PUBLIC

Code Assessment

of the Core Vault Smart Contracts

Jan 21, 2025

Produced for

by

Contents

1 Executive Summary

Dear August,

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

The assessed contracts are the base layer for an investment protocol. The main contract is an upgradable ERC4626 vault with a custom delayed withdrawal feature.

The most critical subjects covered in our audit are correct asset accounting, fee and share handling, functional correctness, and standard compliance. We identified issues in the current vault implementation regarding the share price calculations and fee accounting (see Incorrect price per share and Withdrawal Fees Are Counted Towards the Vault Assets). All issues where addressed.

Less severe issues were caused by custom implementation instead of using libraries. Therefore, we highly recommend using audited libraries whenever possible. Some major findings are acknowledged or the risk is accepted and the mitigation is based on an off-chain solution (e.g., Blacklisted address can redeem shares) run by August to ensure the correct functioning of the system. These mitigations are out of scope for this audit. We recommend August to ensure that the infrastructure is secure and reliable and to define clear specifications and tests for it to ensure its correct functioning.

The documentation provided is sparse and many assumptions are present regarding different components, actions and roles. We highly recommend ramping up the documentation, writing clear specifications and assumptions for the system, and extending the test suite but the communication with the team was always good and prompt.

In summary, we find that the codebase provides a good level of security. Yet, it is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	3
• Code Corrected	2
• Risk Accepted	1
Medium-Severity Findings	6
• Code Corrected	6
Low-Severity Findings	8
• Code Corrected	2
• Specification Changed	1
• Risk Accepted	1
• Acknowledged	4

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the source code files inside the Core Vault repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	18 Dec 2024	8ce6b80b26be065d633e5b0cf4e6e9924566c3d2	Initial Version
2	9 Jan 2025	945837ba4e6e23a2c783b6f138b4cb72961da222	After Intermediate Report
3	13 Jan 2025	e3d4f8eca4b913a6f3b7277d98c2ca8272ecfbfe	Final version

For the solidity smart contracts, the compiler version `^0.8.19` and `>= 0.8.26` were chosen.

2.1.1 Included in scope

- src/core/AddressWhitelist.sol
- src/core/BaseOwnable.sol
- src/core/BaseReentrancyGuard.sol
- src/core/LightweightOwnable.sol
- src/core/TimelockedCall.sol
- src/pools/base/BaseUpgradeableERC20v2.sol
- src/pools/base/BaseUpgradeableERC4626v2.sol
- src/pools/base/TimelockedClaimOnlyERC4626.sol
- src/tokenized/IAllocable.sol
- src/tokenized/TokenizedAccount.sol
- src/tokenized/BaseTokenizedAccount.sol

2.1.2 Excluded from scope

Any contracts inside the repository that are not mentioned in Scope are not part of this assessment. Tests and deployment scripts are excluded from the scope as well as third party libraries.

2.2 System Overview

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

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

August implements a time locked ERC4626 where withdrawals are delayed by a set time period. Users can deposit funds into a time locked vault where subaccount managers can invest the funds into various other on-chain protocols on different chains.

TimelockedCall

The `TimelockedCall` contract is used to delay certain functionalities by a predefined `timeLockDuration`. Additionally, it restricts the calling of functions by requiring that function calls be registered and scheduled in the `queue` before execution.

Whitelisted addresses can initialize a scheduler by calling `initScheduler()`. A whitelisted scheduler is authorized to call `schedule`, which sets a `TimelockedCallInfo` struct in the `queue` mapping. This struct includes the `targetEpoch`, `createdBy`, and `co`nsumerAddress`. By calling `consume()` or `consumeOwnership()`, the scheduled call can be executed by the `consumerAddress` once the current block time exceeds the `targetEpoch` specified in the `TimelockedCallInfo`.

When deployed, the deployer provides a non-zero address that becomes the owner of the contract. The owner has the authority to enable and disable addresses to become whitelisted through the `_whitelistedAddresses` mapping. The owner can also transfer ownership directly to any address using `transferOwnership`.

The `queue` is shared among all schedulers, allowing multiple schedulers to manage and execute different scheduled calls concurrently.

It is assumed that all hashes are safely constructed and that there are no collisions.

BaseUpgradeableERC20v2

The `BaseUpgradeableERC20v2` contract is an abstract contract that implements the ERC20 interface. The contract inherits from `Initializable` and `BaseReentrancyGuard`. Additionally, the `_maxSupply` can be configured with `_setMaxSupply()`. The supply is increased when new tokens are minted with `_mintErc20()` and decreased when tokens are burned with `_burnErc20()`. `_mintErc20()` verifies through `_canMint()` that the maximum supply is not exceeded.

The contract also implements blacklist functionality through the `isBlacklisted` mapping.

BaseUpgradeableERC4626v2

The `BaseUpgradeableERC4626v2` contract is an abstract contract that partially implements the ERC4626 interface. The contract inherits from `BaseUpgradeableERC20v2`. Vault deposits and withdrawals can be paused through the `depositPaused` and `withdrawalPaused` flags, which can be set using `_setPause()`. Furthermore, maximum deposit and withdrawal amounts can be configured through `maxDepositAmount` and `maxWithdrawalAmount` by calling `_updateIssuanceLimits()`. The maximum share supply can also be modified with `_updateIssuanceLimits()`.

`deposit()` and `mint()` are restricted as they cannot be called by blacklisted addresses, and the receiver address of the vault shares is also verified to not be blacklisted.

A `withdrawalFee` is defined, which is used in `previewRedeem()` to calculate the amount of underlying assets that will be redeemed to the user for a given amount of shares.

`redeem()` and `withdraw()` from the ERC4626 interface are not implemented.

TimelockedClaimOnlyERC4626

`TimelockedClaimOnlyERC4626` is an abstract contract that inherits from `BaseUpgradeableERC4626v2` and implements the time-locked withdrawal logic for the vault.

The contract implements `withdraw()` and `redeem()` to always revert according to the ERC4626 standard.

`requestRedeem()` allows users to create a redeem request. A redeem request is delayed by a set time period defined by `lagDuration` and a 5-minute manipulation window. The redemption timestamp is

then converted to a (year, month, day) tuple. Therefore, the exact redemption hour and minute are not considered. The tuple is then hashed to compute a `dailyCluster` which identifies the unique day to which the redemption request belongs. Then, the `claimableEpoch` timestamp is computed from the tuple and the addition of `_DEFAULT_LIQUIDATION_HOUR` which is 0. Therefore, a new claimable epoch starts every day at 00:00:00 UTC and lasts for 24 hours.

Between the redemption request and the redemption execution, the shares are held by the vault. For every `dailyCluster`, the contract keeps track of the total amount of shares and assets that are redeemable during that day. It also records the list of `receivers` that will receive assets on that day. It is important to note that the share price is calculated when the redemption request is submitted. Thus, the share price is fixed for every pending redemption request, independent of the current share price.

`requestRedeem()` will revert if at least one of the `msg.sender` address, `receiverAddr`, or `holderAddr` is blacklisted. In case the `holderAddr` is not `msg.sender`, the `msg.sender` needs to have an allowance by the `holderAddr`.

Anyone can call `claim()` to execute a redemption for a given `receiver` and `(year, month, day)` epoch. The function will revert if the `msg.sender` or the `receiverAddr` is blacklisted. If the request is still pending, the function will also revert.

If the request has fulfilled the time lock duration, the shares are burned, and the assets are transferred to the `receiverAddr` address. The redemption request data is then cleared from storage.

Additionally, the contract implements several view functions such as `getWithdrawalEpoch()`, `getRequirementByDate()`, `getClaimableAmountByReceiver()`, `getBurnableAmountByReceiver()`, and `getScheduledTransactionsByDate()` to query the state of the current redemption requests.

BaseTokenizedAccount

The `BaseTokenizedAccount` contract is an abstract contract that inherits from `TimelockedClaimOnlyERC4626` and `BaseOwnable`. It implements `initialize()` from the `Initializable` contract. The owner and ERC20 are set during initialization. Furthermore, the vault deposits and withdrawals are paused. `configure()` is then required to be called to set parameters such as `maxDepositAmount`, `maxWithdrawalAmount`, or `maxChangePercent`. The `TimelockedCall` contract is also used to configure the owner with a permissioned scheduler. The delay for the time lock is set to 1 day.

The functions that are delayed by the `TimelockedCall` contract are:

- `BaseTokenizedAccount.transferOwnership()`
- `BaseTokenizedAccount.updateTimelockDuration()`
- `BaseTokenizedAccount.updateManagementFee()`
- `BaseTokenizedAccount.updateMaxChangePercent()`

`whitelistedSubAccounts` can be configured by the owner. Funds can be deposited and withdrawn to these subaccounts through `withdrawFromSubaccount()` and `depositToSubaccount()` by the operator. The subaccounts are then able to invest the funds into other on-chain products.

`processAllClaimsByDate()` is a function that allows anyone to process all redemption requests for a given date. The function will revert if the `msg.sender` is blacklisted. If the `receiverAddr` is blacklisted, the function will not revert. Instead, the assets will be redeemed and sent to a settlement account.

An `emergencyWithdraw` function is implemented, which allows the owner to withdraw all assets from the vault to a non-blacklisted address.

A `managementFeePercent` is defined, which is used to calculate the management fee for the vault. The management fee percentage represents a yearly fee rate and can be adjusted with `updateManagementFee()` by the owner. For the management fee to be applied, `chargeManagementFee()` must be called regularly. The fee is calculated from the total assets of the vault and the time since the last fee charge. Management and withdrawal fees can be sent to the `feesCollector` address by calling `collectFees()`.

Furthermore, the owner has the capability to:

1. `updateOperator()`: Updates the operator address.
2. `updateSettlementAccount()`: Updates the settlement account address.
3. `removeFromBlacklist()`: Removes an address from the blacklist.
4. `addToBlacklist()`: Adds an address to the blacklist.
5. `updateWithdrawalFee()`: Updates the withdrawal fee.
6. `pauseDepositsAndWithdrawals()`: Changes the pause state of deposits and withdrawals.
7. `updateIssuanceLimits()`: Updates the maximum deposit, withdrawal amount, and the maximum supply.
8. `updateTimelockDuration()`: Updates the duration of the withdrawal time lock.

TokenizedAccount

The `TokenizedAccount` contract inherits from `BaseTokenizedAccount` and implements a single function `updateTotalAssets()`. This function allows the operator to update the current amount of externally held assets by different subaccounts. The function will revert if the percentage change in assets is greater than `managementFeePercent`. Furthermore, it records the timestamp at which the asset amount was updated as the `managementFeePercent` is computed from `maxChangePercent`, which is a daily percentage change, and the last update timestamp.

2.3 Roles and Trust Model

The following roles are defined in the system:

Owner: The owner of the vault is fully trusted. The owner can modify the parameters of the vault, blacklist addresses from interacting with the vault. The owner can also withdraw all assets from the vault in case of an emergency.

Operator: The operator is in charge of managing the subaccounts and the vault. The operator can deposit and withdraw funds to and from the subaccounts and is fully trusted. The operator is defined by the owner and is therefore considered trusted by the owner to manage the vault.

Scheduler: Schedulers can call `schedule` and queue a call. They are assumed to act responsibly and schedule the calls as intended.

It is assumed that the decimals will be configured correctly when deploying the vault. More generally, the deployment parameters are assumed to be correct.

Tokens are only added after their compatibility has been assessed and properly tested.

The client is expected to perform a sufficiently large first deposit to the vault to prevent subsequent inflation attacks. This might be needed each time the vault starts from a zero balance, too.

The vault is assumed to be deployed behind a 1967 proxy contract. The proxy contract is assumed to be secure.

The time lock duration is considered to be larger or equal to 24 hours. The lag duration for withdrawals is assumed to be always selected in a way that leaves enough time for users to withdraw their funds before a parameter change can be executed. Thus, a time lock of 24 hours implies a lag duration of 0.

2.3.1 Changes in Version 2

The `TimelockedCall` contract was modified to additionally hash the consumer address of a scheduled call such that different consumers can consume the same call with the same parameters at the same time.

`TimelockedClaimOnlyERC4626` was modified to keep track of the `globalLiabilityAssets`, the assets pending for redemption. This is required for `BaseTokenizedAccount._getTotalAssets()` to calculate the total assets of the vault by accounting for the pending redemption requests.

Furtheremore, `_getTotalSupply()` was added to calculate the total supply of shares without the pending shares to be redeemed.

The blacklist checks in `TimelockedClaimOnlyERC4626.requestRedeem()` and `TimelockedClaimOnlyERC4626.claim()` were removed.

Additional assumption were made such as:

- The pool is assumed to be used with well-known tokens like USDC or WETH.
- August will monitor all subsequent actions of blacklisted addresses to prevent them from redeeming their shares as it is not enforced by the contract. Please refer to Blacklisted address can redeem shares for more information.
- It is assumed that `lagDuration` is set to reasonable value such that it allows August enough time to react to redeems from blacklisted addresses.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 Findings

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

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

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

Critical-Severity Findings	0
High-Severity Findings	1

- Blacklisted Address Can Redeem Shares Risk Accepted

Medium-Severity Findings	0
Low-Severity Findings	5

- Argument Sanitization Risk Accepted
- Fragmented Code Used Acknowledged
- Ineffective Check Maximum Mint Check Acknowledged
- Timestamp Manipulation Window Ignored Acknowledged
- Vault Is Not ERC4626 Compliant Acknowledged

5.1 Blacklisted Address Can Redeem Shares

Security High Version 2 Risk Accepted

CS-AUGCORE-001

In Version 2 of the code, checks for blacklisted users have been removed from `TimeLockedClaimOnlyERC4626.requestRedeem()`. Therefore, by calling `requestRedeem()` with a different `receiverAddr` then the `holderAddr` a blacklisted address can redeem shares for tokens, after the waiting period, to an alternate address that is not yet blacklisted.

Risk accepted:

August accepts the risk with the following statement:

We recognize that this is a possible edge case. However, if address A1 is already blacklisted, which required human intervention, we think it is reasonable to assume that we would be monitoring all subsequent actions of malicious address A1 (we have an internal process to add blacklisted address to a hexagate monitor). If A1 `requestRedeems()` with A2 as `receiverAddr` it is reasonable to assume that we would subsequently also blacklist A2, especially since we have the `lagDuration` time to react.

However, it is important to configure `lagDuration` to be large enough to allow for enough time to react. Indeed, `lagDuration` does not guarantee that the redeem request are delayed by at least `lagDuration` (refer to Redeem requests can be lagged by less than lagDuration for more details), resulting in a potentially to small time window to react.

5.2 Argument Sanitization

Correctness | Low | Version 1 | Risk Accepted

CS-AUGCORE-009

For several functions in the codebase, arguments are not sanitized before being used :

1. In `BaseTokenizedAccount.configure()`, `newUnderlyingAsset` and `newManagementFeePercent` are not sanitized.
2. In `BaseTokenizedAccount.initialize` the `_owner` is not checked if blacklisted.
3. In `BaseTokenizedAccount.updateTimelockDuration()`, `newDuration` is not checked to be smaller than 24 hours.
4. In `BaseTokenizedAccount.updateManagementFee()`, `managementFeePercent` is not checked to be smaller than 100% and to follow the format of being a percentage with 2 decimal places.
5. In `BaseTokenizedAccount.addToBlacklist` add to blacklist does not check if it is address zero or already blacklisted. `removeFromBlacklist` does not check if it has been blacklisted before.
6. In `BaseTokenizedAccount.updateMaxChangePercent()`, `maxChangePercent` bounds are not checked.
7. In `BaseTokenizedAccount.updateWithdrawalFee()`, `withdrawalFee` is not checked to be smaller than 100% and to follow the format of being a percentage with 2 decimal places.
8. In `BaseTokenizedAccount.enableAddress()` the `addr` is not checked for address zero but `enableAddresses` does.
9. The `DateUtils` library does not sanitize the inputs for `timestampFromDateTime` and `_daysFromDate`. Only year is sanitized. But as month is capped at `12` and days at `31` this should also be sanitized to avoid incorrect return values.
10. `BaseTokenizedAccount.transferOwnership` does not check if the new owner is the old owner.

Risk accepted:

August accepts the risk of passing in non-sanitized parameters.

5.3 Fragmented Code Used

Design | Low | Version 1 | Acknowledged

CS-AUGCORE-010

The code base is structured in an unconventional way and many definitions are fragmented and not used where they should be used. The following list illustrates the unconventional design:

- The variables `decimals`, `symbol` and `name` are defined in `BaseUpgradeableERC20v2` but not set in the constructor. Instead, they are set in `BaseTokenizedAccount`'s constructor.
- The mapping `isBlacklisted` is part of the contract `BaseUpgradeableERC20v2` but not used there and not part of the ERC20 standard.
- The error `InvalidTimestamp` and `MaxAllowedChangeReached` defined in `BaseTokenizedAccount` but used in `TokenizedAccount`.

- `AddressWhitelist` is responsible for the whitelisting functionality and not the owner management but sets the `_owner` in the constructor instead of the `BaseOwnable` which defines the state variable.
- `BaseTokenizedAccount` implements `initialize()` and calls `_disableInitializer` in its constructor. The OpenZeppelin library `Initializable` is imported in `BaseUpgradeableERC20v2`.
- `BaseUpgradeableERC4626v2` defines the `feesCollector` but all initialization and main interaction happens in `BaseTokenizedAccount`.
- In `TimelockedClaimOnlyERC4626` the state variable `settlementAccount` is defined but not used.

Acknowledged:

August acknowledges the code fragementation.

5.4 Ineffective Check Maximum Mint Check

Correctness | Low | Version 1 | Acknowledged

CS-AUGCORE-011

The `BaseUpgradeableERC4626v2.mint` function checks if `shares > maxMint(receiver)`. But `maxMint` returns `_maxSupply` in:

```
function maxMint(address) public view virtual override returns (uint256) {
    return _maxSupply;
}
```

Comparing a marginal increase with the `_maxSupply` possible does only limit each single deposit to the maximum possible supply and would break the intended invariant. However, this issue is only rated low as in `_mintErc20` the correct check `_canMint(amount)` is done. This raises the question about the need and effectiveness of the `_maxSupply` check. Additionally, there is a `maxDeposit` check limiting the possible amount to deposit in one call.

Acknowledged:

August decided to keep the code unchanged and is aware of the behavior.

5.5 Timestamp Manipulation Window Ignored

Correctness | Low | Version 1 | Acknowledged

CS-AUGCORE-006

In `TimelockedClaimOnlyERC4626._registerRedeemRequest()` a `_TIMESTAMP_MANIPULATION_WINDOW` of 5 minutes is added to the sum of the current timestamp and the `lagDuration`. However, due to the fact that `DateUtils.timestampFromDateTime()` rounds to the current day this additional 5 minutes is only effective if it pushes the timestamp to the next day. This means that the window is effectively ignored in most cases.

Acknowledged:

August acknowledges the finding.

5.6 Vault Is Not ERC4626 Compliant

Correctness | Low | Version 1 | Acknowledged

CS-AUGCORE-014

In `BaseUpgradeableERC4626v2`, the `previewWithdraw()` function is not ERC4626 compliant. The standard states : "MUST be inclusive of withdrawal fees.". However, the function does not include the withdrawal fee in the previewed withdrawal amount.

`maxDeposit()` and `maxMint()` are not ERC4626 compliant. Indeed, the standard states : "MUST return the maximum amount of assets deposit / *shares mint) would allow to be deposited for receiver and not cause a revert". However, `maxDeposit()` always returns `maxDepositAmount` without taking into account the share supply cap. Similarly, `maxMint()` always returns `_maxSupply` not accounting for the already minted shares. For example, if a vault would already hold some assets and had minted shares then the maximum amount of shares a user can mint is smaller than `_maxSupply`.

Furthermore, the standard defines that `convertToAssets` and `convertToShares` MUST NOT be inclusive of any fees that are charged against assets in the Vault. Due to the fact that the vault charges fees and they are kept in the vault together with the assets until withdrawn (as described in issues Withdrawal fees are counted towards the vault assets and Management fees are counted towards the vault assets) the calculation includes charged fees.

See ERC4626

In Version 2, `BaseTokenizedAccount_getTotalAssets()` was modified and made non-compliant with the ERC4626 standard as it now can revert if the total assets in the vault are less than the sum of the total fees and liability assets.

Acknowledged:

August acknowledges the findings with the following statement:

"Our tokenized account is not required to be compliant with the EIP-4626 standard. It supports most of the ERC4626 functionality though."

Therefore, users or developers interacting with the vault should be aware that the behavior of the vault is not fully be compliant with the ERC4626 standard.

6 Resolved Findings

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

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

Critical-Severity Findings	0
High-Severity Findings	2
Medium-Severity Findings	6
Low-Severity Findings	3
Informational Findings	1

- Blacklisted Can Be Bypassed Code Corrected
- Incorrect Price per Share Code Corrected
- Blacklisted Address Can Delay Legitimate Redeems Code Corrected
- Incorrect Balance Post Condition Code Corrected
- Management Fees Are Counted Towards the Vault Assets Code Corrected
- Queued Calls in TimelockedCall Are Shared Between Consumers Code Corrected
- Withdrawal Fees Are Counted Towards the Vault Assets Code Corrected
- processAllClaimsByDate() Can Revert Due to Inccorect receiverAddr Code Corrected
- Non-ERC20 Compliant Token Definition Code Corrected
- Tautology in Asset Comparison Code Corrected
- _maxSupply Cannot Be Set to 0 Specification Changed
- Inconsistent Pragma Use Code Corrected

6.1 Blacklisted Can Be Bypassed

Security High Version 1 Code Corrected

CS-AUGCORE-023

Certain addresses can be blacklisted by the vault owner to prevent these addresses of depositing, minting or redeeming shares. However, in `BaseUpgradeableERC20v2`, nothing prevents a blacklisted address from transferring shares to another address with `transfer()` or `transferFrom()`. This means that a blacklisted address can bypass the blacklist by transferring shares to a non-blacklisted address and interacting with the vault through the non-blacklisted address rendering the blacklist feature ineffective.

Code corrected:

`transfer` and `transferFrom` functions now check if the sender and the receiver are not blacklisted before transferring. If either the sender or the receiver is blacklisted, the function reverts.

6.2 Incorrect Price per Share

Correctness | High | Version 1 | Code Corrected

CS-AUGCORE-002

The system transfers the ownership from a user to the vault in case of a redemption request. It additionally locks the asset amount for the user when the redemption request was filed. This will lead to incorrect return values for all price per share calculations (including view functions). Most severely, the price per share calculation will be inconsistent for future deposits.

Consider the following scenario:

- User A has 50 shares
- User B has 50 shares
- Total assets 100

User A files a redemption request for their 50 shares at a 1:1 price. Thus, the redemption request will have 50 assets for user A. Let's assume the vault has a profit of 10 assets after the redemption request was filed. These 10 assets should belong to user B solely.

User C wants to deposit another 50 into the vault. There are now two options:

1. User C deposits WHILE user A's funds are still in the queue. The price per share would be 110 assets / 100 shares = 1.1
2. User C deposits AFTER user A withdraws the funds. The price per share would be 60 assets / 50 shares = 1.2

The correct price per share should be as soon as the profit is accounted 1.2.

Another explanation would be the following :

1. User C deposits WHILE user A's funds are still in the queue. User C receives 45 shares (45.45 rounded down) for their 50 assets. The total in the vault is now 145 shares and 160 assets. Then, if A withdraws their 50 shares for 50 assets the total in the vault will be 95 shares and 110 assets.
2. User C deposits AFTER user A withdraw the funds. User C receives 41 shares (41.66 rounded down) for their 50 assets. The total in the vault is now 91 shares and 110 assets.

The state of the vault should not be affected by the redemption request however, scenario 1 and 2 show that the amount of shares and assets in the vault are different, depending on if user withdraws before or after the deposit of user C.

Code corrected:

`BaseTokenizeAccount._getTotalAssets()` has been modified to not account for assets that are in the withdrawal queue. Furthermore, `TimelockedClaimOnlyERC4626._getTotalSupply()` has been implemented to not account for the shares that are in the withdrawal queue. Therefore, the amount of shares that are currently pending to be withdrawn for a given amount of assets are no longer accounted for in the vault.

6.3 Blacklisted Address Can Delay Legitimate Redeems

Security | Medium | Version 2 | Code Corrected

CS-AUGCORE-024

In Version 2, `TimelockedClaimOnlyERC4626.claim()` no longer checks if the `msg.sender` or the `receiverAddr` are blacklisted but instead `_claim()` sends the assets to the `settlementAccount` if the `msg.sender` or the `receiverAddr` are blacklisted. Due to the fact that `claim()` is permisionless, a blacklisted address can delay the legitimate redemption of shares for tokens by calling `claim()` for a legitimate redeem request. The consequence of this is that the assets will be redeemed to the `settlementAccount` instead of the intended `receiverAddr`.

Code corrected:

`_claim()` no longer sends the assets to the `settlementAccount` if the `msg.sender` or the `receiverAddr` are blacklisted. Instead, `_claim()` will now revert if the `msg.sender` or the `receiverAddr` are blacklisted.

6.4 Incorrect Balance Post Condition

Correctness | Medium | Version 1 | Code Corrected

CS-AUGCORE-003

In `_claim()`, the following check if performed at the end of the function:

```
if (balanceBefore - claimableAssets < IERC20(_underlyingAsset).balanceOf(address(this))) revert BalanceCheckFailed();
```

This check ensures that the balance of the vault cannot be inflated during the claim process. However, this check does not cover the case where the balance of the vault loses funds and decreases more than expected. The check will pass, if the vault has fewer funds than expected. This is also not in-line with the check in `BaseTokenizedAccount` where the following condition is used in a similar case:

```
IERC20(_underlyingAsset).balanceOf(address(this)) != balanceBefore - assetsToSend
```

Code corrected:

Both checks have been removed. August states that the checks are not required as the pool will use well known tokens such as USDC or WETH. Thus, it is now assumed that the tx will revert during the ERC20 transfer if the condition does not hold.

6.5 Management Fees Are Counted Towards the Vault Assets

Correctness | Medium | Version 1 | Code Corrected

CS-AUGCORE-004

A managemenent fee yearly rate is charged to the vault's total assets. However, the management fee is not subtracted from the vault's total assets until `collectFees()` is called. This leads to a slight discrepancy in the management fee calculation over time. For example, if `chargeManagementFee()` is called multiple times in a row without calling `collectFees()`, the management fee will be calculated on the total assets including the management fee from the previous calls to `chargeManagementFee()`. This leads to a slight overestimation of the management fee collected. Additionally, the share price will be inflated until the fees are collected.

This issue is connected with issue Withdrawal fees are counted towards the vault assets.

Code corrected:

`totalCollectableFees` is now substracted from `totalAssets` in `chargeManagementFee()` such that fees are not paid on uncollected fees. Furthermore, the share price is no longer inflated as the total assets no longer include the fees.

6.6 Queued Calls in `TimelockedCall` Are Shared Between Consumers

Design | Medium | Version 1 | Code Corrected

CS-AUGCORE-005

In `TimelockedCall`, a scheduler can `schedule()` a call to a consumer. The call is represented as the `keccak256` hash of the function selector and the arguments.

However, without specifications we assume that `TimelockedCall` could be used from multiple vaults. This means that if a call is scheduled on vault A, then the same call with the same argument cannot be scheduled on vault B assuming that A and B share the same `TimelockedCall` instance. Both calls will hash to the same key used in `queue`. Therefore, the call on vault B can only be scheduled once the call on vault A has been executed. This would be problematic if multiple vaults require the same parameter change at the same time.

Code corrected:

The address of the consumer is now hashed together with the hash of the function selector and the arguments. This ensures that the same call with the same arguments can be scheduled for different consumers.

6.7 Withdrawal Fees Are Counted Towards the Vault Assets

Correctness | Medium | Version 1 | Code Corrected

CS-AUGCORE-007

When a user withdraws their funds from the vault, the withdrawal fee is applied to the user's withdrawal amount. However, the withdrawal fee is not subtracted from the vault's total assets and remains in the vault's assets until `collectFees()` is called. Additionnaly, the shares backing the user's total withdrawal amount (including the fee) are burned, which inflates the vault's share price until the fees are collected.

Another consequence of this is that the management fee is inflated if the vault has unclaimed withdrawal fees. The management fee is computed as a percentage of the total assets in the vault and the percentage is a yearly rate. However, due to the fact that unclaimed withdrawal fees still count towards the vault's total assets, the management fee is inflated. This leads to a slighlty larger management fee being collected if the vault has unclaimed withdrawal fees.

A similar issue exists for the managment fees. See Management fees are counted towards the vault assets.

Code corrected:

`_getTotalAssets()` now substracts `totalCollectableFees` from the total assets.

6.8 `processAllClaimsByDate()` Can Revert Due to Inccorect `receiverAddr`

Design | Medium | Version 1 | Code Corrected

CS-AUGCORE-008

In `TimelockedClaimOnlyERC4626.requestRedeem()` the `receiverAddr` which will receive the withdrawn funds can be any address. However, if the `receiverAddr` is set to the vault address then `processAllClaimsByDate()` will revert due to the balance check performed at the end. Therefore, a single redeem request can block others redeem request from being processed with `processAllClaimsByDate()`. Nevertheless, it is still possible to execute single requests with `claim()`.

Code corrected:

A check was added to `registerRedeemRequest()` to prevent the vault address from being set as the receiver address. This should prevent the balance check in `processAllClaimsByDate()` from reverting.

6.9 Non-ERC20 Compliant Token Definition

Correctness | Low | Version 1 | Code Corrected

CS-AUGCORE-012

The current token contract implementation in `BaseUpgradableERC20` is not ERC-20 compliant. The ERC-20 definition clearly states

> Note Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event

But the implementation will revert if the value is `0` due to:

```
require(value > 0, "Amount cannot be zero");
```

Code corrected:

The check has been removed from the code to comply with the ERC-20 standard.

6.10 Tautology in Asset Comparison

Correctness | Low | Version 1 | Code Corrected

CS-AUGCORE-013

In `TimelockedClaimOnlyERC4626._registerRedeemRequest` one of the earliest conditions checks that the share balance of the `holderAddr` is bigger than the share to be redeemed. After the shares has been converted into assets, another condition checks that the `assetsAmount` needed after fees is bigger than the assets one would get for the `holderAddr`'s shares.

The asset amounts are converted with the same function `_convertToAssets` but `assetsAmount` potentially has some fees deducted.

```
assetsAmount = _convertToAssets(shares) - fees
maxWithdraw(holderAddr) = _convertToAssets(_balances[holderAddr])
```

As we know that `_balances[holderAddr] >= shares`, `assetsAmount` minus some fees will never exceed `maxWithdraw(holderAddr)` because the same conversion is used for two values that we compared before.

Code corrected:

The condition `assetsAmount > maxWithdraw(holderAddr)` was modified to `assetsAmount >= maxWithdrawalAmount` to check if the amount of assets to be withdrawn does not exceed the maximum amount allowed.

6.11 `_maxSupply` Cannot Be Set to 0

Correctness | Low | Version 1 | Specification Changed

CS-AUGCORE-015

In `BaseUpgradeableERC20v2`, `_maxSupply` represents the maximum circulating supply of the token. Moreover, the natspec indicates that if it is set to 0 it represents an unbound supply. However, `_setMaxSupply` will revert if the new supply is set to 0.

Specification changed:

The NAT spec has been updated to reflect the new behavior. The new specification states that *_maxSupply* cannot be 0.

6.12 Inconsistent Pragma Use

Informational | Version 1 | Code Corrected

CS-AUGCORE-019

The project is supposed to be deployed on different chains. As the used EVM version might differ on each chain, the deployer needs to ensure that the code is supported by all versions that potentially will be used. Most contracts use version `^0.8.19`. But multiple contracts like `BaseReentrancyGuard`, `DateUtils`, `TimelockedCall`, `LightweightOwnable` and more use `>= 0.8.26`. We recommend to:

- properly test the compatibility of the code with each chain it shall be deployed,
- to use a fixed (non-floating) and consistent pragma throughout the whole code base and
- to compile with a consistent EVM version.

Code corrected:

All contracts are now using the pragma `^0.8.19`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Code Duplication

Informational Version 1 Acknowledged

CS-AUGCORE-016

`BaseTokenizedAccount.processAllClaimsByDate` and `TimelockedClaimOnlyERC4626._claim` share critical functionality. To prevent inconsistencies it might be beneficial to have a shared function to handle the state updates.

Acknowledged:

August acknowledges the code duplication but will not perform any changes.

7.2 Disabling a Scheduler Does Not Invalidate Queued Requests

Informational Version 1 Risk Accepted

CS-AUGCORE-017

In case a scheduler is disabled but already scheduled certain calls, the scheduler cannot add to the queue anymore but the remaining queued request are not invalidated. In case a rogue scheduler shall be disabled, this might leave remaining calls in the queue.

Risk accepted:

August accepts the risk and does not plan to implement any changes.

7.3 Emergency Withdraw Blacklist

Informational Version 1 Acknowledged

CS-AUGCORE-018

Currently, an emergency withdraw can withdraw to any address that is not blacklisted. It might increase trust if funds will be withdrawn to a whitelisted address that is known instead of any address not blacklisted.

Acknowledged:

August acknowledges the information without further changes.

7.4 Missing Events

Informational | Version 1 | Code Partially Corrected

CS-AUGCORE-025

Several functions in the codebase do not emit an event when modifying relevant state. Without specifications, this list might be incomplete and August need to check if all events are emitted as intended:

1. The constructor of `AddressWhitelist` does not emit an event when the owner is set.
2. `BaseUpgradeableERC4626v2._updateIssuanceLimits` or `updateIssuanceLimits` do not emit an event when updating the limits.

Many functions in `BaseTokenizedAccount` do not emit events including:

1. The function `addToBlacklist` and `removeFromBlacklist` do not emit an event when an address is added or removed from the blacklist.
2. The function `removeWhitelistedSubaccount` and `addWhitelistedSubaccount` do not emit an event when a subaccount is added or removed from the whitelist.
3. The function `updateSettlementAccount` does not emit an event when the settlement account is updated.
4. `updateTimelockDuration` does not emit an event when updating `lagDuration`.
5. `collectFees` silently collects the fees.
6. `updateMaxChangePercent` does not emit an event when updating the `maxChangePercent` limit.
7. `updateWithdrawalFee` does not inform the users about a changed withdraw fee.
8. `updateSettlementAccount` changes the settlement account silently.
9. `depositToSubaccount` and `withdrawFromSubaccount` send and receive funds to and from a subaccount silently.
10. `addToBlacklist` and `removeFromBlacklist` do not inform users about changes in the blacklist
11. `addWhitelistedSubaccount` and `removeWhitelistedSubaccount` do not inform about changes in the whitelisted subaccounts.

Code partially corrected:

The events `MaxChangePercentUpdated()` and `FeesCollected()` were added.

7.5 Non-indexed Events

Informational | Version 1 | Acknowledged

CS-AUGCORE-020

We identified the following events with parameters that potentially could be indexed to make filtering and searching easier:

- `HashScheduled`
- `HashConsumed`
- `SchedulerEnabled`
- `SchedulerDisabled`

- `OnAddressEnabled`
- `OnAddressDisabled`
- `WithdrawalRequested`
- `WithdrawalProcessed`
- `FeeCollectorUpdated`
- `OnEmergencyWithdraw`
- `OperatorUpdated`

Acknowledged:

August acknowledges the finding and will not implement any changes.

7.6 Owner Can Be Address Zero

Informational | Version 1 | Acknowledged

CS-AUGCORE-021

The `LightweightOwnable` contract allows the owner to directly transfer the ownership to any address without sanity checks. This allows to set the owner to address zero or another address that is not controlled. Ultimately, revoking ownership. Without proper specification, we don't know if this is intended behavior. Hence, we hereby highlight this behavior to check if this is intended.

Acknowledged:

August did not change the code base.

7.7 Unused Functions

Informational | Version 1 | Acknowledged

CS-AUGCORE-022

The function `_reentrancyGuardEntered` in `BaseeRentrancyGuard` is not used in the code and marked as internal.

August acknowledges the finding but did not perform any changes.

8 Notes

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

8.1 Centralized Power

Note **Version 1**

As described in the roles and trust assumptions, the system requires significant trust in certain roles. Users should carefully assess the trust assumptions and inherent risks before using the protocol.

8.2 External Asset Amount Capped Downside

Note **Version 1**

In `TokenizedAccount.updateTotalAssets()`, the external asset amount can be updated by the operator. However, if the change in the external asset amount exceeds `maxAllowedChangePerc` the transaction will revert. However, this cap is also applied if the external asset amount is being decreased. In the unlikely event that the external asset amount decreases significantly, the update to the external asset amount will revert. This would prevent the operator from updating the external asset amount to the correct value thus overpricing shares for users.

8.3 Inflation Attack

Note **Version 1**

The current code base is vulnerable to the well-know inflation attack on vaults by donating significant funds to the vault after the first deposit was a small amount. Through rounding errors all subsequent users will be rounded down and lose part or all of their vault contribution.

August is aware of this and tries to mitigate this by doing a trusted first deposit that adds sufficient funds to mitigate the issue.

8.4 Redeem Requests Can Be Lagged by Less Than `lagDuration`

Note **Version 1**

When a redeem request is registered, the `dailyCluster` hash and the `claimableEpoch` are computed. These values determine the epoch from which the redeem request can be executed.

```
(year, month, day) = DateUtils.timestampToDate(block.timestamp + _TIMESTAMP_MANIPULATION_WINDOW + lagDuration); // _TIMESTAMP_MANIPULATION_WINDOW = 5 min

// The hash of the cluster
bytes32 dailyCluster = keccak256(abi.encode(year, month, day));

// The withdrawal will be processed at the following epoch
claimableEpoch = DateUtils.timestampFromDateTime(year, month, day, _DEFAULT_LIQUIDATION_HOUR, 0, 0);  // _DEFAULT_LIQUIDATION_HOUR = 0
```

The above code shows that the redemption time is rounded down to day. Therefore, if the `lagDuration` is 1 hour, a redeem request submitted at 2:00 PM will be processed immediately because the `year`, `month`, and `day` will remain the same as the current date. This results in the same `dailyCluster` hash

and the same `claimableEpoch`. Consequently, `lagDuration` must be bigger than 24h but still smaller than `timeLockDuration` to allow users to withdraw funds in case they disagree with certain scheduled parameter changes.

8.5 Theoretical Assets Can Be Accounted Multiple Times

Note Version 1

A subaccount can invest the funds on their own discretion. They need to ensure to not have significant cyclic dependencies when investing funds. With a cyclic dependency we mean funds that are invested into a protocol that directly or indirectly will deposit into the vault that the subaccount took the funds from. Investors and subaccounts should properly assess the flow of funds and prevent this form happening.

8.6 Timelockduration Fixed

Note Version 1

Once a `timeLockDuration` is set in `TimelockedCall.initScheduler` it cannot be changed anymore.

8.7 Token Decimals

Note Version 1

The shares of a vault will implicitly have the same decimals as the underlying tokens of the first deposit into the vault. Consequently, the decimals need to be configured correctly when setting up the vault. In case of very low decimals rounding errors might lead to issues like a more effective inflation attack if not sufficient funds are initially deposited from August as planned or fees might have bigger rounding issues. Therefore, each token should be assessed and tested carefully before using it.

8.8 Token Support

Note Version 1

On request August stated the vault shall support ERC20 tokens. We need to highlight that the vault will not be able to support all ERC20 tokens. Even though tokens are ERC20 compliant they might exhibit incompatible behavior. E.g., tokens that charge fees on transfers, re-basing tokens, tokens with low decimals and more. August MUST assess and test each token thoroughly before using it in a vault.

8.9 `claim()` and `processAllClaimsByDate()` Differences

Note Version 1

In `claim()`, if the `receiverAddr` is blacklisted the transaction will revert. However, in `processAllClaimsByDate()`, the transaction will not revert if the `receiverAddr` is blacklisted.

Instead, the funds will be redeemed and sent to a settlement account. August expects this behavior as `processAllClaimsByDate()` would prevent legitimate withdrawals from succeeding if it would revert.