OpenZeppelin | security

Restakefi Audit

January 19, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2023-12-06 To 2023-12-15
Languages	Solidity

Total Issues	27 (9 resolved, 3 partially resolved)
Critical Severity Issues	2 (1 resolved, 1 mitigated)
High Severity Issues	1 (0 resolved)
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	8 (4 resolved, 1 partially resolved)
Notes & Additional Information	14 (2 resolved, 2 partially resolved)
Client Reported Issues	0 (0 resolved)

Scope

We audited the DigitalMOB2/refi-protocol repository at the `dc842fd071f225c9d1ff9ad4677b986970125cf9` commit.

In scope were the following files:

```
contracts
├── Controller.sol
├── EigenSharePricer.sol
├── StakerM1.sol
├── interfaces
│   ├── IController.sol
│   ├── IERC20SnapshotRebase.sol
│   ├── IStaker.sol
│   └── external
│       └── eigenlayer
│           └── M1
│               ├── IDelegationManager.sol
│               ├── ISignatureUtils.sol
│               └── IStrategy.sol
├── storage
│   ├── StorageBaseERC20.sol
│   ├── StorageController.sol
│   ├── StorageERC20Snapshot.sol
│   ├── StorageERC20SnapshotRebase.sol
│   └── StorageStaker.sol
└── tokens
    ├── BaseERC20.sol
    ├── ERC20Snapshot.sol
    └── ERC20SnapshotRebase.sol
```

System Overview

Restake Finance is a modular liquid staking protocol developed on EigenLayer, utilizing users' liquid staking tokens (LSTs) as collateral in crypto-economic security for EigenLayer's validated services. This approach, known as rehypothecation, allows depositors to earn both Ethereum staking rewards and EigenLayer's native rewards. Upon depositing, users receive restaked ETH (rstETH) shares, issued at a 1:1 ratio. These pooled LSTs are then managed by DAO-controlled smart contracts, facilitating deposits into EigenLayer. This process offers users the flexibility to restake their LSTs liquidly without locking their assets.

EigenLayer introduces restaking, a novel crypto-economic security primitive within the Ethereum ecosystem. It enables ETH holders, whether staking natively or through LSTs, to participate in restaking via EigenLayer's smart contracts. This process not only secures the network but also opens avenues for additional rewards.

RestakeFi builds upon EigenLayer to offer a streamlined, user-friendly interface for earning rewards from LSTs. By automating the operator selection process, the system simplifies the experience for users. The `StrategyManager`'s expertise guides fund allocation, creating a hassle-free and efficient path to reward generation for users.

Currently, RestakeFi exclusively supports liquid restaking and interacts with EigenLayer M1 contracts.

User Interactions With the Protocol

Protocol Deposits

In the `Controller` contract, users can deposit underlying assets into the protocol, which are then allocated to EigenLayer by the `StrategyManager`. In response, the `Controller` grants protocol token shares to the users, maintaining a balanced ratio between the deposited assets and the equivalent token balance in the protocol. The share allocation for users is determined through the `balanceToShares` process in the `ERC20SnapshotRebase` contract, initiated by the `_update` mechanism, itself set in motion by the `mint` activity of the `BaseERC20` contract.

Protocol Withdrawals

To withdraw assets from the RestakeFi system, users must follow a two-step process. The first step involves requesting a withdrawal using the `requestWithdraw` function, specifying the desired amount in underlying units. This amount is then converted into shares and, if all balance checks pass, transferred back to the `Controller` contract. The request is subsequently added to the pending list, awaiting processing by the StrategyManager or any user. Each request is tagged with a unique ID. Once the withdrawals are processed, users can claim their underlying assets using the `redeemUnderlying` function, passing their request ID as an argument. This function performs several security checks, such as verifying the caller's ownership of the request ID. It then converts the shares tied to the request into underlying units and transfers this amount to the user. To preserve the correct balance-to-shares ratio, the shares are then burned after the transfer.

Interactions With EigenLayer

The `StrategyManager` is in charge of depositing the `Controller`'s underlying balance into one of the associated Staker contracts with the `staker_deposit` function. This amount is then directly transferred to Eigenlayer's contracts.

The `StrategyManager` is also in charge of checking whether the `Controller` has enough funds to process protocol users' withdrawals. In case it does not, they have the power to request and process withdrawals to EigenLayer through the `staker_queueWithdrawal` and `staker_completeWithdrawal` functions. Additionally, there might be situations where the `Controller` does not have sufficient funds to fulfill protocol users' withdrawal requests, but the Staker contracts do, without needing to withdraw them from EigenLayer. In such cases, the `StrategyManager` or any user can invoke the `staker_pullUnderlying` function, which will transfer the tokens held by a given Staker to the `Controller`.

Finally, as mentioned in the **Privileged Roles** section, the `StrategyManager` is responsible for delegating to an operator and terminating delegation.

Protocol Token

The Protocol Token is an upgradeable token that inherits the basic functionality from OpenZeppelin's `ERC20Upgradeable` contract. On top of that, the token follows the following inheritance chain and functionalities:

- `BaseERC20`: includes the token setup based on the roles and feature parameters. Adds access-controlled functionalities to mint, transfer, and burn associated shares to a defined underlying amount, and the pausability feature for the respective role.
- `ERC20Snapshot`: keeps a record of every fund-movement change into different checkpoints sorted by their timestamp. Implements the respective functionalities to query, update, and search checkpoints, while still keeping the original storage layout from the `ERC20Upgradeable` contract for balances.
- `ERC20SnapshotRebase`: adds the functionality to interact with an external oracle for the price of the share. Introduces a couple of access-controlled functions to mint and burn shares directly, and rewrites logic to work in terms of shares (rebased token). Part of the implementation overrides logic from previous contracts. This would be the uppermost layer that will be used as the entry point for the `Controller` contract.

Every contract mentioned above uses its own storage layout at a different slot in storage by following the EIP-7201 standard. No initial tokens will be minted on deployment and initialization.

Caveats of the Most External API

Due to the rebase nature of the Protocol Token, also known as Shares, the internal accounting inherited from the `ERC20Upgradeable` contract has been changed mainly in the `ERC20SnapshotRebase` contract. The `Controller` contract uses 2 different units when interacting with the Protocol Token, depending on the method used, but as the accounting system only keeps track of a single unit (including the checkpoints), the `ERC20SnapshotRebase` contract must convert some parameters from underlying into shares and vice-versa.

This means that this contract on the top layer completely overrides the `_update` function used by most of the traditional ERC-20 methods and converts its `value` input from underlying units into share units. The calculation depends on the `sharePricer` used, and currently uses the total supply of shares and total underlying tokens deposited into the protocol. Due to this change in units, and because the Protocol Token keeps track of shares and not the underlying tokens, the `_update` function calls the `_updateShares` function that updates the balances in storage and creates a checkpoint using the internal function from the `ERC20Snapshot` contract. It is worth mentioning that the `_update` function from the `ERC20Snapshot` and `BaseERC20` contract, which implement the checkpoint record and the pausability check, are bypassed and not used by the uppermost `_update` function. That means that the pausability feature is then added by the usage of the `whenNotPaused` modifier.

However, because the `Controller` contract still needs to query data from the underlying units, in a few cases, the contract also overrides the most common getters to re-convert the share balances into the current underlying units. All of these converted methods keep the traditional naming scheme intended by the ERC-20 standard. This convoluted workaround can introduce errors when not handled correctly, and it is also harder for users or developers to interact with.

Integration With the Protocol Token

As mentioned, the Protocol Token alters how the storage layout behaves to accommodate the units in shares but still handles operations in underlying units by the top layer change in the `_update` function. However, because ERC20 methods such as the transfer function make use of this function, a user transferring shares will actually input the value in underlying units, to then be converted into shares, which will become the actual value transferred.

If protocols want to integrate with such a token, they must be aware that its storage is denominated in shares and not in the value passed by the `transfer` or `balanceOf` functions.

Moreover, there are functions denominated in the underlying asset that do not go through the `_update` function, such as the `approve` function. The `allowance` mapping in storage (from the `ERC20Upgradeable` contract) will be denominated in underlying units while the `_balances` mapping in storage will be in share units. This could cause confusion if the storage is read for any reason (e.g., debugging) without using the getter functions.

Possible Flexibility Due to Initialization

During the initialization of the Protocol Token, the deployer has the flexibility to set specific features and roles. Such features, when enabled, allow the holder of the admin role to grant the designated role for said feature. If the feature is not set during initialization, it is not changeable unless upgraded. The features are as follows:

- `mintable` allows the token to be minted by the holder of the minter role.
- `pausable` allows the token to be paused or unpaused by the holder of the pauser role or unpauser role respectively.

- `initialSupply` and `initialSupplyHolder` go hand-in-hand with each other. If a value greater than zero is used for `initialSupply` and `initialSupplyHolder` is a non-zero address, the `initialSupplyHolder` will be minted the `initialSupply`.
- `allowControlled` allows the token to be controlled by the holder of the `Controller` role. Giving them the ability to use `controlledTransferFrom`, `controlledBurnFrom`, and `controlledBurnSharesFrom`.

Security Model and Trust Assumptions

The `StrategyManager` is a centralized role. We assume that this role is trusted and correctly executes a series of actions, ranging from depositing funds into Eigenlayer to withdrawing funds from it. Additionally, we assume all roles mentioned below are also trustworthy.

Privileged Roles

There are several privileged entities and roles that have access to sensitive operations.

Admin Role

- Can remove an existing privileged role
- Can assign a role to a specific address
- Can revoke a role from a specific address
- Can set the maximum amount of Staker contracts linked to the `Controller` contract

Strategy Manager Role

- Can process requested withdrawals
- Can deposit the underlying balance of the system into EigenLayer
- Can pull the underlying balance from a `Staker` contract to the `Controller`
- Can delegate the entire stake of a `Staker` to a specified operator
- Can undelegate from a Staker
- Can queue withdrawals from EigenLayer for a given Staker
- Can complete withdrawals from EigenLayer for a given Staker
- Can add new Stakers to the system
- Can remove existing Stakers from the system

Several of the functionalities mentioned above are not exclusively restricted to the `StrategyManager`. However, even when these functions are paused, the `StrategyManager` retains the ability to use them regardless.

Pauser Role: Can pause the protocol. For the `Controller`, the system defines different pausing features:

- *All*: All functions that implement pausing get paused
- *Deposit*: For deposit-related functionality, specifically the `deposit` function in the Controller contract
- *RequestWithdraw*: For withdrawal requests, specifically the `requestWithdraw` function in the Controller
- *ProcessWithdrawalsUser*: For withdrawal processing, specifically the `processWithdrawals` function
- *StakerPullUnderlyingUser*: For the process of pulling underlying assets from a Staker to the `Controller`
- *StakerCompleteWithdrawalUser*: For the process of completing withdrawals from EigenLayer

Additionally, the pauser can also pause all the functions if the protocol contract that calls the `_update` hook.

Unpauser Role: Can unpause any of the aforementioned pausing features

Upgrader Role: Can upgrade the protocol token, the `StakerM1` contract and the `Controller` contract.

Minter Role: Can mint protocol tokens (see below)

Controller Role: Can burn protocol tokens (see below)

Manager Role: The `ERC20SnapshotRebase`, built upon `BaseERC20`, includes a manager role that has the ability to change or remove the `SharePricer` for the protocol token.

Further Information on Privileged Roles

The deployer of the `ERC20SnapshotRebase` (the uppermost layer contract) will be granted the admin role. They will have full control to either grant or revoke any role. There are several privileged roles identified where most roles are located in `BaseERC20`.

The upgrader role will be able to upgrade the contract to a new implementation.

With the pauser role, they will be able to pause the share token and, vice versa, the unpauser role will be able to unpause the share token. This is only the case if the pausable feature is enabled during initialization.

The minter role, granted only to the `Controller` contract, will be able to mint new shares upon user deposits of the underlying token.

The `Controller` role is also granted to the `Controller` contracts and can do the following:

- Burn shares from users who are redeeming them for the underlying tokens with `controlledBurnSharesFrom`
- Burn an amount from users that is converted to shares with `controlledBurnFrom`, which is currently unused by the `Controller`
- Transfer shares from the `StakerM1` to the `Controller` when a user is requesting a withdrawal of the underlying tokens with `controlledTransferFrom`

`ERC20SnapshotRebase`, built upon `BaseERC20`, includes a manager role that has the ability to change or remove the `SharePricer` for the protocol token.

Update: *The Restakefi team provided us with pull requests addressing the issues presented in this report. Each issue was updated to reference the relevant pull request that addressed the issue. Pull request #11 has been merged after the audit started and used to create all fixes. While reviewing fixes to issues presented in this report, only changes related to the specific issue were reviewed and any unrelated modifications to the codebase were not. Thus, we strongly recommend the codebase undergoes a future code review that includes all changes to mitigate the risk of potential vulnerabilities resulting from the out-of-scope changes, taking into account that such changes may affect fixes from issues or other logic implemented in the code.*

Critical Severity

C-01 Attacker Can Downscale All Protocol Shares by 18 Decimals

The `deposit` function of the `Controller` contract enables users to deposit underlying assets into the protocol, which are subsequently deposited into EigenLayer by the `StrategyManager`. In return, the Controller mints protocol token shares for the user, maintaining a 1:1 ratio between the underlying deposited token and the underlying token balance of the protocol token. The shares minted for the user are calculated in the `balanceToShares` function of the `ERC20SnapshotRebase` contract, invoked by the `_update` hook triggered by the `mint` function of the `BaseERC20` contract.

The `balanceToShares` function relies on two other functions: `_sharesPrecision` and `_shareValue`. If the share price contract is set, `_sharesPrecision` returns a fixed value of `1e18`, and `_shareValue` returns either `1e18` or the ratio between the total underlying token deposited in the controller and the total amount of protocol token shares minted, defined here as:

```
share_price = controller_total_underlying * 1e18 / protocol_token_total_shares
```

The issue arises because the first depositor can manipulate both the underlying token balance held by the protocol (by either the controller or the staker contracts) and the protocol token's total share amount to *downscale* all future users' shares and the total shares stored in the contract by 18 decimals.

Example:

- Alice (malicious) deposits 1 underlying token into the protocol through the `deposit` function. As the protocol token's total shares are initially 0, the `_shareValue` function will return 1e18, as will the `_sharesPrecision` function, and Alice will be minted 1 share.
- Alice then transfers any amount with 18 decimals of precision (e.g., 10e18) of the underlying token to either the Controller or any of the Stakers. At this point, `total_shares = 1`, `alice_shares = 1`, and `total_underlying = 10e18 + 1`. Now Alice owns the protocol's only share **without decimal precision, which is unexpected**, representing the total underlying token balance held by the protocol (with 18 decimals of precision), as well as all future user shares. This leads to several undesired scenarios.

Loss of Funds

Any deposit amount lower than the underlying balance held by Alice that occurs after the sequence mentioned above will be owned by Alice.

Let's say Bob (an honest actor) deposits 10e18 underlying tokens into the protocol. Since the protocol token's total shares are now 1 (not 0), the `_shareValue` function will return a value with 36 decimals based on this formula:

```
controller_total_underlying = 10e18 + 1
protocol_token_total_shares = 1
```

```
share_price = controller_total_underlying * 1e18 / protocol_token_total_shares
share_price = ~10e18 * 1e18 / 1 = ~10e36 (> 10e36)
```

- The shares minted for Bob are calculated as:

```
bob_shares = amount * shares_precision / share_price
bob_shares = 10e18 * 1e18 / ~10e36 (>10e36) = 0
```

Bob deposits 10e18 underlying tokens but receives 0 shares in exchange, meaning the underlying tokens he deposited will belong to Alice.

Rounding Errors

In certain scenarios, due to rounding issues arising from the loss of 18 decimals of precision, depositors will lose part of their deposit shares. Following the aforementioned example where Alice deposits 1 token and then transfers 10e18 tokens to the Controller, if Bob subsequently deposits, for example, 15e18 tokens into the protocol, his shares will be calculated as follows:

```
share_price = controller_total_underlying * 1e18 / protocol_token_total_shares
share_price = ~10e18 * 1e18 / 1 = ~10e36 (> 10e36)```
bob_shares = 15e18 * 1e18 / ~10e36 = 1
```

Bob's deposit will represent 50% of the pool, instead of 60% (15e18 in a total of 25e18).

A step-by-step proof-of-concept can be found in this secret gist.

Consider tracking the balance of the underlying token in Staker contracts and the controller locally, and using these amounts instead of relying on the `balanceOf` function in the `getTotalUnderlying` function.

Update: *Mitigated in pull request #28 at commit 3b7214d. In pull request 28, the team introduced a sacrifice position, deposited by the* `Controller` *contract itself during initialization, which is then permanently locked. This strategy makes it highly impractical to return to a scenario where the underlying amount is much larger than the number of shares, and it will lead to a substantial loss for the attacker.*

It is also important to notice that a small downscale due to a rounding issue and a side transfer might occur. In such a scenario, a depositor could end up withdrawing less underlying than what was originally deposited. Nevertheless, the error produced is insignificant and cannot be mitigated in this type of setting.

Initially, the RestakeFi team added pull request #15 at commit 0b350a1, which included a minimum deposit requirement that took into account the decimal precision of the underlying *asset. This is to prevent users from minting an excessively small number of shares. However, this minimum deposit amount, as well as the minimum share quantity for initial depositors, could be manipulated, potentially reverting to the problematic scenario initially identified. The attack would consist of the following:*

The first depositor deposits the lowest permissible amount, say `1e13`, receiving an equivalent number of shares (`1e13`). In the same transaction, a malicious user could request a withdrawal of (1e13 - 1) of the underlying asset. They would then execute their withdrawal using the `processWithdrawals` function and subsequently redeem their underlying asset. At this point, the attack is back to the initial issue: the first depositor is left with just 1 share, and the value of the underlying asset in the `Controller` contract is reduced to 1.

Moreover, another attack could be made by directly calling either of the `burn` or `burnShares` functions (the latter one introduced in the same pull request as the fix for M02), which would allow the user to burn all but one shares directly without altering the underlying asset balance and return to the same attack scenario.

On a different subject, we noted that after both fixes, a downscaling of approximately `1e-5` still occurs during transfers of the underlying asset between deposits, even when the minimum deposit is enforced from the start of the attack. As a result, the percentage of share balances per user becomes roughly similar to those seen in normal operations, though they are not exactly identical. This situation does not appear to pose any significant risk at first glance. However, we have not yet exhaustively investigated all possible scenarios where these discrepancies might lead to more substantial impacts.

C-02 Shares Not Burned After Redemption of Underlying Assets

The asset withdrawal process from the protocol is a two-step procedure. Initially, depositors must request a withdrawal using the `requestWithdraw` function, requesting the amount in underlying units. This amount is then converted into shares and, if all balance checks pass, transferred back to the `Controller` contract. The request is subsequently added to the pending list, awaiting processing by the `StrategyManager`. Each request is assigned an ID and includes multiple parameters, such as the request owner and the amount of shares associated with that request.

Once the `StrategyManager` processes the withdrawals, users can redeem their underlying assets using the `redeemUnderlying` function, inputting the request ID. The function converts the shares tied to the request into underlying units using the sharesToBalance function, and then this amount of underlying is transferred to the user. The request is removed from the list of withdrawal requests and the function attempts to burn the shares associated with the request using the `controlledBurnSharesFrom` function, specifying the `req.shares` value involved as the burn amount.

However, as this request is removed from the list once the underlying assets are transferred to the user, the value of `req.shares` will be zero. Due to this, shares will never be burned and will remain stuck in the `Controller` contract.

Moreover, as all calculations for converting balances to shares (and vice versa) depend on the total supply of shares, the underlying balances will not properly reflect the conversion due to an always-increasing `totalShares` value.

A step-by-step proof-of-concept can be found in this secret gist.

Consider either burning the shares before removing the request from the list of withdrawal requests, or using a temporary variable to store the value that will later be burned.

Update: *Resolved in pull request #14 at commit 376f4e4.*

High Severity

H-01 Attacker Can DoS Withdrawals

For users who want to re-stake their tokens in EigenLayer using the protocol, the following steps should be followed:

- The user deposits an asset through the `deposit` function of the Controller contract. In return, the Controller mints protocol token shares for the user, maintaining a 1:1 ratio between the deposited underlying and the underlying token balance of the protocol token.
- After some time, the user might want to withdraw their funds (plus potential earnings) from the protocol. To do this, they first have to request a withdrawal through the `requestWithdraw` function in the Controller. This request gets queued in a double-linked list, where the user address, the nonce of the request, and the share amount are saved. These funds cannot be redeemed until the `StrategyManager` processes this user's withdrawal request.

- The `StrategyManager` processes this and potentially other withdrawal requests by calling the `processWithdrawals` function. Withdrawal requests are processed in order, as a FIFO (First In, First Out), which means that to process the user's withdrawal request, the `StrategyManager` has to process any remaining requests done previously by other users in order. An iterator is updated each time a withdrawal request is processed, so they don't get processed again. Before processing withdrawals, the `StrategyManager` must ensure enough funds in the Controller contract to pay for the selected batch of consecutive withdrawal requests.
- Finally, the user can call the `redeemUnderlying` function, where their protocol tokens get burned, and the protocol transfers the underlying shares to the user.

The issue lies in the fact that there are no restrictions on users making multiple withdrawal requests, which opens up griefing attack scenarios:

- An attacker could deposit a relatively low amount of underlying assets (e.g., 1,000 tokens) and use multicall to call `requestWithdrawal` 1,000 times, each time with 1 unit as a parameter (in different batches if necessary, in case they run out of gas). In this scenario, the `StrategyManager` would have to process 1,000 withdrawal requests before being able to address legitimate users' requests.
- Once the `StrategyManager` decides to fulfill all these requests, the attacker can then create another 1,000 withdrawal requests. In an extreme case, the attacker could continuously prevent the `StrategyManager` from catching up, forcing them to process thousands of malicious requests again before being able to fulfill the next batch of legitimate requests.
- In an extreme scenario, the attacker could initially create an even higher number of malicious withdrawal requests, compelling the protocol team to upgrade the protocol with a new design and pause withdrawal requests. This would prevent legitimate users from transferring their protocol tokens to the Controller contract, thereby virtually "losing" their positions.

These scenarios will make the protocol unusable for users since they will not be able to redeem their assets, or will have to wait until the `StrategyManager` processes malicious requests. and will incur financial expenses for the `StrategyManager` to be able to fulfill the malicious withdrawal requests, as well as fill the double-linked list with trash requests, hindering the task of off-chain services searching and filtering for specific requests.

Consider limiting the number of withdrawal requests that a user can make. Otherwise, consider adding a larger minimum amount that could be withdrawn by a user, for instance, 50% of their shares. Note that even with these mitigations, an attacker could still perform this attack using different accounts. We recommend designing a new withdrawal system where these scenarios are not possible.

Update: *Not resolved in [pull request #27]() at commit [d3ce3e7](). The RestakeFi team stated:*

> *We changed the logic to add up all the withdrawals from the same user in the same existing position until a strategy manager engages with the withdrawal logic. This change together with the fact that creating withdrawal requests is almost three times more expensive than processing them (iterating over them), and the minimum deposit amount, should solve this issue.*

The same attack is still possible, just with an extra step. The attacker now has to use multiple addresses to deposit and call `requestWithdrawal` 1,000 times to block future withdrawals. This also creates a new issue. An attacker can force users who have withdrawals that would be processed in the next batch to be delayed. This is possible due to an attacker who is front-running the `processWithdrawal` function, increasing their withdrawing amount and leaving the honest users' withdrawals unprocessed and delayed.

Moreover, the RestakeFi team stated:

> *The high-severity issue is also theoretical in a sense because it pivots on the existence of an irrational attacker willing to waste large amounts of funds to do a DDoS attack and grief withdrawals. It's a similar issue that can be brought up for almost any system - presuming some attacker is irrational and extremely well-funded can also apply to Ethereum itself for example. In our case, this issue can be dealt with for way cheaper by the strategy manager, as part of its responsibilities. the first in first out approach for the withdrawal queue - given the way they are implemented in Eigen layer - is the fairest approach. We would also like to mention that the system is upgradeable - so we can unlock funds if this becomes a practical issue.*

Medium Severity

M-01 DoS in Controller's `depositWithPermit` Function

The [`depositWithPermit` function]() in the Controller allows users to deposit a certain amount of underlying tokens without prior token approval, bypassing methods like `approve()` or `increaseAllowance()`. It does this by leveraging the `permit` functionality of the `ERC20Permit` contract.

However, the `depositWithPermit` function can be front-run by a malicious actor. This actor can observe the transaction in the mempool, replicate the parameters being sent, and directly invoke the `ERC20Permit#permit` function of the underlying token. As a result, the original user's call will fail.

Consider implementing a `try/catch` block within the `depositWithPermit` function. In this block, if the call to `permit` fails, the contract should execute the `deposit` anyway if the allowance is sufficient.

Update: *Resolved in pull request #16 at commit b80abef.*

M-02 Funds Can Get Stuck Due to Incorrect Initialization

The `BaseERC20` contract implements the functionality to set the available features in the Protocol Token based on the input passed by the admin, and it can only be done once.

However, if the `allowControlled` flag is set to `false`, then all the operations after the tokens have been deposited will not work and the underlying asset stored in the protocol will be stuck. It is worth mentioning that the `false` value is the default value and it could be passed by mistake, realizing this only when the withdrawals of funds are happening.

This happens because the `deposit` function from the `Controller` contract only needs to be the `MINTER_ROLE` role to successfully complete the deposit, but needs the `CONTROLLER_ROLE` role to then call the `controlledTransferFrom`, `controlledBurnFrom`, and `controlledBurnSharesFrom` functions. That means that after depositing funds, users will not be able to take back the underlying assets.

Moreover, as the flag is only set during the initialization, the admin will not be able to change it without a protocol upgrade. Furthermore, since it is always expected for the `Controller` contract to be in control of those functionalities (`controlledTransferFrom` and `controlledBurnSharesFrom` functions), the flag should always be up.

Consider removing this flag or always setting it to `true` to prevent funds from becoming stuck in the protocol. Moreover, consider checking that the passed `roles.controller` address is indeed a `Controller` type of contract with an ERC-165 or other introspection standard.

Update: *Resolved in pull request #17 at commit 80958e7. The flag no longer exists. However, the `controlledBurnSharesFrom` function has been renamed `burnShares` and its access control has been eliminated. This function resembles the `burn` function inherited by the `ERC20BurnableUpgradeable` contract, although in this case its value is not converted from underlying into shares by going through the `_update` function. It is worth noting that with both external functions, users can lose their positions as it does not process the underlying withdrawal after the shares are removed from their account, so users should not be allowed to call them.*

Low Severity

L-01 Protocol Token Cannot Be Unpaused

The `BaseERC20` contract implements the basic role management for the other contracts inheriting it.

In particular, it handles the `PAUSER_ROLE` role.

Although the `PAUSER_ROLE` is the one able to pause the token, another role (`UNPAUSER_ROLE` role) is the one in charge of unpausing it after it has been paused.

However, all other roles are granted in the `__BaseERC20_init_unchained` function, except for the `UNPAUSER_ROLE`, meaning that nobody will have the ability to call the `unpause` function.

Although the admin, currently a multisig controlled by the team, can later grant the role to a new address, it would be wise to prevent the situation of delaying the resuming of the protocol's activity due to the extra task of granting the role beforehand. Moreover, to reduce the attack surface and improve the readability of the code, it is suggested that all roles should have the same granting process.

Consider granting the `UNPAUSER_ROLE` role along with the rest of the roles in the `__BaseERC20_init_unchained` function.

Update: *Resolved in pull request #18 at commit d094552.*

L-02 Protocol Token Initialization Might Not Be Properly Configured

The `BaseERC20` contract uses flags to define certain characteristics and the roles available for the token.

However, there are situations where the inputs are not entirely being handled by the implementation, meaning that it may cause problems later on if the input was mistakenly crafted. For example, if a token is not meant to be paused, the pauser address should be zero so it is not skipped silently.

As a safety check, consider validating that addresses for their respective flags are zero for features that will not be used, and reverting the initialization in the alternate scenario.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *For our specific case, this is acceptable. We are setting all the flags we need.*

L-03 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

Throughout the codebase, there are multiple floating pragma directives:

- The file `BaseERC20.sol`
- The file `Controller.sol`
- The file `ERC20Snapshot.sol`
- The file `ERC20SnapshotRebase.sol`
- The file `EigenSharePricer.sol`
- The file `IController.sol`
- The file `IDelegationManager.sol`
- The file `IERC20SnapshotRebase.sol`
- The file `ISignatureUtils.sol`
- The file `IStaker.sol`
- The file `IStrategy.sol`
- The file `StakerM1.sol`
- The file `StorageBaseERC20.sol`
- The file `StorageController.sol`
- The file `StorageERC20Snapshot.sol`

- The file `StorageERC20SnapshotRebase.sol`
- The file `StorageStaker.sol`

Consider using a fixed pragma version on all files.

Update: *Resolved in pull request #19 at commit 70437a5.*

L-04 Improper Namespace NatSpec Tag for ERC-7201

Currently, `StorageController` does not follow the NatSpec tag specification that should be annotated to the contract's struct for ERC-7201. The `ControllerStorage` namespace tag is `@custom:storage-location erc7201:ProtocolStorage` and has the formula `keccak256(abi.encode(uint256(keccak256("ControllerStorage")) - 1)) & ~bytes32(uint256(0xff));`. This is incorrect according to ERC-7201:

> A namespace in a contract should be implemented as a struct type. These structs should be annotated with the NatSpec tag `@custom:storage-location <FORMULA_ID>:<NAMESPACE_ID>`, where `<FORMULA_ID>` identifies a formula used to compute the storage location where the namespace is rooted, based on the namespace id.

Consider changing the namespace tag to `@custom:storage-location erc7201:ControllerStorage` to properly follow the ERC-7201 specification.

Update: *Resolved in pull request #20 at commit c03320c.*

L-05 Missing Docstrings

Throughout the codebase, there are several parts that do not have docstrings:

- Line 15 in `BaseERC20.sol`
- Line 26 in `BaseERC20.sol`
- Line 27 in `BaseERC20.sol`
- Line 28 in `BaseERC20.sol`
- Line 29 in `BaseERC20.sol`
- Line 30 in `BaseERC20.sol`
- Line 48 in `BaseERC20.sol`
- Line 114 in `BaseERC20.sol`

- Line 164 in `BaseERC20.sol`
- Line 180 in `BaseERC20.sol`
- Line 195 in `BaseERC20.sol`
- Line 199 in `BaseERC20.sol`
- Line 203 in `BaseERC20.sol`
- Line 19 in `Controller.sol`
- Line 29 in `Controller.sol`
- Line 30 in `Controller.sol`
- Line 32 in `Controller.sol`
- Line 33 in `Controller.sol`
- Line 40 in `Controller.sol`
- Line 82 in `Controller.sol`
- Line 281 in `Controller.sol`
- Line 11 in `ERC20Snapshot.sol`
- Line 19 in `ERC20Snapshot.sol`
- Line 39 in `ERC20Snapshot.sol`
- Line 50 in `ERC20Snapshot.sol`
- Line 15 in `ERC20SnapshotRebase.sol`
- Line 21 in `ERC20SnapshotRebase.sol`
- Line 22 in `ERC20SnapshotRebase.sol`
- Line 24 in `ERC20SnapshotRebase.sol`
- Line 33 in `ERC20SnapshotRebase.sol`
- Line 42 in `ERC20SnapshotRebase.sol`
- Line 67 in `ERC20SnapshotRebase.sol`
- Line 86 in `ERC20SnapshotRebase.sol`
- Line 108 in `ERC20SnapshotRebase.sol`
- Line 117 in `ERC20SnapshotRebase.sol`
- Line 123 in `ERC20SnapshotRebase.sol`
- Line 130 in `ERC20SnapshotRebase.sol`
- Line 135 in `ERC20SnapshotRebase.sol`
- Line 159 in `ERC20SnapshotRebase.sol`
- Line 166 in `ERC20SnapshotRebase.sol`
- Line 172 in `ERC20SnapshotRebase.sol`
- Line 179 in `ERC20SnapshotRebase.sol`
- Line 185 in `ERC20SnapshotRebase.sol`
- Line 195 in `ERC20SnapshotRebase.sol`
- Line 8 in `EigenSharePricer.sol`
- Line 9 in `EigenSharePricer.sol`

- Line 11 in `EigenSharePricer.sol`
- Line 12 in `EigenSharePricer.sol`
- Line 19 in `EigenSharePricer.sol`
- Line 31 in `EigenSharePricer.sol`
- Line 8 in `IController.sol`
- Line 50 in `IController.sol`
- Line 51 in `IController.sol`
- Line 57 in `IController.sol`
- Line 62 in `IController.sol`
- Line 67 in `IController.sol`
- Line 68 in `IController.sol`
- Line 69 in `IController.sol`
- Line 70 in `IController.sol`
- Line 71 in `IController.sol`
- Line 73 in `IController.sol`
- Line 6 in `IERC20SnapshotRebase.sol`
- Line 17 in `IERC20SnapshotRebase.sol`
- Line 19 in `IERC20SnapshotRebase.sol`
- Line 21 in `IERC20SnapshotRebase.sol`
- Line 23 in `IERC20SnapshotRebase.sol`
- Line 25 in `IERC20SnapshotRebase.sol`
- Line 30 in `IERC20SnapshotRebase.sol`
- Line 32 in `IERC20SnapshotRebase.sol`
- Line 34 in `IERC20SnapshotRebase.sol`
- Line 9 in `IStaker.sol`
- Line 39 in `IStaker.sol`
- Line 40 in `IStaker.sol`
- Line 41 in `IStaker.sol`
- Line 42 in `IStaker.sol`
- Line 43 in `IStaker.sol`
- Line 44 in `IStaker.sol`
- Line 66 in `IStaker.sol`
- Line 68 in `IStaker.sol`
- Line 70 in `IStaker.sol`
- Line 72 in `IStaker.sol`
- Line 74 in `IStaker.sol`
- Line 76 in `IStaker.sol`
- Line 103 in `IStrategy.sol`

- Line 32 in `StakerM1.sol`
- Line 39 in `StakerM1.sol`
- Line 160 in `StakerM1.sol`
- Line 192 in `StakerM1.sol`
- Line 256 in `StakerM1.sol`
- Line 262 in `StakerM1.sol`
- Line 268 in `StakerM1.sol`
- Line 272 in `StakerM1.sol`
- Line 4 in `StorageBaseERC20.sol`
- Line 8 in `StorageController.sol`
- Line 39 in `StorageController.sol`
- Line 43 in `StorageController.sol`
- Line 47 in `StorageController.sol`
- Line 51 in `StorageController.sol`
- Line 55 in `StorageController.sol`
- Line 61 in `StorageController.sol`
- Line 65 in `StorageController.sol`
- Line 76 in `StorageController.sol`
- Line 82 in `StorageController.sol`
- Line 86 in `StorageController.sol`
- Line 105 in `StorageController.sol`
- Line 4 in `StorageERC20Snapshot.sol`
- Line 4 in `StorageERC20SnapshotRebase.sol`
- Line 6 in `StorageStaker.sol`
- Line 28 in `StorageStaker.sol`
- Line 32 in `StorageStaker.sol`
- Line 36 in `StorageStaker.sol`

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *Acknowledged. Not an issue for us.*

L-06 Confusing Usage of `Staker` Index

The `staker_deposit`, `staker_pullUnderlying`, `staker_delegateTo`, `staker_undelegate`, `staker_queueWithdrawal`, and `staker_completeWithdrawal` functions require both the `Staker` contract and the `index` as parameters to locate the staker in the array of Stakers in the Controller's storage. If the index is invalid, or if the staker provided as a parameter does not correspond to the one at the specified index, these functions will revert.

However, in the `removeStaker` function, if a staker is removed and is not the last in the array, the last element of the array replaces the removed Staker's position, and then the last element is removed. This process changes the position of the last staker in the array. As this change is not recorded in an event, it becomes challenging to track the new index of the relocated staker.

Consider emitting an additional event if a Staker is moved to a different position in the array. This will prevent any hindrance caused by a lack of visibility into the current indices of the Stakers.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *The index is used just for double-checking that it is the actual staker that the admin wants to remove. It is not used for anything else. Not an issue.*

L-07 Use of Boolean Literal as Conditional

Boolean literals in code have limited legitimate uses. Other uses (in complex expressions, as conditionals) indicate either an error or, most likely, the persistence of faulty code.

The boolean literal `False` within the `BaseERC20` contract in `BaseERC20.sol` is a misuse.

To avoid misusing Boolean literals, consider changing the style to maintain consistency within `if` clauses.

Update: *Resolved in pull request #21 at commit 7bb1659.*

L-08 Incorrect Value Emitted in Event

In the `controlledBurnSharesFrom` function, `updateShares` is called to update the balance of shares of the account from which the shares are being burned, and to adjust the total supply of shares, as this is a burning operation. However, an issue arises with an event that emits the amount of balance being burned, calculated using the `sharesToBalance` function. The problem is that `sharesToBalance` depends on the total supply of shares, which has just been updated. Therefore, the emitted value will be wrong. This pattern is also observed in other parts of the code (e.g., in the `mintShares` function).

It is important to note that in this protocol, `controlledBurnSharesFrom` is invoked in the `Controller` immediately following a transfer. As such, this will not pose an issue and the value emitted will be correct.

Consider documenting and explicitly stating this behavior, as other parts of the code that integrate this function could encounter future problems.

Update: *Partially resolved in pull request #22 at commit 58de4f7. As we mentioned, the issue pertained to the `controlledBurnSharesFrom` function. However, due to the manner in which the `Controller` interacted with it, no adverse effects occurred. With the function now corrected, it must be utilized properly in conjunction with the `Controller`. Specifically, in the `redeemUnderlying` function, it is crucial to conduct the `transfer()` after executing `burnShares()`. If not, the balances become inconsistent. This inconsistency arises because `shareToBalance()` is invoked before the burn, resulting in a certain value. Following this, the `transfer()` occurs, altering the total underlying in the protocol. Subsequently, when the burn function is executed, `shareToBalance` is recalculated, leading to a different value. Ensure that the transfer is executed post-burn.*

Notes & Additional Information

N-01 Constants Not Using `UPPER_CASE` Format

Throughout the codebase there are constants not using `UPPER_CASE` format. For instance:

- The `BaseERC20StorageLocation` constant declared on line 14 in `StorageBaseERC20.sol`
- The `ControllerStorageLocation` constant declared on line 26 in `StorageController.sol`
- The `ERC20SnapshotStorageLocation` constant declared on line 17 in `StorageERC20Snapshot.sol`

- The `ERC20SnapshotRebaseStorageLocation` constant declared on line 11 in `StorageERC20SnapshotRebase.sol`
- The `StakerStorageLocation` constant declared on line 15 in `StorageStaker.sol`

According to the Solidity Style Guide, constants should be named with all capital letters with underscores separating words. For better readability, consider following this convention.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *We followed the OpenZeppelin upgradable contracts library syntax. Not an issue.*

N-02 Todo Comments

During development, having well-described TODO/Fixme comments will make the process of tracking and solving them easier. Without this information, these comments might age and important information for the security of the system might be forgotten by the time it is released to production. These comments should be tracked in the project's issue backlog and resolved before the system's deployment.

Multiple instances of TODO/Fixme comments were found in the codebase. For instance:

- The `TODO` comment on line 321 in `Controller.sol`.
- The `TODO` comment on line 229 in `StakerM1.sol`.

Consider removing all instances of TODO/Fixme comments and instead tracking them in the issues backlog. Alternatively, consider linking each inline TODO/Fixme to the corresponding issues backlog entry.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *These `TODO` comments are for future upgraders to be aware of. Not an issue.*

N-03 Lack of Indexed Event Parameters

Throughout the codebase, several events do not have their parameters indexed. For instance:

- The `nonce` of the `WithdrawalQueued` event of `IStaker.sol`
- The `root` hash of the `WithdrawalCompleted` event of `IStaker.sol`

Consider indexing event parameters to improve the ability of off-chain services to search for and filter for specific events.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *For our needs, this is fine and more gas efficient.*

N-04 Non-Explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease the clarity of the code, and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Throughout the codebase, global imports are being used. For instance:

- The `StorageController.sol` import in `IController.sol`
- The `IDelegationTerms.sol` import in `IDelegationManager.sol`
- The `IERC20.sol` import in `IStrategy.sol`

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *Not an issue.*

N-05 Unused Errors

Throughout the codebase there are unused errors. For instance:

- The `MustHaveAdminRole` error in `BaseERC20.sol`
- The `ERC20SnapshotInvalidOperation` error in `ERC20Snapshot.sol`
- The `Controller_MustPullUnderlying` error in `IController.sol`

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused errors.

Update: *Resolved in pull request #23 at commit 0cadceb.*

N-06 Unused Imports

Throughout the codebase, there are multiple imports that are unused and could be removed. For instance:

- Import `StorageController.sol` in `IController.sol`
- Import `IDelegationManager.sol` has an unused alias `IDelegationManager` in `StakerM1.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *Not an issue.*

N-07 Unused Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function body for the purpose of being returned as the function's output. They are an alternative to explicit in-line `return` statements.

Throughout the codebase, there are multiple instances of unused named return variables. For instance:

- The `headId` return variable in the `getWithdrawalRequestQueue_Info` function of `StorageController.sol`.
- The `tailId` return variable in the `getWithdrawalRequestQueue_Info` function of `StorageController.sol`.
- The `count` return variable in the `getWithdrawalRequestQueue_Info` function of `StorageController.sol`.

Consider either using or removing any unused named return variables.

Update: *Resolved in pull request #24 at commit 7f22405.*

N-08 Unused Structs

Throughout the codebase, there are unused structs. For instance:

- The `WithdrawalRequest` struct in `IController.sol`

- The `SignatureWithExpiry` struct in `ISignatureUtils.sol`

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused structs.

Update: *Partially resolved in pull request #25 at commit 346f20d. The RestakeFi team stated:*

> *Fixed one. `ISignatureUtils` is external and we do not want to modify it.*

N-09 New `maxStakers` Value Not Being Checked

The `setMaxStakers` function allows the `StrategyManager` to modify the maximum number of Stakers. However, it lacks checks to ensure that the new maximum amount specified as a parameter is equal to or greater than the current number of Stakers.

While this does not inherently pose a security risk, it could lead to confusion. Consider incorporating a check ensuring that the new maximum Staker amount is not set lower than the existing number of Stakers.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *Prefer not to modify. Not an issue for us.*

N-10 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice proves beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. Additionally, if the contract incorporates third-party libraries and a bug surfaces in these, it becomes easier for the maintainers of those libraries to make contact with the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are contracts that do not have a security contact:

- The `BaseERC20` contract
- The `Controller` contract
- The `ERC20Snapshot` contract
- The `ERC20SnapshotRebase` contract

- The `EigenSharePricer` contract
- The `IController` contract
- The `IDelegationManager` contract
- The `IERC20SnapshotRebase` contract
- The `ISignatureUtils` contract
- The `IStaker` contract
- The `IStrategy` contract
- The `StakerM1` contract
- The `StorageBaseERC20` contract
- The `StorageController` contract
- The `StorageERC20Snapshot` contract
- The `StorageERC20SnapshotRebase` contract
- The `StorageStaker` contract

Consider adding a NatSpec comment containing a security contact on top of the contracts' definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Acknowledged, not resolved.*

N-11 Inaccurate Revert Messages

The `processWithdrawals`, `staker_pullUnderlying` and `staker_completeWithdrawal` functions in the Controller contract will revert either when the current operation being triggered is paused, or when the caller does not possess the `STRATEGY_MANAGER_ROLE`, among other reasons.

However, if a caller lacking the `STRATEGY_MANAGER_ROLE` attempts to invoke these functions, they will erroneously revert with a pause-related error message, instead of the more appropriate `Controller_Unauthorized` message. To improve clarity and visibility, consider splitting these `if` statements into two separate checks so that the functions can revert with the correct error message when a user without authorization attempts to call them.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *The pause flags allow those functions to be paused only for the user. The strategy manager is still able to call them. When not paused, anyone can call those functions. So, we consider the error to be fit for the use case.*

N-12 Refactor Opportunities

Throughout the codebase, there are instances where the code can be refactored. In particular:

- Within the `BaseERC20` contract, these getters could be reused instead of recreating the same logic in the `controlledTransferFrom`, `controlledBurnFrom`, `mint`, and `pause` functions.
- In the `EigenSharePricer` contract, an external call to query the `totalShares` value is unnecessary as the same value is stored in the `totalShares` local variable.

Consider applying these refactoring opportunities in the codebase to reduce its size and improve readability.

Update: *Partially resolved in pull request #26 at commit b184d96. The RestakeFi team stated:*

> *The second bullet point is resolved. The first bullet point is not an issue for us.*

N-13 Incomplete Parameters in Emitted Event

The `MaxStakersSet` event should emit both the old maximum value as well as the new one. In general, when modifying a state variable in the system, consider emitting both its old and new value to notify off-chain clients monitoring the contracts' activity.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *Not an issue for us.*

N-14 Error In Function Name

Consider changing the `requestWithdraw` function's name to `requestWithdrawal`.

Update: *Acknowledged, not resolved. The RestakeFi team stated:*

> *Not an issue for us.*

Conclusion

While evaluating the protocol, certain aspects suggest it may benefit from further refinement before being considered fully production-ready.

Design

- In the protocol token, the method of inverting balances between shares and underlying assets introduces complexity, potentially leading to errors and posing challenges for developers considering adoption or integration. Additionally, the pattern of implementing logic that is later overridden by new developments, while leaving the original elements unused, raises concerns about potentially increasing the attack surface.
- The overall design, spanning from deposit to asset redemption, could benefit from reconsideration. The use of double-linked lists might be excessive, given that the structure functions more as a FIFO list and is not iterated in both directions. Moreover, permitting unrestricted request submissions of any amount by anyone opens the system to potential spam attacks, as highlighted earlier in this report. While some mitigations for reducing the attack surface have been suggested, this design may still harbor other vulnerabilities not identified during this audit.

The current design of the protocol suggests it is still evolving, with room for further enhancement and refinement towards becoming a polished, secure, and finalized product.

Testing

The project currently utilizes Hardhat tests to assess the functionality of its various components. Although the test coverage is satisfactory and covers most of the lines, this is not an indication that the code is protected against all scenarios nor potential edge cases, as there is a notable reliance on hardcoded values for testing different scenarios.

An important instance was a critical severity issue that could have been easily detected where it stemmed from an incomplete assessment of the controller's actions' effects. While these actions were tested, their impacts were not thoroughly evaluated. Implementing fuzz testing for unexpected values and correctly checking their effects is advisable to enhance confidence levels. This can be achieved by transitioning the existing tests to Foundry, utilizing its built-in fuzz testing mechanism to encompass a broader range of cases.

Additionally, while most tests primarily focus on individual unit testing, the codebase could greatly benefit from more comprehensive integration tests. Such tests should simulate common user interactions and potential attacker behaviors, thoroughly examining the system's integrated functionalities. These tests could involve more than one user interacting with different parts of the protocol.

In summary, adopting a more rigorous and comprehensive testing approach, which includes both extensive unit and integration tests, would significantly bolster the system's overall robustness and reliability.

Update: *During the fix review, it was noticed that likely due to the limited time frame assigned to understand the issues, think of thorough solutions, their possible limitations, caveats, and potential integration problems, the presented fixes were not able to tackle the root of the problems and study their possible impact throughout the codebase. Due to this, integration issues between the fixes and the codebase have been found for higher severity issues. Due to the limited fix review time frame, there might be more possible attack vectors that were not explored. As security generally relies on the time allocated to it when developing the code, it is recommended to continue exploring deeper solutions to mitigate the reported issues, improving the testing suite to catch expected and unexpected/edge scenarios, and developing a threat model for the protocol. Furthermore, even the less severe issues reported could be translated into tools that attackers may use when the opportunity presents itself, so it is advised to fix all issues instead of accepting the risk. The calculations for the rebasing functionality of the token currently suffer from manipulation scenarios that break premises such as the precision or final value of the shares. For these operations, it is recommended to develop invariant tests that could catch further situations that could put the protocol in danger, either due to the values or in an unexpected order of operations. Moreover, it would be wise to formally verify all possible math operations during such calculations. Finally, some code was introduced which has not been audited. It is recommended to add it in a future audit to reduce the possibility of introducing new issues.*