFinalizeWithdrawalsFailsWithIncorrectOwner() (gas: 483799)
[PASS] testFinalizeWithdrawalsFindsProcessedRateAfterManyProcessesAndHasntBeenClaimedForAWhile() (gas: 1593731)
[PASS] testFinalizeWithdrawalsFindsProcessedRateAfterManyProcessesAndItsJustBeenProcessed() (gas: 1662363)
[PASS] testFinalizeWithdrawalsMaintainsProcessedStateAfterClaim() (gas: 562052)
[PASS] testFinalizeWithdrawalsRevertsWhenClaimingWithdrawalForTokenAlreadyClaimed() (gas: 558383)
[PASS] testFinalizeWithdrawalsUsesCreatedRateWhenSmaller() (gas: 554313)
[PASS] testFinalizeWithdrawalsUsesProcessedRateWhenSmaller() (gas: 645258)
[PASS] testGetLastTokenIdProcessedReturnsCorrectlyWhenManyTokensProcessed() (gas: 1356324)
[PASS] testGetLastTokenIdProcessedReturnsCorrectlyWhenOneTokenProcessed() (gas: 604515)
[PASS] testGetLastTokenIdProcessedReturnsZeroWhenNoOtherTokensProcessed() (gas: 15201)
[PASS] testGetProcessedRateForTokenIdReturnsFalseForUnproccessedToken() (gas: 1844838)
[PASS] testGetProcessedRateForTokenIdReturnsFalseWhenNoTokensProcessed() (gas: 15449)
[PASS] testGetProcessedRateForTokenIdReturnsFalseWhenThisTokenNotProcessed() (gas: 1375342)
[PASS] testGetProcessedRateForTokenIdReturnsFalseWhenTokenIdDoesntExist() (gas: 15538)
[PASS] testGetProcessedRateForTokenIdReturnsTrueAndCorrectRateAfterWithdrawalRequestProccessed() (gas: 3759874)
[PASS] testGetProcessedRateForTokenIdReturnsTrueAndCorrectRateFirstToken() (gas: 1376822)
[PASS] testGetProcessedRateForTokenIdReturnsTrueAndCorrectRateLastToken() (gas: 1828304)
```

```
[PASS] testGetProcessedRateForTokenIdReturnsTrueAndCorrectRateMiddleToken() (gas: 1828392)
[PASS] testProccessWithdrawalsEmitsEvent() (gas: 1358370)
[PASS] testProcessWithdrawalsDeductsCorrectTotalExitingETHAmount() (gas: 1355677)
[PASS] testProcessWithdrawalsDoesNotOverrideRateWithSameLastTokenIdToProcess() (gas: 613151)
[PASS] testProcessWithdrawalsFailsOnIncorrectCaller() (gas: 92687)
[PASS] testProcessWithdrawalsFailsOnInvalidTokenId() (gas: 38111)
[PASS] testProcessWithdrawalsFailsTokenIdLessThanLastProcessedTokenId() (gas: 607537)
[PASS] testProcessWithdrawalsIncreasesTotalUnprocessedETHAmount() (gas: 1355677)
[PASS] testProcessWithdrawalsSuccessDuplicateLastTokenIdToProcess() (gas: 558881)
[PASS] testProcessWithdrawalsSuccessTokenIdZero() (gas: 35435)
[PASS] testProcessWithdrawalsUsesCreatedRateWhenLowerOfTwoRates() (gas: 1354524)
[PASS] testProcessWithdrawalsUsesProccessedRateWhenLowerOfTwoRates() (gas: 1478533)
[PASS] testProcessWithdrawalstransfersETHFromDepositManager() (gas: 1356568)
[PASS] testSetBaseURIFailsWIthInvalidCaller() (gas: 97549)
[PASS] testSetBaseURISucccess() (gas: 72470)
[PASS] testSetWithdrawRequestMaximumFailsMaxLessThanMin() (gas: 88912)
[PASS] testSetWithdrawRequestMaximumFailsOnInvalidCaller() (gas: 94775)
[PASS] testSetWithdrawRequestMaximumSuccess() (gas: 63728)
[PASS] testSetWithdrawRequestMinimumFailsMinGreaterThanMax() (gas: 35531)
[PASS] testSetWithdrawRequestMinimumFailsOnInvalidCaller() (gas: 94729)
[PASS] testSetWithdrawRequestMinimumSuccess() (gas: 89456)
[PASS] testTokenURIFailsForNonExistentToken() (gas: 22763)
[PASS] testTransferToZeroAddressFails() (gas: 456524)
Suite result: ok. 53 passed; 0 failed; 0 skipped; finished in 8.03s (25.69ms CPU time)

Ran 2 tests for test/Foundry/RateProviders/METHRateProvider.t.sol:METHRateProviderTest
[PASS] testGetRateMainnetForkSuccess() (gas: 847375)
[PASS] testMETHRateProviderInitialize() (gas: 17125)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 11.55s (10.55s CPU time)

Ran 6 tests for test/Foundry/RateProviders/SwETHRateProvider.t.sol:SwETHRateProviderTest
[PASS] testDepositSwETHFailsWhenNoRateContractSet() (gas: 189288)
[PASS] testDepositSwETHFailsWhenZeroAmount() (gas: 187054)
[PASS] testDepositSwETHLocalSuccess() (gas: 322197)
[PASS] testDepositSwETHMainnetSuccess() (gas: 13967100)
[PASS] testGetRateMainnetForkSuccess() (gas: 13732906)
[PASS] testSwETHRateProviderInitialize() (gas: 17203)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 12.67s (17.73s CPU time)

Ran 1 test for test/Foundry/DepositManagerLegacy.t.sol:DepositManagerLegacyTest
[PASS] testWithdrawNonBeaconChainETHFromDeprecatedPod() (gas: 225501)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 16.44s (2.83s CPU time)

Ran 19 test suites in 17.44s (108.37s CPU time): 180 tests passed, 0 failed, 0 skipped (180 total tests)
```

**Remarks about Swell test suite**

The **Swell** team has diligently crafted a test suite that encapsulates the fundamental workflows and integrations with EigenLayer contracts. These tests did not cover some specific complex scenarios of the contract's functionality, which led to issues pointed by Nethermind Security team. However, during the fixing process, the **Swell** team improved the test suite by providing additional tests to cover these cases.

# 9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

**Blockchain Security:** At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

**Blockchain Core Development:** Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

**DevOps and Infrastructure Management:** Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

**Cryptography Research:** At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

**Smart Contract Development & DeFi Research:** Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

**Our suite of L2 tooling:** Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

**Learn more about us at nethermind.io.**

## General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

## Disclaimer