Fractal Network

August Vault

Security Assessment Report

Version: 2.1

August, 2024

Contents

Introduction

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

Disclaimer

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

Document Structure

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

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

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

Overview

The `LendingPool` contract is an ERC4626-compliant token vault designed to increase the utilisation of a liquid restaking token (LRT) through lending and borrowing activities. Users with LRT can become lenders by depositing their tokens into the pool. The pool operator then lends the collected tokens or funds to borrowers through loan contracts. Lenders can withdraw from the scheme at any time, and the pool operator reserves the right to call the loans whenever necessary to ensure the required funds are available for returning to the lenders.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Fractal Protocol repository.

The scope of this time-boxed review was strictly limited to files at commit 4923890. The fixes of the identified issues were assessed at commit d510a02.

The list of assessed contracts is as follows:

- `LendingPool`
- `BaseUpgradeableERC20`
- `BaseUpgradeableERC4626`
- `TimelockedERC4626`
- `OwnableLiquidityPool`
- `AbstractLender`
- `HookableLender`
- `BaseLendingPool`
- `BaseOwnable`
- `BaseReentrancyGuard`
- `DateUtils`
- `IPermissionlessLoansDeployer`
- `ILenderHook`
- `IPeerToPeerOpenTermLoan`

Note: third party libraries and dependencies, such as OpenZeppelin and Murky, were excluded from the scope of this assessment.

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

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

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

Output for these automated tools is available upon request.

Coverage Limitations

Due to a time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 13 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 3 issues.
- Low: 2 issues.
- Informational: 7 issues.

Detailed Findings

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

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

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

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

Summary of Findings

ID	Description	Severity	Status
AUG-01	Two `nonReentrancy` Modifiers Prevent `liquidate()` Execution	High	Resolved
AUG-02	Restricted Token Redeem Period When `lagDuration == 0`	Medium	Resolved
AUG-03	Inflation Attack On Empty Vault Can `DoS` The Vault	Medium	Resolved
AUG-04	Recent Solidity Versions May Not Be Supported By Layer 2 Systems	Medium	Resolved
AUG-05	Index Swapping May Prevent Subsequent Calls To `claim()`	Low	Resolved
AUG-06	Excessive Redeem Requests May Cause `processAllClaimsByDate()` to Revert	Low	Resolved
AUG-07	Potentially Inefficient Search On `getReceiverIndex()`	Informational	Resolved
AUG-08	Potential Double Funding Of Loan Contracts	Informational	Resolved
AUG-09	Ownership Mechanism Lacks Additional Safeguards	Informational	Closed
AUG-10	Deviation From Common Interface Naming Convention	Informational	Resolved
AUG-11	Variable `_maxSupply` Is Used For Two Different Purposes	Informational	Closed
AUG-12	Rogue Owner May Siphon Assets Through `emergencyWithdraw()`	Informational	Closed
AUG-13	Miscellaneous General Comments	Informational	Resolved

AUG-01	Two `nonReentrancy` Modifiers Prevent `liquidate()` Execution		
Asset	`pools/base/AbstractLender.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Two `nonReentrant` modifiers are executed in a single call, causing `ReentrancyGuard` to revert during `liquidate()`.

When the function `AbstractLender.liquidate()` is called, the following call sequence occurs.

1. `AbstractLender.liquidate()` is called on the lender contract. The function has a `nonReentrant` modifier. At this point, `_reentrancyStatus` is set to `_REENTRANCY_ENTERED`.
2. `IPeerToPeerOpenTermLoan(loanAddr).liquidate()` is called on the respective loan contract. This code may execute `InitializableOpenTermLoan.liquidate()`.
3. `IHookableLender(lender).notifyLoanMatured()` is called on the lender contract. The function has a `nonReentrant` modifier as described in `HookableLender.sol`.

Since the `_reentrancyStatus` of the lender contract was already set to `_REENTRANCY_ENTERED` in step (1), the call in step (3) would cause a revert on `BaseReentrancyGuard._nonReentrantBefore()`. The result is, `AbstractLender.liquidate()` will revert.

The impact is rated as medium severity as being unable to call `liquidate()` prevents losses being accounted for at the pool level.

Recommendations

Consider removing one of the `nonReentrant` modifiers either on `AbstractLender` or `HookableLender` contract.

Resolution

The issue was resolved on commit 2196d53. The `nonReentrant` modifier on `AbstractLender.liquidate()` was removed.

AUG-02	Restricted Token Redeem Period When `lagDuration == 0`		
Asset	`pool/base/TimelockedERC4626.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

If `lagDuration == 0`, users may have only a limited period, potentially just 1 hour in a day, to claim their token.

When `lagDuration == 0`, the pool is not time-locked. Therefore, users are supposed to redeem their tokens instantly. However, this is not the case, as the redeem phase still depends on the `liquidationHour`. Indeed, when `lagDuration == 0`, the function `requestRedeem()` calls the internal function `_claim()` in the following code from lines [**130-133**]:

```
if (lagDuration == 0) {
    claimableEpoch = block.timestamp;
    _claim(year, month, day, 0, receiverAddr);
}
```

If the call happens before the `liquidationHour`, the call will revert because of the `require` statement on line [**428**]:

```
require(block.timestamp + _TIMESTAMP_MANIPULATION_WINDOW >= DateUtils.timestampFromDateTime(year, month, day, liquidationHour, 0,
    ↪ 0), "Too early");
```

Hence, the current timestamp is checked with `year`, `month`, `day` and the `liquidationHour`. For example, when `liquidationHour = 23`, users would have only 1 hour a day, just between *11:00 PM UTC and 11:59 PM UTC*, to redeem their tokens.

Recommendations

Update the `liquidationHour` to `0`, when updating the `lagDuration` to `0` in the function `updateTimelockDuration()`. Also, consider reverting if there is a call to update the `liquidationHour` to a value other than `0` when `lagDuration == 0` in the function `updateProcessingHour()`.

Resolution

The development team has fixed this issue in commit d510a02, by adding the following the require statement when the `lagDuration > 0`.

```
require(block.timestamp + _TIMESTAMP_MANIPULATION_WINDOW >= DateUtils.timestampFromDateTime(year, month, day, liquidationHour, 0,
    ↪ 0), "Too early");
```

AUG-03	Inflation Attack On Empty Vault Can DoS The Vault		
Asset	`pools/base/BaseUpgradeableERC4626.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The first depositor may execute an inflation attack to prevent other users from depositing.

A malicious first depositor can inflate the rate between the shares and assets. This can be done by first depositing the lowest possible amount of supported assets to the vault, then transferring a large amount of assets to the vault contract directly without calling `deposit()` or `mint()`. The asset transfer will artificially inflate the share price for future depositors. If their deposited amount is less than a specific value, legitimate users cannot deposit due to the check statement on line [**100**] because their shares will be `0` due to the share price inflation.

Consider the following attack scenario:

1. First we assume that the vault is freshly generated.
2. The attacker deposits 1 asset by calling `deposit(1)`. Thus, `totalAssets()==1`, `totalSupply()==1`.
3. The attacker inflates the rate by transferring the underlying asset directly to the vault. Let us assume they transfer `10_000e6 - 1`. Now we have `totalAssets() == 10_000e6` and `totalSupply() == 1`.
4. At this point, legitimate users cannot deposit an amount less than `10_000e6`. This is because if the deposited amount is less than `10_000e6`, the expected shares will be zero, since `shares = totalSupply * depositedAmount / totalAssets`, and therefore we have `shares = 1 * depositedAmount / 10_000e6` which yields zero shares.

Recommendations

The simplest countermeasure to inflation attacks is ensuring that the vault is never empty. This can be achieved by depositing into the vault in the deployment/initialisation script.

Resolution

The development team has confirmed that they are going to manually perform the first deposit into the vault.

AUG-04	Recent Solidity Versions May Not Be Supported By Layer 2 Systems		
Asset	`src/*.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Solidity versions starting from `0.8.25` use the `MCOPY` op-code by default. The op-code `MCOPY` will cause a revert if it is called on chains that are not upgraded to the EVM version *Cancun*. The Solidity operations that include the `MCOPY` by default:

- The helper function `abi.encode()`;
- Functions which return byte array;
- Functions which return string types.

The testing team has compiled the contract and discovered occurrences of `MCOPY` in the bytecode.

The impact of deploying the contract on chains that do not upgrade their EVM to *Cancun* is high. This is because the funds can be stuck. `MCOPY` will not be used when depositing funds allowing funds to enter the protocol. However, when trying to redeem, the transaction will revert since the function `_registerRedeemRequest()` contains an `abi.encode()` instruction which uses `MCOPY` by default.

Recommendations

Consider setting the Solidity compile flag `--evm-version` such that it is an earlier EVM version. `paris` is a good choice for the EVM version as it does not contain either `PUSH0` or `MCOPY` opcodes.

Alternatively, using a compiler version which is strictly less `< 0.8.25`.

Note that `shanghai` (the default in Solidity from `0.8.20` to `0.8.24`) introduces the opcode `PUSH0` which may or may not be supported by other chains.

Furthermore, validate the EVM version of each chain before compiling contracts and deploying to that chain.

Resolution

The development team are resolving the issue by setting the EVM version to Paris during compilation.

AUG-05	Index Swapping May Prevent Subsequent Calls To `claim()`		
Asset	`pools/base/TimelockedERC4626.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The internal function `_deleteReceiver()` swaps the index of a receiver address with the last item, if the index is not the last item.

This algorithm is commonly used in Solidity as a cost-effective solution for removing an array item. However, since `receiverIndex` is used as one of the inputs in the `claim()` function, the index swapping could cause a revert if there are subsequent calls to `claim()`.

Consider the following mock case. Let us assume there are ten users who wish to withdraw at the same time.

1. First, they will query their respective `receiverIndex` through the `getReceiverIndex()` function.
2. Using this information, each user calls `claim()`. Ten transactions are created almost at the same time.
3. We assume that the first user's transaction is successful. At this point, the last item is swapped to be the first item in the array.
4. If the last user's transaction is executed, this transaction reverts with an `Invalid receiver index` message.

Recommendations

The `receiverIndex` information can be cheaply computed onchain if the basic array is replaced with alternative solutions such as OpenZeppelin's EnumerableSet.

Resolution

The development team has resolved this issue by extracting the value of `receiverIndex` on-chain in the `_claim()` function, through the `receiverAddr` by using a new mapping `_receiverIndexes`.

AUG-06	Excessive Redeem Requests May Cause `processAllClaimsByDate()` to Revert		
Asset	`pools/base/TimelockedERC4626.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Users can submit a large number of redeem requests on the same day. This could prevent a successful call to `processAllClaimsByDate()` if the call breaches the block gas limit.

The function `processAllClaimsByDate()` loops through all redeem requests for a single day. If the number of requests is too large, the result is excessive gas consumption potentially larger than the block gas limit.

Gas limit tests indicate that the block gas limit will be breached when there are roughly 1,000 redeem requests.

Consequently, users may need to call the `claim()` function for individual requests.

Recommendations

Consider adding a limit as an input to the `processAllClaimsByDate()` function so that requests can be processed in multiple batches when needed. Also, consider adding a minimum withdrawal to prevent dust amounts from being withdrawn.

Resolution

The first recommendation has been implemented in commit d510a02. The function `processAllClaimsByDate()` now has an argument `maxLimit` which represent the number of request to process for a `dailyCluster`.

AUG-07	Potentially Inefficient Search On `getReceiverIndex()`
Asset	`pools/base/TimelockedERC4626.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The code on lines [**336-338**] uses a basic loop to find the index in the array that contains an address. The gas cost of the query increases proportionally with the number of requests. The case of significant gas cost may arise deliberately or maliciously, if there are numerous requests with different `receiverAddr` values received by the contract in one day.

It is also worth noting that if there are multiple requests with the same `receiverAddr`, this query only returns the first request in the array.

Recommendations

To improve the efficiency of the search mechanism, consider implementing OpenZeppelin's EnumerableSet to store addresses.

Resolution

The function `getReceiverIndex()` has been removed in commit d510a02 and it is replaced by mapping `_receiverIndexes` which stores the index of each unique receiver per cluster.

AUG-08	Potential Double Funding Of Loan Contracts
Asset	`pools/base/BaseLendingPool.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `fundLoan()` function in `BaseLendingPool` does not perform a pre-check on the loan state to ensure that it requires funding.

```
function fundLoan(address loanAddr) external override onlyIfInitialized nonReentrant ifConfigured onlyLoansOperator {
// ... existing code ...

    IPeerToPeerOpenTermLoan(loanAddr).fundLoan();

// Post checks
    require(IPeerToPeerOpenTermLoan(loanAddr).loanState() == LOAN_ACTIVE, "Funding check failed");
// ... other checks ...
}
```

While there is a post-funding check to ensure the loan is active, there is no pre-funding verification that the loan requires funding.

The issue is raised as informational severity, as the check is performed in `InitializableOpenTermLoan`. If other implementations of `IPeerToPeerOpenTermLoan` do not implement the check, double funding could be possible.

Recommendations

Consider adding a pre-funding check in `fundLoan()` to ensure the loan state is `LOAN_FUNDING_REQUIRED` before proceeding with the funding operation.

Note that this check would increase gas costs in making an additional call to the `loanAddr`.

Resolution

The recommended check has been implemented in commit d510a02.

AUG-09	Ownership Mechanism Lacks Additional Safeguards
Asset	`core/BaseOwnable.sol & pools/LendingPool.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `BaseOwnable` and `LendingPool` contracts implement a basic ownership mechanism. While functional, it lacks safeguards against accidental transfers to invalid addresses.

The current transfer of ownership pattern calls the function `transferOwnership(address newOwner)` which instantly changes the owner to the `newOwner`. This allows the current owner of the contracts to set an arbitrary address.

If the address is entered incorrectly, the owner role of the contract is lost forever. Thus, a user would not be able to pass the `onlyOwner` modifier.

Recommendations

This scenario is typically mitigated by implementing a two-step transfer pattern, whereby a new owner address is selected, then the selected address must call an `acceptOwnership()` before the owner is changed. This ensures the new owner address is accessible.

Consider adopting OpenZeppelin's Ownable2Step pattern or implementing a similar two-step ownership transfer process.

Resolution

The development has decided not to fix the issue. They mentioned that the owner will be protected by a multi-sig account.

AUG-10	Deviation From Common Interface Naming Convention
Asset	`loans/interfaces/ILenderHook.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The interface `ILenderHook` deviates from common naming conventions. Typically, interfaces are named using the format `I` + contract name.

The contract implementing `ILenderHook` is named `HookableLender`.

Recommendations

Consider renaming the interface to `IHookableLender` for improved clarity and adherence to standard naming practices.

Resolution

The interface was renamed to `IHookableLender` as per the recommendation in commit d510a02.

AUG-11	Variable `_maxSupply` Is Used For Two Different Purposes
Asset	`pools/base/BaseUpgradeableERC4626.sol and pools/base/BaseUpgradeableERC20.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The variable `_maxSupply` is used as the return value of the function `maxMint()` which, according to the NatSpec comment of this function, is specified as *"the maximum amount of the Vault shares that can be minted for the receiver, through a mint call"*. However, this variable is also used as the maximum value of `totalSupply` in the function `BaseUpgradeableERC20._canMint()`.

Recommendations

To avoid confusion, use another variable as the return value of the function `maxMint()` or rename the function `maxMint()` to `maxSupply()`.

Resolution

The development team has decided to mark this issue as a *Won't fix*. They confirmed that the max supply is the max mint as well.

AUG-12	Rogue Owner May Siphon Assets Through `emergencyWithdraw()`
Asset	`pools/base/OwnableLiquidityPool.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `emergencyWithdrawal()` function lacks preconditions or restrictions, allowing the owner to withdraw any tokens at will.

There is a risk to the integrity of the pool and the security of depositors' funds. Specific concerns for depositors include:

- No defined conditions that constitute an "emergency," giving the owner discretion to use this function at any time without justification.
- A malicious or compromised owner exploiting this function to perform an exit scam, instantly draining all assets from the pool.
- Undermining the trustless nature of the DeFi protocol, as depositors must rely entirely on the owner's integrity.

The contrary view is that an `emergencyWithdraw()` potentially allows benevolent owner to extract funds to a safe address in the event of a protocol compromise.

Recommendations

Ensure the `owner` is a multi-signature wallet or the relevant DAO.

Furthermore, consider the trade-offs between safely extracting user funds in the case of an emergency and the risk of a compromised owner address stealing funds stored in the protocol.

Resolution

The development team has decided not to fix this issue as they will use a multi-sig account to call the function `emergencyWithdraw()`.

AUG-13	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **Redundant Code Adds Minimal Impact To The Contract**

   Related Asset(s): pools/LendingPool.sol

   The following code snippet indicates an effort to initialise the value of `_reentrancyStatus`.

   ```
   _reentrancyStatus = _REENTRANCY_NOT_ENTERED;
   ```

   The variable `_reentrancyStatus()` is utilised to store a flag to identify whether reentrancy has occurred or not. The utilisation of this variable is done mainly on `BaseReentrancyGuard` contract.

   A `uint256` Solidity variable would be assigned to a default value of zero and therefore, the initialisation code above is not necessary to make modifier `nonReentrant` of `BaseReentrancyGuard` contract works.

   The code on line [30] of `LendingPool` contract can be safely removed.

2. **Inaccurate Data In Event `WithdrawalRequested`**

   Related Asset(s): pools/base/TimelockedERC4626.sol

   The event emittance on line [405] uses `assetsAmount` instead of `effectiveAssetsAmount`. This does not conform to the NatSpec specification for event `Requested` on line [51] as follows:

   ```
   @param assets The amount of underlying assets to transfer.
   ...
   event WithdrawalRequested (address ownerAddr, address receiverAddr, uint256 shares,
                                           uint256 assets, uint256 fee, uint256 year, uint256 month, uint256 day);
   ```

   Consider replacing `assetsAmount` with `effectiveAssetsAmount`.

3. **Duplicate Code Present In Notifying Loans**

   Related Asset(s): pools/base/HookableLender.sol

   The functions `notifyLoanMatured()`, `notifyLoanClosed()` and `notifyPrincipalRepayment()` in the `HookableLender` contract are nearly identical. While these functions perform the same operations, they serve distinct semantic purposes in the contract's logic. However, code duplication should be minimised as much as possible.

   To reduce code duplication while maintaining semantic clarity, consider implementing an internal function containing the shared logic.

4. **Out Of Place Function**

   Related Asset(s): pools/base/BaseLendingPool.sol

   The `collectFees()` function in `BaseLendingPool` appears to be out of place:

```
function collectFees() external onlyIfInitialized nonReentrant ifConfigured onlyOwner {
    require(feesCollector != address(0), "Fee collector not set");
    require(totalCollectableFees > 0, "No fees to collect");
    _collectFees();
}
```

This function calls `_collectFees()`, which is likely inherited from `TimelockedERC4626`. These fees are typically associated with user redemptions of deposits, rather than being directly related to loan operations. Its presence in a contract focused on loan deployment and management may lead to confusion about the source and nature of these fees.

Consider moving this function to a more appropriate contract that deals with user deposits and withdrawals, or clearly document its purpose and fee source in the contract.

5. **Redundant Code And Unnecessary Conditional Structure**

   Related Asset(s): pools/LendingPool.sol

   The current implementation of `updateTimelockDuration` contains redundant code and a potentially unnecessary conditional structure:

```
if (newDuration > lagDuration) {
    lagDuration = newDuration;
} else {
    require(globalLiabilityShares == 0, "Process claims first");
    lagDuration = newDuration;
}
```

   Refactor the code to remove duplication and improve readability:

```
if (newDuration < lagDuration) {
    require(globalLiabilityShares == 0, "Process claims first");
}
lagDuration = newDuration;
```

6. **Contract Contains Commented-Out Code**

   Related Asset(s): pools/LendingPool.sol

   The `updateTimelockDuration` function in the `LendingPool` contract contains commented-out code:

```
//require(newDuration >= 2 hours, "Timelock too short");
```

   Remove the commented-out code. If this check is no longer needed, it should be deleted entirely. If it might be required in the future, document the rationale in a comment or move it to development notes outside the contract.

7. **Unnecessary `onlyIfInitialized` Modifier**

   Related Asset(s): src/*.sol

   The use of the `onlyIfInitialized` modifier in functions that have the modifier `ifConfig` is unnecessary and leads to more gas consumption. This is because the function `configurePool()` has the modifier `onlyIfInitialized`.

   Remove the modifier `onlyIfInitialized` from the functions that have `ifConfig`.

Recommendations

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

Resolution

The development team's responses to the raised issues above are as follows.

1. The issue was resolved as suggested. The initialisation of the variable `_reentrancyStatus` is removed.
2. The issue was acknowledged by the development team as the effective amount can be calculated offchain `assetsAmount - applicableFee`.
3. The issue was acknowledged by the development team as the internal function will increase the contract's code size.
4. The issue was resolved by removing the function `_collectFees()` from the contract `TimelockedERC4626`. The external function `collectFees` is updated accordingly.
5. The code was refactored as suggested.
6. The code was removed as suggested.
7. The issue was resolved as suggested by removing the modifier `onlyIfInitialized`.

Appendix A Test Suite

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

```
Ran 1 test for test/tests-local/pools/base/OwnableLiquidityPool.t.sol:OwnableLiquidityPoolTestSigp
[PASS] test_sigp_emergencyWithdraw_USDC() (gas: 14975324)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 12.70ms (2.49ms CPU time)

Ran 4 tests for test/tests-local/pools/base/BaseUpgradeableERC4626.t.sol:BaseUpgradeableERC4626TestSigp
[PASS] test_sigp_deposit_USDC(uint256) (runs: 1002, μ: 14806114, ~: 14772111)
[PASS] test_sigp_deposit_USDC_inflationAttack() (gas: 15023930)
[PASS] test_sigp_mint_USDC(uint256) (runs: 1002, μ: 14807385, ~: 14774765)
[PASS] test_sigp_mint_USDC_donationAttack() (gas: 15150679)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.87s (3.63s CPU time)

Ran 2 tests for test/tests-local/pools/base/BaseLendingPool.t.sol:BaseLendingPoolTestSigp
[PASS] test_sigp_deployLoan_fundLoan_USDC_WBTC() (gas: 19913658)
[PASS] test_sigp_deployLoan_fundLoan_twice_USDC_WBTC() (gas: 20112060)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 3.24s (9.30ms CPU time)

Ran 8 tests for test/tests-local/pools/LendingPool.t.sol:LendingPoolTestSigp
[PASS] test_sigp_configurePool(uint256,uint256,uint256,uint256,address,address,address,uint8) (runs: 1002, μ: 14626428, ~:
     ↪ 14558876)
[PASS] test_sigp_pauseDepositsAndWithdrawals(bool,bool) (runs: 1002, μ: 14732481, ~: 14732505)
[PASS] test_sigp_transferOwnership(address) (runs: 1002, μ: 14575925, ~: 14575925)
[PASS] test_sigp_updateFeesCollector(address) (runs: 1002, μ: 14754282, ~: 14754282)
[PASS] test_sigp_updateIssuanceLimits(uint256,uint256,uint256) (runs: 1002, μ: 14736716, ~: 14737018)
[PASS] test_sigp_updateProcessingHour(uint8,bool) (runs: 1002, μ: 14642503, ~: 14709447)
[PASS] test_sigp_updateTimelockDuration(uint256) (runs: 1002, μ: 14729662, ~: 14729087)
[PASS] test_sigp_updateWithdrawalFee(uint256) (runs: 1002, μ: 14732862, ~: 14726989)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 3.66s (12.48s CPU time)

Ran 5 tests for test/tests-local/pools/base/AbstractLender.t.sol:AbstractLenderTestSigp
[PASS] test_sigp_callLoan_USDC_WBTC() (gas: 20119322)
[PASS] test_sigp_liquidate_USDC_WBTC() (gas: 20136742)
[PASS] test_sigp_returnCollateral_USDC_WBTC() (gas: 20135092)
[PASS] test_sigp_seizeCollateral_USDC_WBTC() (gas: 20128056)
[PASS] test_sigp_seizeCollateral_callLoan_USD_WBTC() (gas: 20133381)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 10.39s (25.99ms CPU time)

Ran 12 tests for test/tests-local/pools/base/TimelockedERC4626.t.sol:TimelockedERC4626TestSigp
[PASS] test_sigp_processAllClaimsByDate(uint256) (runs: 1002, μ: 16032280, ~: 16029174)
[PASS] test_sigp_processAllClaimsByDate_multi_oog(uint256) (runs: 1002, μ: 18579696, ~: 18550860)
[PASS] test_sigp_redeem(uint256,uint256) (runs: 1002, μ: 14964413, ~: 14964136)
[PASS] test_sigp_requestRedeem_claim_with_fees(uint256,uint256) (runs: 1002, μ: 15136160, ~: 15136710)
[PASS] test_sigp_requestRedeem_claim_zero_fees(uint256,uint256) (runs: 1002, μ: 15083092, ~: 15083898)
[PASS] test_sigp_requestRedeem_claim_zero_fees_changing_index() (gas: 17322746)
[PASS] test_sigp_requestRedeem_getReceiverIndex_duplicateReceiver(uint256) (runs: 1002, μ: 15339612, ~: 15347793)
[PASS] test_sigp_requestRedeem_getReceiverIndex_multi(uint256) (runs: 1002, μ: 20253518, ~: 20436968)
[PASS] test_sigp_requestRedeem_zero_fees_zero_lagDuration(uint256,uint256) (runs: 1002, μ: 15179266, ~: 15179412)
[PASS] test_sigp_requestRedeem_zero_fees_zero_lagDuration_claimableEpoch(uint256,uint256) (runs: 1002, μ: 15040359, ~: 15040926)
[PASS] test_sigp_requestRedeem_zero_fees_zero_liquidationHour(uint256,uint256,uint256) (runs: 1002, μ: 15185739, ~: 15185717)
[PASS] test_sigp_withdraw(uint256,uint256) (runs: 1002, μ: 14964420, ~: 14964299)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 10.77s (48.76s CPU time)

Ran 6 test suites in 10.78s (29.94s CPU time): 32 tests passed, 0 failed, 0 skipped (32 total tests)
```

Appendix B Vulnerability Severity Classification

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

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

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

References

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

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