PUBLIC

# Code Assessment

## of the Protocol Smart Contracts

May 02, 2024

Produced for

MOEBIUS

by

CHAINSECURITY

# Contents

# 1 Executive Summary

Dear all,

Thank you for trusting us to help Moebius with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Protocol according to Scope to support you in forming an opinion on their security risks.

Moebius implements a transferrable token that represents stake in EigenLayer. Liquid staking tokens and native tokens can be deposited into the protocol to mint such tokens. Deposited native tokens are handled custodially by the protocol's third party operators running Ethereum validators.

The most critical subjects covered in our audit are functional correctness and front-running resilience.

Front-running resilience is good as long as operations admins deploy validators with the appropriate arguments.

Functional correctness is high but some functionality is missing that will be added at a later stage (see Verified validator balance not counted).

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 2 |
| • Code Corrected | 2 |
| Low-Severity Findings | 11 |
| • Code Corrected | 9 |
| • Risk Accepted | 2 |

# 2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

## 2.1 Scope

The assessment was performed on the source code files inside the Protocol repository based on the documentation files. All files inside the `contracts` directory were in scope of this audit. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 04 April 2024 | 4d502594d75f36e20cc03ad5d3c9efd1c719791a | Initial Version |
| 2 | 25 April 2024 | 3aba59e690c8d66bc3bd5800bfba801debcc4ac9 | After Intermediate Report |
| 3 | 30 April 2024 | 3c9cdfac5784c8d4289a2d63176169c0cd6fc856 | Fixes After Discussion |

For the solidity smart contracts, the compiler version `0.8.21` was chosen.

### 2.1.1 Excluded from scope

Any contracts inside the repository that are not mentioned in Scope are not part of this assessment. All external libraries and imports are assumed to behave correctly according to their high-level specification, without unexpected side effects.

Tests and deployment scripts are excluded from the scope.

## 2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Moebius offers a system facilitating depositing into and withdrawing from the EigenLayer M1 protocol. Users willing to participate in the protocol can deposit either Liquid Staking Tokens (LSTs) or native funds (ETH) and receive a protocol-native token, named moETH, in return.

LSTs are directly deposited into EigenLayer to receive share. These shares start generating yields as soon as delegation is activated both on EigenLayer and Protocol depending on the operator the funds are allocated to.

Native tokens are accumulated until the 32 ETH threshold to spin up a validator is reached. Privileged operations admins can then deploy a validator and verify it - after it has been activated in the Beacon Chain - to mint shares on EigenLayer.

The current state of the system only implements deposits. Other functionalities like migration (in case the immutable `DepositManager` has to be upgraded) and withdrawals are not fully implemented during this review.

In what follows, we elaborate on different parts of the system and how they interact with each other.

### 2.2.1 MoebiusCNC and MoebiusBase

`MoebiusCNC` (Command and Control) implements the following functionalities:

1. Pausing/unpausing the contract.
2. Keeping an address repository.
3. Keeping a role repository.

`MoebiusBase` is a wrapper around `MoebiusCNC`. It performs access control through `MoebiusCNC` (using the address and role repositories), as well as providing the system with pausing functionality. All other contracts of the system inherit from this contract.

### 2.2.2 EigenLayerProxy

This contract acts as an adapter between EigenLayer M1 contracts and Moebius contracts and implements an upgradeable proxy pattern to allow logical upgrades. When a new Moebius node operator gets onboarded, a new `EigenLayerProxy` gets deployed and associated with this node operator. Node operators cannot control the flow of funds through EigenLayer proxies, nor can they claim any funds circulated in this system for themselves. They are responsible for running the validators associated with ETH deposits to EigenLayer.

Proxies can deposit funds into EigenLayer strategies that have been added to the `DepositManager`. Each proxy is always able to deposit into each onboarded strategy. The underlying balance of a proxy is then determined by summing up all balances currently held in each of the strategies.

`EigenLayerProxy` allows the following operations:

- `deprecateStrategy()`: The owner can deprecate a strategy for a given token and set a mapping from the new strategy to the old strategy. This is required in case a strategy is switched in the `DepositManager` (DM) (see below for details) and the EigenLayerProxy still holds funds in the old strategy. Multiple such mappings for one strategy are not supported.
- `clearDeprecationState()`: The owner can remove a deprecated strategy from being included in balance calculations once the proxy does not hold any balance in the strategy anymore.
- `deposit()`: As can be seen in DepositManager, the DM forwards deposited tokens/ETH to this contract. This function is access-controlled to allow only the DM to call it to deposit LSTs into EigenLayer.
- `stake()`: After accumulating 32 ETH, the DM calls this function to deploy a validator. The ETH are forwarded to the EigenLayer's `EigenPodManager` which deposits the tokens into the Beacon Chain deposit contract along with withdrawal credentials pointing to the associated `EigenPod` of the proxy.
- `verifyWithdrawalCredentials()`: After a validator is included in the Beacon Chain state root, the operations manager can call this function to verify the withdrawal credentials of the pod owner as well as the current (not effective) balance of the validator, and the provided proof of the ETH validator against the beacon chain state, by calling `EigenPod.verifyWithdrawalCredentialsAndBalance()` which is exposed in the M1 version of EigenLayer contracts.

ETH donations sent to an `EigenLayerProxy` are getting forwarded to the DM and counted as a reward to the system.

### 2.2.3 DepositManager

This contract provides the end-users with the required functionalities to deposit their funds (native or ERC20 tokens) into the protocol. It implements the following interface, based on the type of user allowed to call them:

1. Privileged:

- `migrate()`: Callable only by the owner of the Deposit Manager (DM). It transfers the ETH balance of the contract to the new destination. Furthermore, it transfers all support token balances to the owner.
- `recoverERC20()`: The owner of the DM can move the balance of any token to a preferred destination.
- `deployValidator()`: Any user holding the `OPERATIONS_ADMIN` role can call this function. It deploys a new validator by staking through an EigenLayer proxy. The operations admin calling this function should define its validator of interest by giving its public key (and the respective signature) as an input parameter to this function. Furthermore, the current Beacon Chain deposit contract state root is added to make sure that no new deposits happen between the creation and execution of a transaction containing this call. This is used to ensure that the transaction cannot be frontrun by the node operator holding the private key to the given validator public key which could result in the withdrawal credentials of the created validator being changed.
- `addToken()`: Called by the owner. It takes a token as well as the associated EigenLayer strategy and stores them in the contract. An invariant of the system is that EigenLayer proxies should all have an infinite approval for each added ERC20 token from DM. Hence, upon adding a token to the system, DM approves all already existing proxies to access infinite amounts of this ERC20 token.
- `addProxy()`: Only the owner can add new proxies to the system. As mentioned in the paragraph above, EigenLayer proxies should have infinite approval for each ERC20 token. Therefore, the newly added proxy receives approvals for all the already added tokens.
- `removeProxy()`: Callable only by the owner. To deactivate a proxy, its ETH as well as supported ERC20 tokens should be completely withdrawn beforehand. It removes this proxy from the local list of active proxies and revokes any approvals.

The owner of the system can also disable a token (stopping any deposits of this token) and set a strategy for a given token. Tokens and their associated strategies cannot be removed from the system once they have been added.

2. Non-privileged: End-users can deposit their funds into the system and receive moETH in return via:

   - `depositETH()`: This function accepts ETH deposits and mints the corresponding moETH.
   - `depositToken()`: It firstly transfers the defined amount of the tokens to the DM, and calculates the moETH amount to be minted. It then chooses an EigenLayer proxy with a minimum overall (including both ETH and ERC20) balance and deposits the tokens to it. Finally, it mints the required moETH. Please note, that in contrast to `depositETH()`, the deposited funds get directly forwarded to an EigenLayer proxy, while `depositETH()` holds the ETH in this contract until later an operations admin deploys a validator.

moETH are minted based on the current total supply of the token as well as the current total balances of the system denominated in ETH. These balances are calculated by iteration over all existing proxies and summing up their ETH balances as well as all LST balances (priced in ETH).

Furthermore, we assume that moETH itself cannot be restaked.

### 2.2.4 PriceProvider

As mentioned above, users can deposit ETH or ERC20 tokens and receive moETH in return. The amount of moETH minted depends on the value of the deposited tokens/funds relative to ETH. To find this relative value, the DM calls into the `PriceProvider`. This contract calls Chainlink oracles to provide prices, expressed in ETH, for assets. This contract implements the following functionalities:

- `fetchPrice()` / `fetchPrices()`: These functions fetch the last two rounds from the given Chainlink feed(s). If they are valid, not stale, and do not diverge by more than 50%, the newly fetched price is stored and returned. Tokens denominated in USD on Chainlink are converted to ETH denomination. Additionally, the prices of tokens not directly supported by Chainlink can be converted using a third-party oracle if available (e.g., wstETH's `stEthPerToken()` function can be used to convert stETH to wstETH).
- `readPrice()` / `readPrices()`: Similar to `fetchPrice()` but does not update the storage.

### 2.2.5 LoadBalancingRouter

Deposits into the DM are balanced between existing EigenLayer proxies using a router contract. Currently, this is done using the `LoadBalancingRouter` which selects the proxy with the lowest overall balance.

### 2.2.6 Changes in Version 2

Version 2 of the protocol implemented some changes compared to the state of this section:

1. `DepositManager` contains a list that specifies which addresses' ETH transfers to the contract are not counted as rewards.
2. `EigenLayerProxy` interacts with the EigenLayer M2 interface.
3. `EigenLayerProxy` implements the new function `delegateTo()` that allows operations admins to delegate all shares of a proxy to an EigenLayer operator.
4. `EigenLayerProxy` implements the new function `verifyAndProcessWithdrawals()` that allows operations admins to perform partial withdrawals of native tokens (i.e., beacon chain rewards).
5. `EigenLayerProxy` implements the new function `handleFailedStaking()` that allows operations admins to remove ETH of validators for which the verification failed from the system balances.
6. `EigenLayerProxy` implements the new function `withdrawNonBeaconChainETHBalanceWei()` that allows the owner to withdraw any ETH that have been sent to an owned `EigenPod` and are not associated with a validator.
7. `EigenLayerProxy` implements the new function `recoverTokens()` that allows the owner to withdraw any ERC-20 tokens that have been sent to an owned``EigenPod``.

### 2.2.7 Roles and Trust Assumptions

Owner of Deposit Manager: Fully trusted. This role can migrate the DM to a new address, as well as set strategies for assets, enable and disable assets, remove proxies, and withdraw any ERC20 tokens in the DM contract.

Owner of Proxy Factory: Fully trusted, as they can deploy new proxies and add them to the list of EigenLayer proxies in their corresponding DM. Users holding this role can also change the implementation contract of the proxies.

Owner of Proxies: Fully trusted as well, because they can deprecate strategies allowing them to change the underlying balance calculation.

Operations Admin: Fully trusted. After accumulating enough ETH to deploy a validator, the operations admin can transfer the required funds a chosen address. While this is supposed to be an EigenLayer proxy contract, the operations admin can theoretically forward the funds to any contract of their choosing. Apart from that, we assume the set of node operators and operations admins are disjoint as this would otherwise introduce more abuse possibilities.

Operations admins are capable of calling `EigenLayerProxy.verifyWithdrawalCredentials()`. Due to some missing functionality in the current state of the contracts, we assume that this function is not called until the functionality is added (see Verified validator balance not counted).

Guardian: Partially trusted, as they can pause the protocol.

Owner of Command and Control Contract: Fully trusted, because they are permitted to change roles and the address registry of the system.

Node operators: Partially trusted, as they have no stake in the system and are running validators which could get slashed if they misbehave. Fully trusted if the operations admins do not set frontrunning protection in their calls to `DepositManager.deployValidator()`.

Additionally, we assume that the system is correctly configured. Correct addresses are set in the CNC address repository, roles are assigned appropriately to the parties, Chainlink oracles in the price provider contract are set and configured (e.g., heartbeat, etc.) correctly and strategies are not replaced if an existing deprecation exists that still holds value.

We further assume that the first deposit in the protocol is performed by the team in order to mitigate risks of inflation attacks.

Most of the contracts in the scope are immutable to ensure predictability. However, EigenLayer proxies are deployed through a beacon proxy pattern.

The system is expected to only be used with standard and rebasing ERC20 tokens and that only single-token EigenLayer strategies are used.

In Version 2, delegation has been enabled. Since the system for detection of operator undelegations is not fully implemented yet, we assume that funds are not delegated at this point. |

# 3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

# 4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

| Likelihood | Impact | | |
|---|---|---|---|
| | High | Medium | Low |
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Low |

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

# 5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 2 |



## 5.1 Loss on Transfer

Correctness Low Version 1 Risk Accepted

*CS-MEOP-007*

Some LSTs (like stETH) are rebasing. Transferring these tokens can result in less tokens transferred than originally anticipated. For example, a transfer of stETH could result in a difference of up to 2 wei. Since a token deposit performs multiple such transfers, the actual amount transferred to EigenLayer could be multiple weis smaller than the amount used to calculate the minting amount for a user, resulting in small losses.

---

**Risk accepted:**

Moebius accepts the risk with the following statement:

> Losses are trivial.

## 5.2 Reward Frontrunning

Security Low Version 1 Risk Accepted

*CS-MEOP-011*

Rewards are distributed by donating ETH to the protocol (either by calling `processReward()` or sending ETH directly to the contract). Depending on the size of the donation, the call can be frontrun to gain instant yields. This is especially true if there will be a way to directly redeem the shares of moETH afterwards when withdrawal functionality is added.

---

**Risk accepted:**

Moebius accepts the risk with the following statement:

> It will be mitigated with delayed withdrawals.

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 2 |

- No Approvals to StrategyManager Code Corrected
- Validator Verification DoS Code Corrected

| | |
|---|---|
| Low-Severity Findings | 9 |

- DepositManager Not Locked After Migration Code Corrected
- ETH Deposits Are Not Routed Code Corrected
- Incorrect Chainlink Validation Code Corrected
- Incorrect Chainlink roundId Calculation Code Corrected
- Missing Checks Code Corrected
- No Safe Approvals Code Corrected
- Pending Validator ETH Cannot Be Removed Code Corrected
- Response Timeout Buffer Code Corrected
- Validity of Current Chainlink Response Not Checked Code Corrected

| | |
|---|---|
| Informational Findings | 8 |

- Unused Variables Code Corrected
- Wrong Comment of depositToken Code Corrected
- Missing Automatic Deprecation Code Corrected
- Price Deviation Not Calculated From Current Price Code Corrected
- Gas Optimizations Code Corrected
- Misleading Comment Code Corrected
- Code Copy Code Corrected
- Events Not Indexed Code Corrected

## 6.1 No Approvals to StrategyManager

Correctness Medium Version 1 Code Corrected

*CS-MEOP-002*

`EigenLayerProxy.deposit()` calls EigenLayer's `StrategyManager.depositIntoStrategy()` which transfers tokens from the proxy to the given strategy. The required approval for this operation is, however, not granted.

**Code corrected:**

Upon adding a token by calling `addToken()`, the proxy grants the strategy manager with an infinite approval by calling `approveStrategyManager()`.

## 6.2 Validator Verification DoS

Security | Medium | Version 1 | Code Corrected

*CS-MEOP-003*

`EigenLayerProxy.stake()` deposits ETH to EigenLayer (and in turn to the Beacon Chain deposit contract) in chunks of 32 ETH. These 32 ETH are then added to the `ethPendingVerification` variable. To activate a validator and earn EigenLayer shares, the validator must be verified on EigenLayer. This is done in a second step in the function `EigenLayerProxy.verifyWithdrawalCredentials()` once the validator public key is in the Beacon Chain state root. The function then proceeds to subtract the current validator balance from the `ethPendingVerification` variable.

This is problematic in the following cases:

1. The EigenLayer verification succeeds even when the validator balance has since decreased by at maximum 1 ETH (for example due to validator inactivity). This reduced amount is then locked forever in the variable.
2. Once a validator public key has been successfully added to the set of known validator keys, beacon chain nodes process further deposits to that validator without checking the signature supplied in the deposit transaction. This allows anyone to top-up validators and might result in an underflow in the function `verifyWithdrawalCredentials()` until the additional balance is distributed back to the associated `EigenPod` which can take up to 5 days.

---

**Code corrected:**

The `ethPendingVerification` variable is now always incremented and decremented by exactly 32 ETH in the described steps.

## 6.3 DepositManager Not Locked After Migration

Design | Low | Version 1 | Code Corrected

*CS-MEOP-004*

`DepositManager` allows the protocol owner to migrate the currently held ETH balance to a new contract. Since there is only protocol-wide pausing functionality, it is still possible to deposit tokens to this contract after a migration has taken place. Depending on the changes going to be implemented in the new `DepositManager` contract, this could be problematic either for the health of the protocol or for the users depositing into the old contract.

---

**Code corrected:**

Moebius has addressed this issue by defining a state variable `migrated` to `DepositManager`. After migration, this flag is set to true. Hence, the following modifier stops calls to `depositETH()` and `depositToken()`:

```
modifier whenNotMigrated() {
    if (migrated) revert DepositManagerInactive();
    _;
}
```

## 6.4 ETH Deposits Are Not Routed

Design | Low | Version 1 | Code Corrected

*CS-MEOP-020*

LST user deposits are assigned to the different `EigenLayerProxy` deployments (and thereby to different operators) using the configured router contract. This is not true for deposited ETH. The `OPERATIONS_ADMIN` decides to which operators these tokens are staked.

---

**Code corrected:**

Moebius has modified the `deployValidator()` function to forward the deposited ETH to the proxy with the lowest balance.

## 6.5 Incorrect Chainlink Validation

Correctness | Low | Version 1 | Code Corrected

*CS-MEOP-005*

`PriceProvider._isValidResponse()` checks that the `answer` of a given Chainlink price feed is non-zero. However, the prices can theoretically also be negative. This case is not covered.

---

**Code corrected:**

`_isValidResponse()` now checks that the `answer` is greater than zero.

## 6.6 Incorrect Chainlink roundId Calculation

Correctness | Low | Version 1 | Code Corrected

*CS-MEOP-006*

`PriceProvider._fetchPrevFeedResponse()` retrieves data for a `roundId` that is calculated in the following way:

```
_currentRoundId - 1
```

Chainlink feed round IDs, however, are not continuously increasing. As explained in Chainlink Docs, upon updating the underlying aggregator implementation, `phaseId` gets incremented, in which case the new `roundId` is not incremented by 1 but jumps significantly. It means that this function can temporarily result in a DoS until another price update is performed.

---

**Code corrected:**

The `aggregatorRoundId` is extracted out of the `roundId` of each call to a Chainlink oracle. In case the `phaseId` is increased and the corresponding new `aggregatorRoundId` of the phase starts at 1, the contract makes an exemption and does not check the previous `roundId`.

## 6.7 Missing Checks

Correctness | Low | Version 1 | Code Corrected

*CS-MEOP-008*

Some checks are missing that could result in invalid state. Here is a list of some examples:

1. `DepositManager.setStrategy()` does not check that a new strategy set for a certain token is valid (i.e., non-zero, valid EigenLayer strategy).
2. `EigenLayerProxy.deprecateStrategy()` does not check if there is an existing deprecation for a given strategy. If that existing deprecated strategy holds any value for the proxy, this can result in loss of funds.
3. `EigenLayerProxy.deprecateStrategy()` does not check whether a given index corresponds to the `canonicalStrategy`. Additionally, `deprecatedStrategy` is not checked to be non-zero. In any of these cases, the whole system might become DoSed.
4. `EigenLayerProxy.deprecateStrategy()` does not check that `canonicalStrategy` is not equal to `deprecatedStrategy`. In this case, funds would be counted double.
5. `PriceProvider._setOracle()` does not check that `sharePriceProvider` and `sharePriceDecimals` have been set if `sharePriceSignature` is set. This could potentially lead to DoS.

---

**Code corrected:**

While `DepositManager.setStrategy()` still allows to set the 0-address, the checks are now sufficient.

## 6.8 No Safe Approvals

Correctness | Low | Version 1 | Code Corrected

*CS-MEOP-009*

Some functions (e.g., `DepositManager.addToken()`) set approvals using the default ERC-20 `approve()` function. The return value is, however, not checked.

---

**Code corrected:**

`approve()` calls have been replaced with `safeApprove()` calls.

## 6.9 Pending Validator ETH Cannot Be Removed

Correctness | Low | Version 1 | Code Corrected

*CS-MEOP-010*

`EigenLayerProxy.stake()` adds 32 ETH to the variable `ethPendingVerification` for every validator that is created. This validator must then be verified on EigenLayer using the function `EigenLayerProxy.verifyWithdrawalCredentials()`. This verification can fail for two reasons:

1. The validator has been slashed (lost more than 1 ETH) in the time between instantiation and verification.
2. The `stake()` function has been called with an empty `validatedDepositRoot` and the transaction has been frontrun by the associated node operator resulting in wrong withdrawal credentials being added to the validator.

In both cases, there is currently no way to remove the added 32 ETH from `ethPendingVerification`.

---

**Code corrected:**

Moebius has implemented a function `handleFailedStaking()` - callable by the operations admin - that allows to adjust the pending ETH.

## 6.10 Response Timeout Buffer

Security Low Version 1 Code Corrected

*CS-MEOP-021*

`RESPONSE_TIMEOUT_BUFFER` in the `PriceProvider` is added to the actual `heartbeat` of a price feed. Assuming that the `heartbeat` is equal to a Chainlink oracle's heartbeat, the additional buffer decreases security as the staleness of a feed can only be detected an hour later than it becomes evident.

---

**Code corrected:**

`RESPONSE_TIMEOUT_BUFFER` has been removed.

## 6.11 Validity of Current Chainlink Response Not Checked

Correctness Low Version 1 Code Corrected

*CS-MEOP-012*

`PriceProvider._getUpdatedRecords()` checks the validity of Chainlink responses when calling `_processFeedResponses()` which in turn calls `_isFeedWorking()`. If this check fails, then the variable `isFeedWorking` of the respective oracle record is accordingly updated. This is, however, not the case if the current response's `roundId` is invalid because `_fetchFeedResponses()` always returns `updated = false` in the case that the `roundId` is smaller or equal to the last stored `roundId`:

```
currResponse = _fetchCurrentFeedResponse(oracle);
if (lastRoundId == 0 || currResponse.roundId > lastRoundId) {
    prevResponse = _fetchPrevFeedResponse(oracle, currResponse.roundId);
    updated = true;
}
```

This results in `_getUpdatedRecords()` not returning an `action` indicating that the oracle state must be updated even though it should be.

**Code corrected:**

`updated` is set to true if for any reason the `currResponse.roundId` is not equal to `lastRoundId`.

## 6.12 Code Copy

Informational | Version 1 | Code Corrected

*CS-MEOP-013*

The functionality of `EigenLayerProxy.getTokenBalance()` is replicated in `EigenLayerProxy.getBalances()`. `getBalances()` could directly call `getTokenBalance()` in order to avoid maintenance problems in future revisions.

---

**Code corrected:**

`EigenLayerProxy.getBalances()` now calls `getTokenBalance()`.

## 6.13 Events Not Indexed

Informational | Version 1 | Code Corrected

*CS-MEOP-014*

Event parameters are not indexed in cases where it would make sense. This makes it harder for off-chain applications to gather data about the state changes in the contracts. Following is a non-exhaustive list of such parameters:

- `StrategySet`: `asset` and `strategy`
- `DepositManagerMigrated`: `newDepositManager`
- `TokenAdded`: `token` and `strategy`
- `ProxyAdded` and `ProxyRemoved`: `proxy`
- `AssetStatusSet`: `token`
- `RewardReceived`: `sender`
- `StrategyDeprecated`: `tokenIdx`, `oldStrategy`, and `newStrategy`
- `DeprecationStateCleared`: `tokenIdx`, `canonicalStrategy`, and `deprecatedStrategy`
- `EigenPodCreated`: `eigenPod`
- `AddressSet`: `key` and `value`
- `roleSet`: `key` and `account`
- `NewOracleRegistered`: `token` and `chainlinkAggregator`
- `PriceFeedStatusUpdated`: `token` and `oracle`

---

**Code corrected:**

Apaert from `AssetStatusSet`, all events have been indexed correctly.

## 6.14 Gas Optimizations

Informational | Version 1 | Code Corrected

*CS-MEOP-015*

The following parts of the code could be optimized:

1. Since strategies cannot be removed from the `DepositManager` and the order of the strategies is important for the deprecation feature in the `EigenLayerProxy`, each proxy always iterates over all strategies even if the associated token is disabled and the proxy does not hold any funds in the strategy.
2. `DepositManager.depositToken()` could transfer the tokens straight to the respective `EigenLayerProxy` (after the mint amount is calculated).
3. The updates in `PriceProvider` to the `lastUpdated` timestamp in each block only make sense if the oracle is called multiple times per block. Otherwise, it is cheaper to remove them in order to save gas on the extra storage writes that occur every block the oracle is called.
4. `DepositManager._getSystemBalances()` unnecessarily copies `tokenBalances[j]` to a local variable `tokenBalance`.
5. `DepositManager._calculateMintAmount()` checks if `amount` is less than `minDeposit`. This check can be done before fetching the prices.

---

**Code corrected:**

Some of the aforementioned optimizations have been implemented.

## 6.15 Misleading Comment

Informational | Version 1 | Code Corrected

*CS-MEOP-016*

Doc comments of the receive function in `EigenLayerProxy` state the following:

> Handles incoming ETH payments, routing them based on the sender or purpose.

The function, however, does not perform such routing.

---

**Code corrected:**

This comment has been removed and the functionality of the `receive()` function has been updated to count rewards based on the `senderStatus`.

## 6.16 Missing Automatic Deprecation

Informational | Version 1 | Code Corrected

*CS-MEOP-017*

When a strategy is replaced with another one in the `DepositManager`, the replaced strategies are no longer accounted for in the proxies. For this reason, each `EigenLayerProxy` allows the protocol owner to deprecate a strategy so that it can still be used for balance calculations until it has been emptied completely. This process has to be done for all existing proxies and is error-prone. If not all deprecations are set up properly, the system loses value resulting in cheaper shares for new depositors.

`DepositManager.setStrategy()` should therefore automatically deprecate a strategy in all proxies for which the strategy has a `userUnderlyingView()` greater than 0.

---

**Code corrected:**

`DepositManager.setStrategy()` now automatically deprecates strategies on each proxy. The proxies, in turn, only deprecate the strategies if they still hold value in them.

## 6.17 Price Deviation Not Calculated From Current Price

Informational Version 1 Code Corrected

*CS-MEOP-022*

`PriceProvider._isPriceChangeAboveMaxDeviation()` calculates the deviation always denominated in the `maxPrice` even though that results in different outcomes depending from which direction the price is moving.

---

**Code corrected:**

The max deviation is now always denominated in the current price.

## 6.18 Unused Variables

Informational Version 1 Code Corrected

*CS-MEOP-019*

The following variables are not used anywhere:

1. `DepositManager.maxNodeDelegatorLimit`
2. `Roles.MINTER`

---

**Code corrected:**

The unused variables have been removed.

## 6.19 Wrong Comment of depositToken

Informational Version 1 Code Corrected

*CS-MEOP-018*

Doc comments of `DepositManager.depositToken()` read:

> Accepts a token deposit, mints moETH to the depositor, and optionally deposits the token into a selected proxy.

However, depositing a token to a proxy is not optional, as either there exists a proxy or the call reverts.

---

**Code corrected:**

The following part has been removed from the comment: *and optionally deposits the token into a selected proxy*.

# 7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

## 7.1 LST Decimals

Note Version 1

The contracts are designed with the assumption that all LSTs are generally using 18 decimals. Special care has to be taken when onboarding new LSTs as there is a chance (even though it might be slim) that an LST uses a different decimal count.

## 7.2 Lower Capital Efficiency With ETH

Note Version 1

Protocol allows users to deposit LSTs as well as native ETH to the protocol. While LSTs are counted 1:1, ETH deposited to EigenLayer only mint shares for 31 ETH even though 32 ETH are deposited (per validator). This results in a slightly lower capital efficiency of native ETH deposits in comparison to LST deposits.

## 7.3 Operator Undelegation

Note Version 2

Since Version 2, Protocol interacts with version M2 of the EigenLayer contracts and allows for delegation to EigenLayer operators. In M2, operators are able to undelegate all shares of a delegatee (forcing a withdrawal). Since this undelegation would result in decreased system balances, Protocol reverts in case this is detected.

It is worth to note that, at the time of this report, this mechanism is not fully implemented.

It is further worth to note that, upon full implementation, operators can temporarily DoS the Protocol contracts.

## 7.4 Verified Validator Balance Not Counted

Note Version 1

Protocol, in its current state, does not support the verification of validators on EigenLayer even though the function `verifyWithdrawalCredentials()` is already implemented. It is of importance that the operations admin does not call this function until the correct counting of the ETH balance of verified validators has been implemented to prevent the following issue from happening:

The value of shares in Protocol is calculated by dividing the total amount of currently held tokens (denominated in ETH) by the total supply of the `MOETH` contract. This total amount is calculated in the function `DepositManager._getSystemBalances()` by iterating over all proxies and accumulating their stake in the associated EigenLayer strategies as well as their currently held ETH balance.

The ETH balance in a proxy is calculated in the function `EigenLayerProxy.getETHBalance()`:

```
function getETHBalance() public view returns (uint256) {
    return ethPendingVerification + ethInFlight + address(eigenPod).balance;
}
```

Once a validator is deployed, the respective 32 ETH are added to the variable `ethPendingVerification` and the tokens themselves are sent to the `EigenPod` which in turn deposits them to the Beacon Chain deposit contract. The function now returns a value of 32 ETH. When the validator is activated, it must further be verified using the function `EigenLayerProxy.verifyWithdrawalCredentials()`. This function now removes the validator balance from the `ethPendingVerification` variable, resulting in `getETHBalance()` returning 0 (the `EigenPod` balance only contains the accrued rewards except when the node operator exits the validator).

This means that, after a validator has been successfully deployed, the corresponding ETH are removed from the overall value calculation, resulting in reduced share prices.