

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	22
4	Terminology	23
5	Open Findings	24
6	Resolved Findings	26
7	Informational	48
8	Notes	58

1 Executive Summary

Dear Resupply team,

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

Resupply implements reUSD, a decentralized stablecoin backed by Collateralized Debt Positions (CDP). The CDPs are created by supplying crvUSD or frxUSD into a Curve Lend or Fraxlend market. Resupply additionally implements a Decentralized Autonomous Organization (DAO) to govern the system with the governance token RSUP.

Across the protocol, the assumption that the price of the underlying tokens is equal to the price of reUSD is made as it is expected that both frxUSD and crvUSD are stable and pegged to 1 dollar. We emphasize that the security of the protocol depends on this assumption holding. Any deviation from the peg could put user funds at risk and falls outside the scope of this security assessment.

Our audit primarily focused on critical subjects such as rounding errors, accurate debt accounting, access control, and integration with underlying lending markets. Security regarding these areas is high.

Additionally, we examined general subjects including functional correctness, reward distribution, vesting correctness, Convex integration, and mechanisms for redemption and liquidation. Overall, security for these general subjects is high.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

-Severity Findings	0
-Severity Findings	0
-Severity Findings	5
•	5
-Severity Findings	17
•	13
•	1
•	3



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 Resupply 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	03 January 2025	7fe8a2921d70bb75bd22277aced5b991373138de	Initial Version
2	13 February 2025	272d24ada93fbc648d5bed63528e83c2bb99c5d0	Second version
3	21 February 2025	b72b8cafcd13701b91d33e2cc4a19d4713cb01ac	Third version
4	24 February 2025	9a4c3962833f77e5e401daf592f888807ce491f4	Fourth version
5	25 February 2025	0beb774a9669869e86422ba3fbb5a2054fbb2aaa	Fifth version

For the solidity smart contracts the version 0.8.28 was chosen and for compilation targets the shanghai evm version was fixed.

The following directories and files were considered in scope for the assessment:

- src/protocol/
- src/dependencies/
- src/dao/staking/
- src/dao/tge/
- src/dao/Core.sol
- src/dao/GovToken.sol
- src/dao/Treasury.sol
- src/dao/Voter.sol
- src/dao/emissions/EmissionsController.sol
- src/dao/emissions/receivers/SimpleReceiver.sol

2.1.1 Excluded from scope

Anything not listed above is considered out of scope, this includes but is not limited to:

- src/dao/emissions/receivers/ExampleReceiver.sol
- src/dao/emissions/receivers/SimpleReceiverFactory.sol
- src/libraries/
- test/
- lib/

2.2 System Overview

This system overview describes the initially received version () of the contracts as defined in the [Assessment Overview](#).

At the end of this report section, we have added subsections for each of the changes according to the versions.

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

Resupply offers the Resupply protocol, a stablecoin backed by Curve Lend and Fraxlend liquidity provider positions.

The system consists of two main components which interact with each other, the DAO and the core protocol. The core protocol contains all the logic for the stablecoin and the DAO is responsible for the governance of the system.

2.2.1 Core Protocol

The following graph shows the interaction between the core protocol contracts, note that it does not include every interaction but rather the most important ones.

2.2.1.1 Resupply Pair

Users can deposit collateral to a `ResupplyPair` contract to mint the system's stablecoin, reUSD. The collateral backing reUSD is in the form of ERC-4626 vault shares, which represent liquidity provider positions in lending markets from *Curve Llama Lend* or *Fraxlend*.

A Resupply Pair specifies the type of `collateral` (vault share token) that users must provide to mint reUSD, paired with the `underlying` asset required to open the external LP position. The allowed underlying assets, `crvUSD` or `frxUSD`, are stablecoins and are assumed to maintain a constant price of one dollar.

The reUSD minted to the borrower represents their debt in the system, which is tracked in terms of debt shares. Each debt share corresponds to a proportional claim on the total outstanding debt. The price of a

debt share depends on the ratio of `totalBorrow.amount / totalBorrow.shares`, where `totalBorrow` represents the Pair's global debt accounting.

```
// Contract Level Accounting
VaultAccount public totalBorrow;

struct VaultAccount {
    uint128 amount;
    uint128 shares;
}
```

To open and manage CDPs on Resupply, the `ResupplyPair` contract provides the following basic functions:

- `borrow()`: mints the specified amount of reUSD to the receiver and computes the respective number of borrow shares in the system.
- `repay()`: repays the specified number of debt shares to the system and burns the respective amount of reUSD from the payer.
- `addCollateralVault()`: increases the borrower's collateral balance according to the deposited collateral amount.
- `addCollateral()`: same as `addCollateralVault()` but user provides the underlying asset which is deposited into the respective vault on the external lending market first.
- `removeCollateralVault()`: decreases the borrower's collateral according to the specified amount and transfers it back to the receiver.
- `removeCollateral()`: same as `removeCollateralVault()` but the collateral token is directly redeemed for the underlying token which is then transferred to the receiver.

Users can borrow reUSD from a Pair, as long as they remain solvent and the `borrowLimit` of the Pair is not exhausted. A borrower is considered solvent if their Resupply position remains within the defined `maxLTV` (maximum Loan-To-Value) ratio or if the `maxLTV` is set to 0. Additionally, a minimum borrowing amount is enforced on every borrow and on partial repayments (user must hold at least `minBorrowAmount` after repay). A Pair may be paused, effectively prohibiting users to borrow more reUSD within that pair, but still allowing the other actions such as repaying.

Interest on the total amount of reUSD borrowed in a Pair accrues at most once per block, causing the borrowers' debt to gradually grow over time. To fully reclaim their collateral, a borrower must repay the borrowed reUSD along with their share of the accrued interest. Hence, as interest accumulates, a borrower's position moves closer to the specified `maxLTV` ratio.

Pairs are deployed through the `ResupplyPairDeployer` using the `CREATE2` opcode. Deployed Pairs are stored in the state of the `ResupplyRegistry`. Multiple pairs with the same {collateral, underlying} tuple may exist simultaneously.

2.2.1.2 Exchange Rate

In several occurrences, such as computing the LTV of a user to check their solvency, liquidations or redemption, it is required to price the debt of a user in terms of their collateral. Using the `BasicVaultOracle` as exchange rate for the collateral in underlying, the value in collateral of some debt shares s can be obtained as $\frac{s}{S} \cdot D \cdot \frac{1}{\text{pricePerShare}}$, where S is the total debt shares of the pair, D is the total debt of the pair and `pricePerShare` is the price of the collateral in underlying.

2.2.1.3 Liquidation handling & Insurance Pool

Liquidations are tied to the protocol's insurance pool. The `InsurancePool` contract allows users to deposit reUSD in exchange for shares. Users are incentivized to deposit into the insurance pool as they can earn rewards in both reUSD and RSUP, as well as potentially other tokens. The contract includes a `withdrawQueue` mechanism that introduces a delay on withdrawals, preventing immediate exits. Assets in the insurance pool are designated to cover liquidations and clear bad debt within the protocol.

For depositors, the `InsurancePool` provides the following basic functionalities:

- `deposit(_amount) / mint(_shares)`: deposits the corresponding amount of reUSD into the insurance pool in exchange for minting pool shares to the caller - given that the user is currently not in the withdrawal queue.
- `exit()`: starts withdrawal cooldown period and puts the caller into the withdrawal queue (default delay = 7 days).
- `withdraw(_amount) / redeem(_shares)`: burns the corresponding number of shares from the caller and transfers back the respective amount of assets - given that enough time has passed since user called `exit()`.

Liquidations in the `Pair` must go through the `InsurancePool`, hence, anyone can liquidate an insolvent borrower via the system's `LiquidationHandler`. Upon a successful liquidation (given sufficient liquidity in the insurance pool):

- The borrower's debt in the `Pair` is reset and the corresponding amount of reUSD is burnt from the insurance pool (debt repaid by insurance pool).
- The collateral amount covering the debt + liquidation fee is transferred from the `Pair` to the liquidation handler. It is redeemed for underlying and forwarded to the insurance pool as rewards.
- Although this action decreases the price per share of the insurance pool, through `getRewards()`, shareholders can claim their share of the rewards, which, because of the fee, should generally be worth more than the burned reUSD.

The caller of the `LiquidationHandler` does not earn any direct rewards for liquidating a borrower. They might only be incentivized to do so if they are a shareholder of the insurance pool and expect to earn sufficient rewards from the liquidation to cover for their gas costs.

2.2.1.4 Fees handling

The system collects fees in the `FeeDeposit` contract, which are then distributed to the insurance pool, treasury and governance tokens stakers, according to a predefined split. A `ResupplyPair` currently accrues the following types of fees:

- Borrow fees (interest gained): charged on borrowed assets, based on the interest rate formula $\max(2, \text{APR})$. Here, APR refers to the underlying lending rate and the risk-free rate is computed using the `sfrax` oracle.
- Mint fees: charged when minting reUSD; currently not planned to be used but can be set through the `Core`.
- Redemption fees: applied when redeeming reUSD for collateral; fixed percentage (default = 1%), of which only a portion of the fee is collected by the protocol, the rest is directed to the debt holders of the pair.

Fees are distributed at most once per epoch. If fees are not withdrawn, they remain in the `Pair` contract accumulating in `claimableFees` or `claimableOtherFees`.

Using `withdrawFees()`, anyone can mint reUSD corresponding to the fees accrued and withdraw them to the `FeeDeposit` contract. Upon receiving the fees, the `FeeDeposit` notifies the `pairEmission` streamer of the amount of interest-based fee obtained by the given pair. This is used by the streamer to adjust the weight of the different pairs, as a pair generating more fees should receive more emissions of the governance token RSUP.

At most once per epoch, and always with one epoch delay, the `FeeDepositController` can call `distributeFees()` on the `FeeDeposit` contract to pull the reUSD and distribute it. The fees are distributed according to the predefined split.

Treasury

The `Treasury` contract receives a portion of the protocol fees. Through the `Core`, these funds can later be withdrawn from the treasury.

Insurance Pool rewards

The Insurance pool shareholders receive part of the protocol fees in reUSD as rewards. The rewards are sent from the `FeeDepositController` to the `RewardHandler` followed by a call to `queueInsuranceRewards()`. The function will do the following:

- Queue the reUSD rewards in the `InsuranceRevenue` streamer.
- Pull from the `InsuranceEmission` simple receiver the emission of RSUP allocated to the insurance pool.
- Queue the RSUP rewards in the `InsuranceEmission` streamer.

Calling `claimInsuranceRewards()` on the `RewardHandler` will distribute the rewards accumulated in the two streamers to the insurance pool shareholders. RSUP will be distributed as a reward token in the pool, reUSD will simply inflate the price per share.

Staking rewards

The stakers of the governance token RSUP receive part of the protocol fees in reUSD as reward. The rewards are again sent from the `FeeDepositController` to the `RewardHandler` but this time followed by a call to `queueStakingRewards()`. The function will do the following:

- Forward and notify the `GovStaker` contract of the reUSD rewards.
- Pull from the `DebtEmission` simple receiver the emission of RSUP allocated to the reUSD borrowers (debt holders in pairs).
- Queue the RSUP rewards in the `PairEmission` streamer.

RSUP rewards stored in the `PairEmission` streamer are re-distributed to the pairs based on their weights, corresponding to the amount of fees they have generated.

2.2.1.5 Rewards handling

The system implements multiple flows of rewards from the following sources:

1. reUSD pair fees (interest based fees, mint fees and redemption fees)
2. A pair might stake their collateral into Convex to earn rewards¹ (CRV, CVX, gauge rewards)
3. RSUP emissions

The rewards are directed to the following actors of the system:

1. `ResupplyPair` shareholders (RSUP, convex staking rewards)
2. Insurance pool shareholders (reUSD and RSUP)
3. Governance token stakers (reUSD)
4. The treasury (reUSD)

Flow of rewards

Convex staking rewards of a given pair are to be distributed to that pair shareholder. This is usually done when the pair calls `Registry.claimRewards()`, itself calling `RewardHandler.claimRewards()`, claiming the convex rewards and sending them back to the `Pair` that will account for them as rewards.

RSUP emissions are directed to the insurance pool and all pairs, based on their weight. As described above in the Fees handling section, calling `distribute()` on the `FeeDepositController` will queue the emission directed at the insurance pool in the `InsuranceEmission` streamer and the emission directed at the pairs in the `PairEmission` streamer. Calling `RewardHandler.claimInsuranceRewards()` will distribute the RSUP rewards accumulated in the `InsuranceEmission` streamer to the insurance pool shareholders. The RSUP rewards accumulated in the `PairEmission` streamer are re-distributed to the pairs based on their weights, corresponding to the amount of fees they have generated. This is done for each pair when it calls `Registry.claimRewards()`.

reUSD rewards are sent to the insurance pool, the governance token stakers and the treasury. As described in the Fees handling section, they get accumulated in the `FeeDeposit` contract and are distributed by the `FeeDepositController`. The rewards directed at the `InsurancePool` get accumulated in the `InsuranceRevenue` streamer and can later be distributed by calling `RewardHandler.claimInsuranceRewards()`. The rewards directed at the governance token stakers and the treasury are directly sent to the corresponding contract.

Distribution to shareholders

The three receiver contracts, `ResupplyPair`, `InsurancePool` and `GovStaker` all implement a checkpoint based mechanism to fairly distribute rewards to their shareholders based on their weight. For the pair and the pool `RewardDistributorMultiEpoch` is used, and `MultiRewardsDistributor` for the `GovStaker`.

Both contracts are very similar and use integral-based checkpoint mechanisms to keep track of the rewards accumulated by the users.

1. A global integral is computed as the reward per token to be distributed. As this depends on the total supply of shares, it must be updated upon any user action as the total supply might change.
2. When a given user calls the contract, their accumulated rewards can be computed as the difference between the global integral and their user specific integral multiplied by their amount of share. Their user specific integral is then updated to the global integral.

In both the `ResupplyPair` and the `InsurancePool`, on top of this, an epoch system is implemented to account for the share refactoring. Share refactoring is a mechanism that, when triggered, scales down the total supply of shares by `SHARE_RECAFTOR_PRECISION` (default set to `1e12`). This means that users cannot catch up with the global integral seamlessly across epochs as their share must be scaled down for each epoch. Hence, integrals are reset for each epoch and users need to iterate over each epoch when making a checkpoint of their rewards.

Other use of the distributors

The checkpoint mechanisms presented above are also used for the following purposes:

- In the pair, the distributor is used to distribute `WriteOff` tokens to the users. These tokens are not rewards nor claimable but are used to adjust the user's collateral balance when a redemption happens.
- In the insurance pool, the distributor is used to distribute various pairs' underlying tokens obtained from a liquidation. Those are not really rewards as the shareholders lost a comparable value in reUSD.

2.2.1.6 Redemptions handling

To maintain the peg of the reUSD stablecoin with the underlying stablecoins (as in a floor price), the protocol implements a redemption mechanism. Anyone who holds reUSD, can redeem their reUSD for collateral at a 1:1 ratio (minus a small fee) with a specific Pair via the `RedemptionHandler`. Since `Resupply` enables *socialized redemptions*, all borrowers within a Pair are affected proportionally by redemptions. A successful redemption (given there is sufficient debt in the pair) of `_amount` debt tokens implies:

- Reducing the total debt amount of the Pair by `_amount - protocolFee` reUSD.

- Burning `_amount` reUSD tokens from the redeemer.
- Transferring the equivalent of `_amount` in collateral tokens (minus redemption fee) to the receiver.

Redemptions are subject to several requirements; they must be above a minimum amount, and the Pair must have sufficient debt to cover the redemption and be left with a given minimum amount of debt that can be set by the `Core`.

The total borrow amount of the Pair is decreased and as such the transferred collateral should be removed proportionally from all borrowers.

For every collateral token that has been freed via redemption, a so-called WriteOff token is minted by the Pair to itself. WriteOff tokens are non-transferable and only mintable by the pair. They serve an accounting function, ensuring that borrowers' collateral is adjusted correctly. WriteOff tokens are treated as any other reward tokens that the pair might receive, except that they are not claimable by users. They get distributed by the system to shareholders with respect to their share of the total debt by using the reward system, using the checkpoint mechanism. Before any user action on the Pair, and after the user checkpoint happens, the WriteOff tokens allocated to the users are "claimed" by the user by removing them from the user accounting and subtracting the corresponding amount from the user's collateral balance.

Redemptions and Refactoring

As redemptions keep the `totalBorrow.shares` constant but decrease the `totalBorrow.amount`, the ratio of `totalShares/totalAmount` will grow over time. Refactoring of borrow shares should counteract this effect and is implemented as follows:

```
if(uint256(_totalBorrow.amount) * SHARE_REFACTOR_PRECISION < _totalBorrow.shares){
    _increaseRewardEpoch(); //will do final checkpoint on previous total supply
    _totalBorrow.shares /= uint128(SHARE_REFACTOR_PRECISION);
}
```

This adjusts the total borrow shares of the system and scales them down by a factor of $1e12$. The borrow shares of each user are adjusted when the respective account is checkpointed.

It is important to note that once the ratio of `totalShares/totalAmount` has surpassed 10^{12} and a refactor is done, for another refactor to be required as consequence of the same redemption, assuming `minimumLeftoverDebt = 10000 * 10 **18`, the minimum amount of reUSD to be borrowed in the system would need to be at least $10^{16} = 10^{16} * 10^{18}$ wei. This scenario is considered infeasible.

2.2.1.7 Leveraging and Swapper

The Resupply Pair implements a `leveragedPosition()` function. It allows borrowers to amplify their positions by borrowing reUSD and swapping it for additional collateral using trusted swapper contracts.

Users start by supplying an initial amount of `collateral` to be added to their account and borrowing a specified amount of asset tokens (reUSD) which is minted, but not necessarily over-collateralized at that point. These borrowed assets are sent to a whitelisted swapper contract, which swaps them for collateral tokens, following a predefined path. The collateral obtained from the swap is then added to the user's existing collateral, effectively increasing their collateral balance.

Similarly, using `repayWithCollateral()`, a user can repay their debt using their collateral, or in other words, "deleverage" their position. The user specifies the amount of collateral they want to use to repay their debt, and the collateral is swapped for reUSD using the swapper contract. The reUSD is then used to repay the debt. In case more reUSD is obtained than needed to repay the debt, the excess is returned to the user.

It is assumed that the swapper contracts use trusted swap pools, and the addition of new pools is access-controlled, allowing only trusted entities through the `Core` to update the swap paths.

2.2.2 DAO

The Resupply DAO is responsible for the governance of the system. The governance token RSUP can be staked to vote on proposals.

The following graph shows the interaction between the DAO contracts, note that it does not include every interaction but rather the most important ones.

2.2.2.1 Core

The `Core` contract is the source of truth for system-wide values and contract ownership. It acts as an access control module. It is expected to be the owner of all the contracts in the system. And can hence call privileged functions on them to adjust various parameters or to perform administrative tasks.

The access control is implemented as follows:

- The `Voter` can always perform arbitrary calls through the `Core` contract. Such calls are the effect of governance proposals that were accepted by the DAO.
- Defined operators perform calls they are allowed to do. Permissions can be set for a given operator for a given target contract and function selector. This allows for fine-grained access control. For more broad permissions, the operator can be given access to a given function selector for arbitrary target contracts.
- Adding or removing permissions to an operator is itself done through a call from the `Core` to itself by calling `setOperatorPermissions()`. Initially, only the `Voter` has permissions to do this but can grant this permission to other operators.
- Changing the address of the `Voter` is done through a call from the `Core` to itself by calling `setVoter()`.

In the rest of the system overview, when we mention that a function can only be called by the `Core`, it means that the function is only callable by the `Voter` following a governance vote, or an operator that has the required permissions.

Additionally, the `Core` contract provides the epoch system to the rest of the system. All other contracts are expected to read the current epoch from the `Core` contract.

2.2.2.2 Treasury

The `Treasury` is responsible for managing the DAO's funds. It can receive ETH and tokens from arbitrary senders while only the `Core` can withdraw funds from it. The following functions can only be called by the `Core`: `retrieveToken()`, `retrieveTokenExact()`, `retrieveETH()`, `retrieveETHExact()`, `setTokenApproval()`.

The treasury is expected to receive part of the protocol fees. Hence, it should receive `reUSD` from the `FeeDepositController`.

2.2.2.3 Governance staking

Through the `GovStaker` contract, users can stake `RSUP` to receive rewards from the protocol, and to vote on proposals.

Staking and Voting power

Staking is done through the `stake()` function, and anyone can stake tokens for someone else. Upon receiving a given amount of `RSUP`, the contracts increment both the total staking supply and the staking balance of the user with `amount`. Staking immediately starts earning rewards. However, the voting power associated with the staked `RSUP` is only updated at the beginning of the next epoch. Until then, the stake is considered pending (as opposed to realized stake).

Unstaking is subject to a cooldown period. Through `cooldown()` or `exit()`, a user, or someone allowed by the user, can start the cooldown period. During that period, the user's `RSUP` tokens are sent to the `GovStakerEscrow`. Hence, the user does not earn rewards anymore and cannot use the stake to vote. After the cooldown period, the user or someone allowed by them can call `unstake()` to finally withdraw the `RSUP` from the escrow. Only realized stake can be unstaked, pending stake has to be realized first.

Perma staking

Using `irreversiblyCommitAccountAsPermanentStaker()`, a user can commit to be a perma-staker. This means that the user will never be able to unstake their `RSUP`. This is a one-way operation. The user will still be able to vote and receive rewards however. It is expected that `Convex` protocol sub-DAO and `Yearn` protocol sub-DAO will be perma-stakers and once available, their vested `RSUP` will be permanently staked.

Instead of trusting the sub-DAO to stake their `RSUP`, they will be registered to the `VestManager` using a `PermaStaker` contract. The contract allows claiming available tokens in the `VestManager` but immediately perma-stake them.

At construction, the `PermaStaker` commits to be a perma-staker. The owner of the contract can add or remove operators. Both the owner and operators can:

- Perform arbitrary calls on behalf of the `PermaStaker` as long as the target is not the `VestManager` using `execute()` and `safeExecute()`.
- Call `claimAndStake()` to claim available tokens in the `VestManager` and immediately perma-stake them.

Rewards

Rewards are accounted for using a checkpoint system implemented in the `MultiRewardsDistributor` contract. After a token has been added to the reward list by the `Core`, the `GovStaker` can receive rewards to be distributed from the corresponding distributor using `notifyRewardAmount()`. This will start a new distribution period over which the rewards are distributed linearly. In case the previous period was not fully distributed, the remaining rewards are added to the next period. Users receive rewards based on their share of staking balance over time.

GovStaker migration



It is possible that the `GovStaker` contract is upgraded. Since the contract is not behind a proxy that would mean deploying a new contract, having users migrate their state and updating the address of the `GovStaker` in the rest of the system. This would mean:

- Deploying a new `Voter` and having the `Core` update the address of the `Voter`.
- Deploying new `RewardHandler`, updating the `Registry` to point to it and having that new `RewardHandler` update the address of the `GovStaker`.
- Updating the different `SimpleReceiver` to change the `approvedClaimer` to the new `RewardHandler`.

Finally, because perma-stakers cannot unstake their RSUP, a specific migration process is needed for them. The `GovStaker` implements a function `migrateStake()` which can only be called by perma stakers. If the address of the `GovStaker` doesn't match the address of the `GovStaker` called, it means that a migration happened. The call will hence unstake the RSUP from the old `GovStaker` and stake it in the new one pointed by the registry, also calling a specific hook `onPermaStakeMigrate()` on the new `GovStaker` to let it know that the given account should be marked as perma staker.

Restricted functions

The following functions can only be called by the `Core`:

- `setCooldownEpochs()` to update the cooldown duration.
- `addReward()` to add a new reward token that can be received by the contract from a given distributor and then be distributed to stakers. A duration is also set, rewards are distributed linearly over that duration.
- `setRewardsDistributor()` to set the distributor of a given reward token.
- `recoverERC20()` to recover ERC20 tokens sent to the contract by mistake.

For each reward token, the following functions can only be called by the reward's distributor:

- `setRewardsDuration()` to update the duration of the reward distribution.
- `notifyRewardAmount()` to notify the contract that a given amount of reward token is available for distribution.

2.2.2.4 Voter

The `Voter` contract is responsible for the governance of the system. Governance stakers can create proposal and vote on them. A proposal that has enough votes can be executed by anyone. In the following staking balance and voting power are used interchangeably, and refers to the amount of RSUP staked by a given account at the previous epoch. Users give approval to other users to create proposal and vote on their behalf.

Proposal creation

Any governance staker with enough voting power can create a proposal using `createNewProposal()`. The voting power required is defined as a percentage of the total staking supply `minCreateProposalPct`, which can be updated by the `Core`. Each staker can create a proposal at most once every 3 days. A proposal contains multiple actions which are calls to be made through `Core.execute()` as the `Core` is the only contract that can perform privileged operations. It should be noted that a proposal calling `Core.setOperatorPermissions()` to update the permissions for an operator to call `cancelProposal()` on any contract or more specifically on the `Voter` contract are enforced to only contain that action.

Proposal voting

Voting can be done through `voteForProposal()`, by providing the proposal ID and percentages to indicate how to split the caller's voting power on yes and no. Votes can be done for 1 week after the proposal creation. Once voted on a proposal, the caller cannot vote again on the same proposal or cancel their vote.

Executing a proposal

After 8 days following the proposal creation, and during 3 weeks, provided that the proposal has enough votes, anyone can execute the proposal using `executeProposal()`, which will sequentially call the actions of the proposal. The proposal will be marked as executed and cannot be executed again. A proposal is considered passed if:

1. The quorum is reached, defined as a percentage of the total staking supply at proposal creation `passingPct`. The quorum can be updated by the `Core`.
2. The percentage of yes votes is greater than the percentage of *no* votes.

Cancelling a proposal

Using `cancelProposal()`, a proposal can be cancelled if it has not been executed yet. The caller must be the `Core`, and hence only the `Voter` or an operator with the required permissions can cancel a proposal. Proposals containing an action to set the permissions of an operator to call `cancelProposal()` on the `Voter` contract cannot be cancelled.

Restricted functions

The following functions can only be called by the `Core`:

- `cancelProposal()`
- `setMinCreateProposalPct()`
- `setPassingPct()`

2.2.2.5 Governance token

The governance token is a simple ERC20 token. At construction the initial supply is minted to the `VestManager` contract to fund the vesting of the different stakeholders.

Restricted functions

- The `minter` can mint new tokens using `mint()`. The `minter` is expected to be the `EmissionsController` contract.
- The `Core` can call `setMinter()` to update the `minter` address as long as the `minter` has not been *finalized*. Once finalized using `finalizeMinter()`, the `minter` cannot be updated anymore.

2.2.2.6 Vesting

Vesting is managed through the `VestManager`. The contract is expected to receive the initial supply of the governance token at construction. The following vests are expected to be managed by the `VestManager`:

- Convex protocol sub-DAO, vesting over 5 years
- Yearn protocol sub-DAO, vesting over 5 years
- Frax Protocol for use of Fraxlend codebase, vesting over 1 year
- Treasury, vesting over 5 years
- Prisma Burns, vesting over 5 years
- Airdrop for the team, vesting over 1 year
- Airdrop for Prisma hack victims who not yet been made whole, vesting over 2 years
- Airdrop lock penalty, vesting over 5 years.

Both Sub-DAO allocations are to be staked perpetually for voting power and reward.

Creation of the vests

After construction, the Core can call `setInitializationParams()` to initialize the vesting of the different stakeholders. This will handle three different category of vests:

- **Vesting with a user target:** For Convex protocol sub-DAO, Yearn protocol sub-DAO, Frax Protocol, and the treasury, an address is passed as the receiver. A new vest is created for each of them during `setInitializationParams()`.
- **Redemptions:** For Prisma burns, `setInitializationParams` is given a maximum amount of tokens that can be redeemed `_maxRedeemable`. A redemption ratio is computed as the governance token allocation for burns divided by `_maxRedeemable`. Calling `redeem()` will burn the user's specified amount of `prisma`, `yprisma` or `cvxprisma` and create a new vest for the user with the corresponding amount of governance token according to the redemption ratio.
- **Airdrops, managed using merkle trees:** For the airdrops, a merkle root is provided at initialization (Or, in the case of the Lock penalty, will be provided later using `setLockPenaltyMerkleRoot()`). Users can call `merkleClaim()` with a merkle proof to prove that they are entitled to a given amount of governance token. The merkle proof is verified by the contract and the user is credited with the corresponding amount of governance token with the creation of a new vest.

Vest that are created in all three cases have the vest manager creation date as starting date. This means that there are no incentives to create vests early in the system. A user might have multiple vests. If the vests have the same duration, they are merged into a single vest. If the vests have different durations, the user can claim the tokens from the different vests seamlessly.

Claiming tokens

Once a vest has been created for a user, tokens can be claimed using:

- `claim()`.
- `claimWithCallback()`, which is only callable by the vest owner or an account allowed by the owner and will call a given callback function after the claim is done.

In both cases, the user can set parameters for their vests using `setClaimSettings()`. They can allow permissionless claims which for both functions will prevent anyone else from claiming the tokens for them (including approved accounts for `claimWithCallback()`). They can also set a custom recipient address.

Restricted functions

The following functions can only be called by the Core:

- `setInitializationParams()` to initialize the vesting of the different stakeholders.
- `setLockPenaltyMerkleRoot()` to set the merkle root and allocation for the lock penalty airdrop.

2.2.2.7 Emission controller

The emission controller is responsible for the emission of the governance token. It can mint new tokens and distribute them to the different emission receivers.

Emission Schedule

At construction, the contract is given an emission schedule, an amount of epochs between emissions rate changes, and a tail rate, which should be used once the emission schedule is over. The emission schedule is a list of emission rates.

When the receiver claims their emission, they also mint new tokens for the current epoch and previous epochs where no calls were made to the Emission controller. The emission controller keeps track of the last epoch where emissions were minted, and will mint tokens for all epochs between this epoch and the current epoch, following the schedule.

Receivers and weights

Valid receivers are given a weight, which represents the percentage of the total emission that they should receive. The sum of the weights should be 100%. The receiver can fetch their emission using

`fetchEmissions()`, this will mint new tokens for the current epoch and previous epochs and compute the allocated tokens for the receiver. In case the receiver is not active, tokens are accrued in a unallocated global variable, and will later be recovered by the Core using `recoverUnallocated()`. `fetchEmissions()` do not transfer the tokens to the receiver, instead `transferFromAllocation()` should be called afterward with a given receiver to transfer the tokens to the receiver.

Restricted functions

The following functions can only be called by the Core:

- `registerReceiver()`: to register a new receiver. Except for the very first receiver to be registered, which gets 100% of the emission, the receivers are being registered with a weight of 0%.
- `deactivateReceiver()`: receivers are active by default, but can be deactivated, in which case their allocation is accrued in the unallocated global variable.
- `activateReceiver()`: to reactivate a receiver.
- `setReceiverWeights()`: to update the weights of the receivers, the sum of the weights should always be 100%. It is not enforced that the new weights should be smaller than the previous ones, it is hence possible to have an increase in emission weight after an update of the schedule.
- `setEmissionsSchedule()`: to update the emission schedule.
- `recoverUnallocated()`: to recover the unallocated tokens.

2.2.2.8 Emission receivers

The `SimpleReceiver` contract is a simple contract that can receive tokens from the `EmissionController`. It can be used to distribute the emission to a receiver. The system have two simple receivers, the `InsuranceEmissionReceiver` for insurance pool shareholders to receive RSUP and the `DebtEmissionReceiver` to distribute RSUP to the debt holders in the pairs. In both cases, token are first fetched by the `RewardHandler` and then distributed to the corresponding streamer.

`allocateEmissions()` is permissionless and can be called to call `fetchEmissions()` in the `EmissionController`. `claimEmissions()` is restricted to the Core or given approved claimers and will forward the emission to a given receiver.

Restricted functions

- `setApprovedClaimer()` can only be called by the Core to set the approved claimer for the contract.
- `claimEmissions()` can only be called by the Core or the approved claimers.

2.2.3 Changes in versions

2.2.3.1

In , various changes were made to the system to fix issues raised in this report. Additionally, the following changes were made:

- The `Stablecoin` and `GovToken` were converted to Layer Zero OFT. Both tokens are owned by the Core contract and ownership cannot be transferred. By default, the Core is the delegate of the tokens in the Layer Zero protocol and can update sensible settings. With this change, the permit functionality of the tokens was removed.
- The `Swapper` now adds the underlying lending market of the pair calling as a pairing in case one of the path steps is `TYPE_UNDEFINED`.
- The requirement for an action updating the `cancelProposal` permission (for a given operator) to be the only action in a proposal was removed. However, a proposal containing such an action cannot be cancelled.

- Migration of the `LiquidationHandler` can only be performed by the new handler pointed to by the registry. The new handler receives the collateral token and the amount of debt cleared from the old handler. The new handler is expected to correctly handle the debt cleared.
- The `_totalDebtAvailable` of the pair is no longer upper bounded by `registry.getMaxMintable()`, which was hardcoded to `type(uint256).max`.
- In the pair, `currentUtilization()` now accounts for interest accrued since the last update.

2.2.3.2

In , various changes were made to the system to fix issues raised in this report. Additionally, the following changes were made:

- `MultiRewardsDistributor.setRewardsDuration()` can now also be called by the Core.
- Proposals in the voter must be given a description, the description can be changed by the Core using `updateProposalDescription()`.

2.2.3.3

In , the following changes were made:

- Users may claim `GovStaker` rewards on behalf of others via `getReward(address _account)`.
- Users can set a redirect address for their `GovStaker` rewards.

2.2.3.4

Various changes were made to the system to fix informational issues raised in this report.

2.3 Trust model

2.3.1 Roles

- The `Core` contract is expected to be the owner of any contract inheriting from `CoreOwnable`.
- In the `FeeDeposit`, the operator is expected to be set as the `FeeDepositController`.
- It is assumed that the `ResupplyRegistry` is an operator of the `Stablecoin` contract.
- It is assumed that the `RewardHandler` is an approved claimer of the `SimpleReceiver` contracts (`DebtEmissionReceiver` and `InsuranceEmissionReceiver`). Other approved claimers should not call `claimEmissions()` with the `RewardHandler` as the receiver as this could forward emissions to the wrong streamer.
- Operators of the `ResupplyPairDeployer` are trusted to deploy non-malicious pairs. That includes meaningful collateral, oracle, rate calculator, max LTV, borrow limit, redemption, mint, and liquidation fee as well as underlying staking protocol address and ID. In the worst case, a malicious operator could deploy a pair with no borrow limit and a max LTV of 0% which would allow for infinite minting of reUSD.
- Adding a permission to an operator in the `Core` is expected to be done only for trusted entities, and with a clear understanding of the permissions given as well as using the least privilege principle. Failing to do so could lead to a malicious operator calling a function that could have a negative impact on the system. In the worst case, an operator could take over the DAO by setting the `Voter` to a malicious contract, or mint an infinite amount of reUSD or governance token.

2.3.2 Tokens

- Both underlying and collateral tokens of pairs are expected to be 18 decimals.



- Collaterals are expected to be compliant with the ERC4626 standard, and not implement any non-standard behavior like fee on transfer or hooks upon transfer, deposit, or withdrawals.
- In general, the system is expected to be used with tokens that are compliant with the ERC20 standard, that have no special behavior like rebasing, fee on transfer, or hooks upon transfer. This includes the various rewards expected to be distributed in the system.
- All tokens used in the system are expected to implement the optional EIP-20 `decimals()` function.
- Reward tokens used in the `MultiRewardDistributor` and `SimpleRewardStreamer` are expected to have 18 decimals.
- `crvUSD` and `frxUSD` are assumed to be the underlying tokens of the pairs. *They are assumed to be both pegged to 1 USD.*
- Across the codebase, the "emission/governance token" and "stablecoin/revenue token" tokens are expected to be the governance token and `reUSD` respectively, implemented in the `GovToken` and `Stablecoin` contracts. Hence, those tokens are expected to be fully compliant with the ERC20 standard.

2.3.3 General trust model and assumptions

The protocol is expected to be deployed on Ethereum mainnet.

The economic model behind the stablecoin was not fully in scope of this review. Specifically, both `frxUSD` and `crvUSD` are assumed to be pegged to 1 USD for this review. In reality, this might not always hold, and both underlying assets could have different prices. In such cases, as long as the pairs whose underlying is the most expensive still contain collateral, redemptions should maintain the peg of `reUSD` to that underlying. Hence, `reUSD` should generally be pegged to the most expensive underlying.

Pairs

All parameters of pairs are expected to be set correctly and with reasonable values by the deployer. A misconfigured pair could allow for infinite mint of `reUSD`. The underlying lending protocol is assumed to be solvent. For example, Curve Lend does not account and socialize bad debt to the liquidity providers, in such case, instead, the last share-holders of the vault would be impacted and could not redeem all their share in case of bad debt.

Pair deployer

The size of the creation code of the pair is expected to be small enough to fit in the two contracts used to store both partitions of the code.

Swappers

The Governance, or approved operators are expected to make sure that pools vaults and tokens whitelisted in the `Swapper` are compliant with their standard, not malicious and behave as expected. Pools used by the swapper should not use native ETH as input or output token.

Liquidations and Insurance Pool

Liquidators do not directly earn a fee for liquidating a position. It is expected that major shareholders of the insurance pool will trigger liquidations, or that liquidations will be triggered by bots run by the Resupply team directly to protect the system.

In the insurance pool, the first reward token is assumed to be `RSUP`. Additionally, the deployer of the insurance pool is expected to send the initial `reUSD` tokens corresponding to the unbacked shares minted in the constructor. The insurance pool is expected to have all underlying tokens of the pairs as reward tokens. Moreover, if the core calls `LiquidationHandler.distributeCollateralAndClearDebt()`, it is expected that the corresponding collateral is also a reward token of the insurance pool. The pool is expected to have a reasonable number of reward tokens, as it might become too gas expensive to distribute them all.

In case of migration of the `LiquidationHandler()`, it is expected that all remaining debt that was cleared in the old handler when migrating is correctly handled by the new handler.

Convex

- In case the Pair uses Convex to stake the collateral, it is assumed that the convex pool have at most 13 rewards tokens including CRV and CVX. Since the WriteOff token and RSUP use 2 of the 15 slots of the pair.
- If Convex is used, it is assumed that only PIDs greater than or equal to 151 will be used, as older PIDs had a different interface.

Vesting

The elements of the different arrays given as parameters to `VestManager.setInitializationParams()` function are expected to follow the order defined in `InitVestManager.s.sol`. In the `VestManager`, a given account is expected to appear at most once in a given merkle tree. The provided `_maxRedeemable` should be greater than or equal to the circulating supply of PRISMA, YPRISMA, and CVXPRISMA combined, if this not the case, the vest manager could become insolvent. Moreover, it is expected that this value account for past and future penalties from vePRISMA lock breaking. PRISMA collected from past penalties is expected to be accounted for but burned and not redeemable.

RewardHandler

The `RewardHandler` is expected to always claim emissions from the `debtEmissionsReceiver` and the `insuranceEmissionReceiver` by itself. Shall an approved claimer do it with the `RewardHandler` as the receiver, the emissions could be forwarded to the wrong streamer.

SimpleReceiver

All simple receivers of the system should be given the `EmissionController` as controller. Approved claimer are trusted to not steal the emission of the system. In the worst case, a malicious approved claimer could claim the emission of the receiver and keep the tokens for themselves.

Emission controller

The emission controller should have at least two receivers: the `debtEmissionReceiver` and the `insuranceEmissionReceiver`. The provided schedule should match the expected RSUP schedule.

Treasury

The treasury contract is expected to receive reUSD from the `FeeDepositController`

GovToken

The minter of the `GovToken` is expected to be the `EmissionController`. In this review we assume that the minter role is never given to any other address. If not the case, the minter could be updated by a malicious actor to mint an infinite amount of tokens. The initial supply of the governance token is expected to be minted to the `VestManager` contract.

Several contracts in the system such as the `GovStaker` or the `Voter` expect the total supply of `GovToken` to be smaller than some value to optimize storage costs. Hence, it is expected that the total supply of the token will never exceed $2^{40} * 2^{18}$.

Core

It is assumed that the `Core` contract is the owner of all `CoreOwnable` contracts of the system. Trust assumptions regarding the *operators* are detailed in the *role* section above.

Voter

Proposals can be executed out of order by the governance, they should not have any side effect that would make them dependent on the order of execution. Parameters of the voter are expected to be set correctly and with reasonable values. In this review, we assume that a majority of the governance stakers are not malicious and will vote in the best interest of the protocol. If this is not the case, the protocol could be taken over by a malicious actor and all the funds drained.

GovStaker

The `GovStaker` is expected to receive a stream of rewards from the `RewardHandler`. When migrating to a new `GovStaker`, it is expected that `onPermaStakeMigrate()` can be called multiple times for a given perma-staker, after the first call, calls are not expected to revert and should be no-op. If this is not the case, perma-stakers might lose the stake they had pending at the time of the migration.

Perma-stakers

Accounts marked as perma-stakers in the `GovStaker` are expected to be addresses where the `PermaStaker` contract is deployed. The owner and operators are trusted to not use the perma-staker's voting power for malicious purposes. In the worst case, given the large allocation of the two sub-DAOs, they could use the power to vote on proposals that could drain the system, or upgrade the `GovStaker` contract to an implementation that would allow them to unstake.

In the case of a `GovStaker` migration, perma-stakers should manually remove their `GovToken` approval to the old `GovStaker` if it is not to be trusted anymore.

Layer Zero

As of [this version](#), both the Stablecoin and the Governance tokens are implemented as Layer-zero Omni-chain fungible tokens (OFT). The configuration required for managing the Layer-zero applications on multiple chains is considered out of scope for this review and should be performed by the `delegate` (set as the `Core` at construction) in the `LayerZero` endpoint, that includes:

- Correctly setting peers to whitelist on each chain.
- Correctly setting the `enforcedOptions` to ensure users pay a predetermined amount of gas for delivery on the destination transaction. It should be computed such that messages sent from a source have sufficient gas to be executed on the destination chain. Setting a gas limit too small could mean that no executor has an incentive to pay for the delivery of the message at the destination, and the message should either be dropped by the admin, or some executor should execute it at a loss to resume messages handling.
- Correctly setting a DVN configuration, including optional settings such as block confirmations, security threshold, the Executor, max message size, and send/receive libraries. If no send and receive libraries are explicitly set, the OFTs will fall back to the default settings set by LayerZero Labs. In case LayerZero Labs changes the default settings, the OFTs will be impacted and use the new default settings which implies a trust in LayerZero Labs.

The OFTs support a max token supply of $(2^{64} - 1) / (10^6)$ and use a shared decimal value of 6. This means that the tokens are expected to never reach such supply. Additionally, cross chain transfers are rounded down to match the shared decimals, meaning that it is not possible to transfer less than 10^{12} wei of the token for example.

It should be noted that the owner of the tokens can set arbitrary `peers` or `delegate` which could lead to draining or losing all funds.

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			
Medium			
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 Open 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:

- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation

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

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-Severity Findings	3

- [Insurance Pool Initial Mint Can Be Reset](#)
- [Pair Might Be Insolvent for Collateral](#)
- [RSUP Reward Can Be Invalidated in the InsurancePool](#)

5.1 Insurance Pool Initial Mint Can Be Reset

CS-RESUPPLY-034

Upon system setup, the insurance pool is initialized with an amount of dead shares (1e18). However, this initial mint is insufficient to fully mitigate potential inflation risks.

Over time, due to refactoring during redemptions, the total amount of shares can decrease, eventually depleting the dead shares. If all depositors withdraw, it could be that the system is left with zero total shares. In this case, the insurance pool is effectively reset without the initial mint.

At this stage, since the pool allows for donations, the system could become susceptible to an inflation attack.

Risk accepted:

Resupply acknowledges the risk, does not make changes to the codebase and states:

We expect changes to not be worth the cost extra gas etc.

Feel a high level of assurances given:

- We will have the Treasury hold a permanent IP position
- `minimumHeldAssets` is `> 0` , requiring all users (including Treasury) to exit in order to become vulnerable

5.2 Pair Might Be Insolvent for Collateral

CS-RESUPPLY-037

Because of how `WriteOff` tokens are distributed, precision losses might happen. It could be that the amount of collateral redeemed is greater than the total amount of collateral being written off to the different shareholders. This means that the last person to remove their collateral might not be able to remove all of it as the protocol would be insolvent. This could be however mitigated by donating collateral to the contract.

Risk accepted:

Resupply acknowledges the risk and states:

Understood there may be some dust remaining but should be very minimal. Donations can also fill.

5.3 RSUP Reward Can Be Invalidated in the InsurancePool

CS-RESUPPLY-006

In the `InsurancePool`, if `_clearWithdrawQueue()` is called but the first reward token `RSUP` is invalidated for any reason, the user's withdrawal queue would be reset, but they would not lose their `claimable_reward` since `reward.reward_token` would be `address(0)` in `claimable_reward[reward.reward_token][_account];`

Risk accepted:

Resupply acknowledges the issue.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the [Open Findings](#) section.

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

-Severity Findings	0
--------------------	---

-Severity Findings	0
--------------------	---

-Severity Findings	5
--------------------	---

- [Core execute\(\) Reentrancy](#)
- [Liquidation Might Revert](#)
- [Pair Timestamp Is Never Updated](#)
- [Perma-stakers Can Unstake](#)
- [Precision Loss](#)

-Severity Findings	14
--------------------	----

- [Floating Dependency Versions](#)
- [Missing Non-Reentrancy Guard in Swapper](#)
- [Missing Sanity Checks](#)
- [Non-ERC20 Compliant Token Can DOS the Reward Token Insertion](#)
- [Pair Creation Code Cannot Be Smaller Than 13,000 Bytes](#)
- [Sfrax Cycle Length Can Differ From REWARDS_CYCLE_LENGTH](#)
- [Unprotected Functions Can Be Reentered](#)
- [Unsafe Approvals](#)
- [Swapper.swap\(\) Always Returns 0](#)
- [Swapper Does Not Check Slippage](#)
- [allocationByType Is Incorrect](#)
- [getMaxRedeemableDebt\(\) Can Be Inaccurate](#)
- [getReceiverId\(\) Result Can Be Misleading](#)
- [totalDebtAvailable\(\) Does Not Account for Interests](#)

Informational Findings	28
------------------------	----

- [Approval Not Removed in Swapper](#)
- [Arbitrary Call Targets](#)
- [Code Readability](#)
- [Conditions Always False](#)
- [Double Pausing a Pair](#)
- [Empty Sfrax Vault Edge Case](#)
- [Event Field Indexing](#)
- [Function Mutability](#)

- Function Visibility Too Broad
- Incorrect Documentation
- LTV Precision
- Missing Events
- Missing or Incomplete NatSpec
- Order of Events When Adding Reward Token
- Protected Keys Should Be Constants
- Selector Taken From the Wrong Interface
- Solidity Version Not Fixed
- Unreachable Code
- Unused Errors
- Unused Events
- Unused Imports
- Unused Parameters
- Unused Struct
- Variable Shadowing
- Vesting Duration for Sub-DAOs
- SimpleReceiver Implementation Can Be Initialized
- getUnstakableAmount Does Not Account for cooldownEpochs
- previewAddInterest() Does Not Set lastTimestamp

6.1 Core `execute()` Reentrancy

CS-RESUPPLY-031

When executing an action through the `Core` contract using `execute()`, one can potentially re-enter the contract and execute another action before the first one is completed. This can lead to unexpected behavior and potential security issues if the first action leaves the system in an inconsistent state.

This means that both, proposal action executions and operator actions, can be re-entered and can re-enter each other.

Implications of this issue include that a proposal cannot be seen as an atomic operation, as it can be re-entered and potentially modified by another action.

Code corrected:

The `execute()` function has been made non-reentrant.

6.2 Liquidation Might Revert

CS-RESUPPLY-032

According to EIP 4626, `maxWithdraw` is the:

Maximum amount of the underlying asset that can be withdrawn from the owner balance in the Vault, through a withdraw call.
MUST return the maximum amount of assets that could be transferred from owner through withdraw and not cause a revert

However, it is never specified that `maxWithdraw(addr)` must return the same value as `redeem(maxRedeem(addr), dst, addr)`.

In the function `processCollateral()` of the `LiquidationHandler`. Assuming we are in the case where `withdrawable <= debtByCollateral[_collateral]`, we have `toBurn == IERC4626(_collateral).maxWithdraw(address(this))`.

Several cases can happen:

1. If `toBurn == withdrawnAmount`, the entire `withdrawnAmount` will be burned from the insurance pool and distributed as rewards (in underlying) as expected.
2. If `toBurn < withdrawnAmount`, `withdrawnAmount` will be distributed to the pool shareholders, but only `toBurn` will be burned. Instead, it could have been that the entire `withdrawnAmount` is burned.
3. If `toBurn > withdrawnAmount`, this means that the result of `maxWithdraw()` is greater than the actual amount redeemed, this will result in a revert with an underflow in the event emission when computing `withdrawnAmount - toBurn`.

The second case is not a vulnerability, but rather a small inefficiency, as the difference between `withdrawnAmount` and `toBurn` is not expected to be large, however, the third case is a vulnerability as it can lead to a revert, for example, it can happen in the case of Fraxlend:

Assuming the state of the Frax lending pair to be:

```
_totalAsset.amount = 120_000 * 10 ** 18
_totalAsset.shares = 115_000 * 10 ** 18
_totalAssetsAvailable = _totalAsset.amount - _totalBorrow.amount = 8_000 * 10 ** 18
```

And assuming the balance of the `LiquidationHandler` to be `10_000 * 10 ** 18` shares.

Calling `pair.maxWithdraw()` returns `8_000 * 10 ** 18` since:

```
_totalAssetsAvailable = _totalAssetAvailable(_totalAsset, _totalBorrow)
                        = 8_000 * 10 ** 18
_totalUserWithdraw = _totalAsset.toAmount(_ownerBalance, false)
                    = 10_000 * 10 ** 18 * 120_000 * 10 ** 18 // (115_000 * 10 ** 18)
_maxAssets = _totalAssetsAvailable < _totalUserWithdraw ? _totalAssetsAvailable : _totalUserWithdraw;
            = _totalAssetsAvailable
            = 8_000 * 10 ** 18
```

On the other side, we get `maxRedeem()` == `_maxShares` where `_maxShares` is computed as:

```
_totalAssetsAvailable = _totalAssetAvailable(_totalAsset, _totalBorrow)
                        = 8_000 * 10 ** 18
_totalSharesAvailable = _totalAsset.toShares(_totalAssetsAvailable, false)
                        = 8_000 * 10 ** 18 * 115_000 * 10 ** 18 // (120_000 * 10 ** 18)
                        = 7_666_666_666_666_666_666_666
_maxShares = _totalSharesAvailable < balanceOf(_owner) ? _totalSharesAvailable : balanceOf(_owner)
            = _totalSharesAvailable
            = 7_666_666_666_666_666_666_666
```

Finally, we have `redeem(maxRedeem()) == _amountToReturn` where `_amountToReturn` is computed as follows (where `_shares == maxRedeem()`):

```
_amountToReturn = _totalAsset.toAmount(_shares, false)
                = 7_666_666_666_666_666_666_666 * 120000 * 10 ** 18 // (115000 * 10 ** 18)
                = 7_999_999_999_999_999_999_999
```

In this case, `toBurn == 8000 * 10 ** 18` and `withdrawnAmount == 8000 * 10 ** 18 - 1` which would lead to an underflow in the event emission.

Code corrected:

The code has been corrected and now recomputes the `toBurn` amount. In case the redeemed amount is greater than the `collateralDebt`, `toBurn` is capped by both the `collateralDebt` and `maxBurnable`, s.t. underflows are avoided and that `burnAssets()` cannot revert.

6.3 Pair Timestamp Is Never Updated

CS-RESUPPLY-003

In the `RewardHandler`, `setPairWeight()` does not update the `pairTimestamp[_pair]` mapping as `==` is used instead of `=`. This means that for all pairs, `lastTimestamp` will always be equal to `block.timestamp - epochLength`. Following, the time delta for computing the rate will always be equal to the `epochLength`, meaning that the pair weights are defined only based on the amount of fee received since the last call, independently of the time that has passed.

Code corrected:

The code has been corrected by replacing the comparison with the `=` operator, s.t. the entry in the `pairTimestamp` mapping is set to the current block timestamp.

6.4 Perma-stakers Can Unstake

CS-RESUPPLY-001

Through the `PermaStaker` contract, perma-stakers should not be able to unstake RSUP. However, in case of a `GovStaker` migration, they can call `claimAndStake()` followed by calling `exit()` to unstake their RSUP. This is possible as long as they do not call `migrateStake()` in the old staker contract. They would be able to recover all RSUP tokens that were available to claim from the vesting contract at that moment.

Code corrected:

The perma-staker can now only use `claimAndStake()` if its locally cached staker address matches the one stored in the registry. In case of migration, to update it to the `GovStaker`, the perma-staker must call `migrateStaker()`, which will make sure that the new `GovStaker` contract is aware that the staker is a perma-staker.

6.5 Precision Loss

CS-RESUPPLY-002

In the `SimpleRewardStreamer` and `MultiRewardDistributor`, when depositing rewards or doing a checkpoint of the rewards, precision loss may occur, and, depending on the reward token's number of decimals and value, be significant.

For the following, we take the example of distributing wBTC token over one week. The token has 8 decimals, and a price of 96000 104000 USD at the time of writing.

- In `notifyRewardAmount()`, when dividing the reward amount by the duration, precision loss may occur. For example, if 1 wBTC is deposited, 0.00208 (216 USD) will be lost in the process.
- In `updateReward()`, each time the global integral is updated, precision loss may occur. For example, assuming a `totalSupply` of $1000 * 10^{18}$, and a rate of 165 wei per second (1 wBTC distributed over one week),

- Calling `updateReward()` only once after one week will increase the global integral by $165 * 604800 * 10^{18} // (1000 * 10^{18}) = 99792$

- Calling the function every block (12 seconds per block) will increase the global integral by $\frac{604800}{12} * (165 * 12 * 10^{18} // (1000 * 10^{18})) = 50400$

This is a difference of 0.49392 wBTC (51k USD) over one week.

In both cases, funds not distributed are locked in the contract and cannot be recovered.

Code corrected::

The `SimpleRewardStreamer` and `MultiRewardDistributor` have been adjusted and now require reward tokens to have a precision of 18 decimals which mitigates the substantial precision loss.

Note that the precision loss is still present and could be significant for tokens whose price per wei is very high.

6.6 Floating Dependency Versions

CS-RESUPPLY-033

The versions of the contract libraries in `package.json` are not fixed.

The caret `^version` will accept all future minor and patch versions while fixing the major version. With new versions being pushed to the dependency registry, the compiled smart contracts can change. This may lead to incompatibilities with older compiled contracts. If the imported and parent contracts change the storage slot order or change the parameter order, the child contracts might have different storage slots or different interfaces due to inheritance.

In addition, this can lead to issues when trying to recreate the exact bytecode.

Code corrected:

The library versions in `package.json` have been fixed.

6.7 Missing Non-Reentrancy Guard in Swapper

CS-RESUPPLY-004

The `Swapper` inherits from `ReentrancyGuard`. However, none of its functions is marked as non-reentrant. While pools and vaults whitelisted are expected to be trusted, either `swap()` should be marked as non-reentrant or `ReentrancyGuard` should be removed from the inheritance chain.

Code corrected:

The function `swap()` has been marked as non-reentrant.

6.8 Missing Sanity Checks

CS-RESUPPLY-035

In general addresses passed to setters are not prevented to be special addresses such as the zero address, the contract address, or the sender address.

Additionally, the following non-exhaustive list shows functions lacking important sanity checks:

1. In the `VestManager`, nothing prevents `setLockPenaltyMerkleRoot` to be called before `setInitializationParams`. This could override the merkle root previously set with zeros.
2. In `MultiRewardsDistributor.addReward()`, it is not enforced that the reward token should not be the `stakeToken`.
3. In `Swapper.addPairing()`, no sanity check on the swap type, or token indexes is performed.
4. In `LiquidationHandler.migrateCollateral()`, no check is performed on `_to`.
5. In `ResupplyPair.setMaxLTV()`, no check is performed on `_newMaxLTV`.
6. In `ResupplyPair.setLiquidationFees()`, no check is performed on `_newLiquidationFee`.
7. In `SimpleRewardStreamer.getReward()`, no check is performed on `_forwardTo`.
8. In `ResupplyPair._updateConvexPool()`, the `_pid` is not checked to be non-zero.
9. In `RewardHandler.checkNewRewards()`, the `_pid` is not checked to be non-zero.
10. In `Voter._containsProposalCancelerPayload()`, the size of data is not explicitly checked, to ensure it can fit both the selector and the `setOperatorPermissions` arguments. If the size is not large enough, currently, the slice or the `abi.decode` should fail.
11. In `EmissionController.constructor()`, it is not checked that the array of rates is in ascending order, although it is done in `setEmissionsSchedule()`.
12. In `EmissionController.setEmissionsSchedule()`, it is not checked that the new rates to be used are smaller than the current one being used. Emissions are not guaranteed to be decreasing.

Code corrected:

All sanity checks have been added to the functions listed above, except for the following which were ignored for the reasons stated:

8. Resupply states: "Ignored in favor of reducing bytecode size"; 12. Resupply states: "Ignored, not actually a requirement. Updated revert message for clarity."

6.9 Non-ERC20 Compliant Token Can DOS the Reward Token Insertion

CS-RESUPPLY-036

In `RewardDistributorMultiEpoch._insertRewardToken()`, the following is done to invalidate reward tokens that would not be compliant with the ERC20 standard or fail on transfer:

```
try IERC20(_token).transfer(address(this), 0){}catch{
    _invalidateReward(_token);
}
```

If the token is not compliant with the ERC20 standard and does not return a value upon transfer (e.g USDT), the try catch block will not catch the revert as the ABI-decoding would cause the exception in the currently executing contract and not the called contract. This would mean that `RewardHandler.checkNewRewards()` would always fail.

Code corrected:

The try-catch statement with the call to `transfer()` has been replaced with a call to `safeTransfer()`, s.t. failure of non-compliant tokens will be caught. Additionally, the `WriteOff` token implementation has been adjusted to always return true when `transfer()` is called (no-op in any case) to avoid reverting upon reward token insertion.

It should be noted that as revert in the transfer function is no longer caught, the reward token insertion will fail if the token is not compliant with the ERC20 standard and not handled by the `SafeERC20` library (e.g revert on transfer).

6.10 Pair Creation Code Cannot Be Smaller Than 13,000 Bytes

CS-RESUPPLY-005

In the `ResupplyPairDeployer`, logic allows for the pair creation code to be either larger than 13,000 bytes (in which case, it is stored in two contracts), or smaller than or equal to 13,000 bytes (stored in one contract). In the latter case however, the function `setCreationCode()` would always revert as `BytesLib.slice(_creationCode, 0, 13_000)` would fail with a `slice_outOfBounds`

Code corrected:

Calling `BytesLib.slice(_bytes, _start, _length)` does not fail anymore, as the `_creationCode` byte array passed as `_bytes` is always at least as long as the sum of `_start` and `_length`. Specifically, for creation codes of 13,000 bytes or less, the entire code is stored in `contractAddress1` and no slicing is performed.

6.11 Sfrax Cycle Length Can Differ From REWARDS_CYCLE_LENGTH

CS-RESUPPLY-007

In `InterestRateCalculator.sfraxRates()` the following is done:

```
ISTakedFrax.RewardsCycleData memory rdata = ISTakedFrax(sfrax).rewardsCycleData();
uint256 sfraxtotal = ISTakedFrax(sfrax).storedTotalAssets();
uint256 maxsfraxDistro = ISTakedFrax(sfrax).maxDistributionPerSecondPerAsset();
fraxPerSecond = rdata.rewardCycleAmount / REWARDS_CYCLE_LENGTH;
```

However, it could be that `rdata.cycleEnd - rdata.lastSync` differs from `REWARDS_CYCLE_LENGTH`. In this case, the computed `fraxPerSecond` would be inaccurate.

Code corrected:

The length of the rewards cycle is now correctly calculated as `rdata.cycleEnd - rdata.lastSync`.

6.12 Unprotected Functions Can Be Reentered

CS-RESUPPLY-038

Most of the user facing functions in the `InsurancePool` and `Pair` contract are reentrancy protected. However, the following functions are not:

- `ResupplyPairCore.userCollateralBalance()`
- `ResupplyPairCore.earned()`
- `InsurancePool.exit()`
- `InsurancePool.cancelExit()`
- `InsurancePool.earned()`

Given the current trust model, no security issues were found, but for the sake of completeness, these functions should be reentrancy protected. In case a reward token would have hooks on transfer, which is outside of the trust model of this review, this could be a security issue as one could reenter the contract in the middle of a checkpoint. This would allow an attacker to get more rewards than they should, making the reward system insolvent.

Code corrected:

The code has been adjusted accordingly and the above functions are now reentrancy protected.

6.13 Unsafe Approvals

CS-RESUPPLY-008

In several occurrences, approval is given to addresses using `approve()`. Tokens that do not conform to the ERC20 standard like USDT would have this call fail as they do not return a boolean value.

- `Swapper.addPairing()`
 - `ResupplyPair.constructor()`
 - `ResupplyPairCore.constructor()`
-

Code corrected:

All occurrences of `approve()` have been replaced with `forceApprove()`.

6.14 `Swapper.swap()` Always Returns 0

CS-RESUPPLY-009

`Swapper.swap()` does not set its return value `amountOut` and will always return 0, no matter the amount sent to `to`.

Code corrected:

The function `swap()` do not return a value anymore.

6.15 `Swapper` Does Not Check Slippage

CS-RESUPPLY-010

In `Swapper.swap()`, `amountOutMin` can be provided as an argument, however, it is not used in the function and no slippage check is performed. If the function is called by a pair to (de)leverage, the check would be performed in the pair, but not for other usages.

Code corrected:

The `amountOutMin` parameter was removed from the function signature to make it clear that no slippage check is performed in the function. If a slippage check is required, it should be performed by the caller.

6.16 `allocationByType` Is Incorrect

CS-RESUPPLY-011

In `VestManager.setInitializationParams()`, `allocationByType` is assigned at each iteration with:

```
allocationByType[allocType] = allocation;
```

Given that both Sub-DAO vests share the same allocation type, this means that the second Sub-DAO will overwrite the first Sub-DAO's allocation. The mapping will be incorrectly reporting the allocation of the type.

Code corrected:

The allocation by type is now incremented instead of being overwritten.

6.17 getMaxRedeemableDebt() Can Be Inaccurate

CS-RESUPPLY-039

The function `getMaxRedeemableDebt()` is a view function that returns the maximum redeemable debt against a given pair in the `RedemptionHandler`. Essentially giving an upper bound on the amount that can be passed to `redeemFromPair()`. The returned value might not be accurate given that:

- If it is below `minimumRedemption` (minimum redemption amount) the redemption would revert.
 - In general, a slightly higher amount of debt can be redeemed than the returned value given that the amount passed to `redeemFromPair()` is not directly removed from the debt. Instead, the protocol fee is calculated from this amount but not deducted from the debt.
-

Specification changed:

The specification of the function has been changed to return how much debt the pair can have redeemed. The NatSpec has been updated accordingly and now states:

```
/// @notice Estimates the maximum amount of debt that can be redeemed from a pair
```

6.18 getReceiverId() Result Can Be Misleading

CS-RESUPPLY-012

In the `SimpleReceiverFactory`, the function `getReceiverId()` can return 0 as receivers are registered in the `EmissionsController` starting with ID 0. In this case it is unclear if the given address is not registered, or if it is registered but has the ID 0.

Code corrected:

The function `getReceiverId()` now ensures that the provided receiver is indeed the address of a registered receiver if the ID is 0.

6.19 totalDebtAvailable() Does Not Account for Interests



In the `ResupplyPairCore`, `_totalDebtAvailable()` does not use its parameter `_totalBorrow` but rather reads `totalBorrow` from storage. This means that `totalDebtAvailable()` can return an incorrect result as it does not account for interests.

Code corrected:

The code has been corrected accordingly and now computes the amount of borrowable assets based on the `_totalBorrow` parameter passed to the function.

6.20 Approval Not Removed in Swapper

Using `addPairing`, a `swappool` can be added for a pair of tokens, effectively overriding the previous `swappool` for the pair. In such cases, approval is not removed from the previous `swappool`.

Code corrected:

In `Swapper.addPairing()`, the approval is removed from the previous `swappool` before adding the new one.

6.21 Arbitrary Call Targets

In several instances in the system, functions callable by anyone make calls with a given function selector to arbitrary targets. In case of a function selector collision, this could be used to call unexpected functions of the system on behalf of the contract. This could lead to unexpected behavior and potentially to security issues.

- `ResupplyRegistry.claimFees()`
- `LiquidationHandler.liquidate()`

An example could be a collateral token that defines a function `liquidate(uint256)` (or any other function with the same selector), Anyone could call `LiquidationHandler.liquidate(collateral, account)` and the liquidation handler would forward the call to the collateral token, which could lead to unexpected behavior depending on the token implementation.

Code corrected:

Both functions have been updated to only allow pairs registered in the registry to be passed as arguments.

6.22 Code Readability

CS-RESUPPLY-041

The following is a non-exhaustive list of potential code readability improvements that were found in the codebase:

1. In `EmissionController.registerReceiver()`, `10_000` could be replaced by `BPS`.
 2. In `InterestRateCalculator.sfraxRates()` the following is commented `address pricefeed = IStakedFrax(sfrax).priceFeedVault();`. It should be removed to avoid confusion.
 3. In `GovStaker._stake()`, the cast `uint(_amount)` is not needed.
 4. In `RedemptionHandler` a "TODO" comment is present: "TODO: add settable contract for upgradeable logic". It should be removed or expanded.
 5. In `ResupplyPairCore._addCollateral()`, the following comment could be removed `// totalCollateral += _collateralAmount;`.
-

Code corrected:

All the issues mentioned above have been corrected in the codebase.

6.23 Conditions Always False

CS-RESUPPLY-042

- In `InsurancePool._isRewardManager()`, the condition `msg.sender == registry` will always be false as the registry has no way to call the pool.
 - In `RewardHandler.queueInsuranceRewards()` and `RewardHandler.queueStakingRewards()`, `msg.sender == feeDeposit` will always be false as the fee deposit has no way to call the reward handler.
-

Code corrected:

The conditions have been changed accordingly.

6.24 Double Pausing a Pair

CS-RESUPPLY-016

In the `ResupplyPair`, calling `pause()` twice in a row means that calling `unpause()` will have no effect as `previousBorrowLimit` would be 0. In such a case, the pair could be unpaused by calling `setBorrowLimit()`.

Code corrected:



The `pause()` function has been updated and is now a no-op if the `borrowLimit` is already 0. Hence, a second call to `pause()` does not set the `previousBorrowLimit` to 0.

Additionally, the `unpause()` function has been updated to set the `borrowLimit` to the `previousBorrowLimit` only if the `borrowLimit` is 0.

6.25 Empty Sfrax Vault Edge Case

CS-RESUPPLY-029

In the unlikely case where `sfrax.storedTotalAssets()` would return 0, `InterestRateCalculator.sfraxRates()` would revert due to a division by 0.

Code corrected:

If `sfraxtotal` is 0, the function now uses `sfraxtotal = 1` to avoid division by 0.

6.26 Event Field Indexing

CS-RESUPPLY-044

To help with looking for event emissions, the following events could be indexed:

1. `AddPair` in `ResupplyRegistry`.
 2. `RewardInvalidated` in `RewardDistributorMultiEpoch`.
-

Code corrected:

Both events have been indexed.

6.27 Function Mutability

CS-RESUPPLY-045

The following functions could be marked as view:

- `ResupplyPairCore._isSolvent()`
- `InsurancePool._checkWithdrawReady()`
- `InsurancePool.maxWithdraw()`
- `InsurancePool.maxRedeem()`

The following functions could be marked as pure:

- `ResupplyRegistry.getMaxMintable()`
 - `InsurancePool.maxRewards()`
-

Code corrected:

Functions listed above were either removed from the codebase, or their mutability was changed to the most restrictive one.

6.28 Function Visibility Too Broad

CS-RESUPPLY-046

The following functions are public, although they could be marked as external since they are not used internally.

- In `Core`, `setOperatorPermissions`.
- In `SimpleReceiverFactory`, `getDeterministicAddress()`.
- In `GovStaker`, `isPermaStaker()`, `isCooldownEnabled()`
- In `MultiRewardsDistributor`, `earnedMulti()`.
- In `InsurancePool`, `redeem()`, `withdraw()`.
- In `RedemptionHandler`, `getMaxRedeemableDebt()`.
- In `ResupplyRegistry`, `getAddress()`, `getAllKeys()` and `getAllAddresses()`.

Across the codebase, several contracts that are not inherited from by other contracts have functions that are internal, they could be set to private. This includes but is not limited to the `Voter`, the `EmissionController`, the `GovStaker`, the `PermaStaker`, the `InsurancePool` and the `SimpleRewardStreamer`.

Code corrected:

All public functions that were not used internally are now marked as external or being used internally. The functions that were internal and not inherited from where kept as internal.

6.29 Incorrect Documentation

CS-RESUPPLY-048

Below is a non-exhaustive list of inaccurate code documentation:

1. In `InitVestManager.s.sol`, the comments for the `_allocPercentages` does not match the actual values and do not sum up to 100%.
2. In `ResupplyPair.SetOracleInfo`, the `NatSpec` refers to some max deviation, but there is no max deviation used in the contract.
3. In `ResupplyPairCore.Repay`, the `NatSpec` refers to `RepayAsset` instead of `Repay`.
4. In `ResupplyPairCore.repay()`, the `NatSpec` refers to `repayAsset` instead of `repay`.
5. In `ResupplyPairCore.repay()`, the `NatSpec` indicates that `_amountToRepay` was transferred in order to repay the Borrow Shares when it was actually burned.
6. In `GovStaker.cooldown()`, the `NatSpec` indicates that "During partial unstake, this will always remove from the least-weighted first.", this not the case as there is no notion of weight in the staking system.

7. NatSpecs of `_deploy()` and `deploy()` in `ResupplyPairDeployer` are not consistent with the actual fields.
8. In `MultiRewardDistributor.getRewardForDuration()`, the documentation indicates that the "Total reward token remaining to be paid out." is returned, but the function actually returns the total rewards to be paid out for the duration.
9. In `ResupplyPairCore.repayWithCollateral()` the following comment is misleading as `_repay()` never transfer funds:

```
// Note: setting _payer to address(this) means no actual transfer will occur. Contract already has funds
_repay(_totalBorrow, _amountOut.toUint128(), _sharesToRepay.toUint128(), address(this), msg.sender);
```

10. In `VestManager.setInitializationParams()`, the NatSpec for `_allocPercentages` is misleading, the last value of the array is not expected to be the total percentage allocated for emissions.

Code corrected:

All the issues have been addressed.

6.30 LTV Precision

CS-RESUPPLY-051

In `ResupplyPairCore._isSolvent()`, the borrower's LTV could be computed more precisely by first multiplying `(_borrowerAmount * _exchangeRate)` by `LTV_PRECISION` before dividing it by `EXCHANGE_PRECISION` instead of the other way around.

Code corrected:

The order of operations has been changed to first multiply by `LTV_PRECISION` and then divide by `EXCHANGE_PRECISION`.

6.31 Missing Events

CS-RESUPPLY-053

The following missing events were found in the codebase:

1. In `ResupplyPairDeployer.constructor()`, `SetOperator` is not emitted for the Core.
2. `WriteOffToken` does not emit any event upon minting tokens.
3. In `GovStaker.constructor()`, `CooldownEpochsUpdated` is not emitted.
4. `InsurancePool.cancelExit()` does not emit any events.
5. In `RewardDistributorMultiEpoch._insertRewardToken()`, no event is emitted when reviving a reward token.
6. In `ResupplyPairCore.constructor()`, the following events are not emitted: `SetOracleInfo`, `SetMaxLTV`, `SetRateCalculator`, `SetLiquidationFees`, `SetMintFees`, `SetBorrowLimit`.

7. In `ResupplyPair.constructor()`, no event is emitted when setting the `convexBooster` and `convexPid`.
 8. In `GovStakerEscrow`, no event is emitted upon withdrawal of tokens.
-

Code corrected:

All findings have been addressed except for 2., 6. and 8. Resupply added that these were intentional design choices and will not be changed.

6.32 Missing or Incomplete NatSpec

CS-RESUPPLY-055

In multiple occurrences, the NatSpec comments are missing or incomplete. It is recommended to provide a detailed description of the functions and variables in the codebase to improve readability and maintainability.

The following shows a non-exhaustive list of functions where NatSpec is written but incomplete:

- `EmissionController.setEmissionsSchedule()`: the `_tailRate` param is missing.
 - `GovStaker.constructor()`: the `_registry` param is missing.
 - `InsurancePool.mint()`: the return is missing.
 - `InsurancePool.deposit()`: the return is missing.
 - `InsurancePool.redeem()`: the `_owner` param and return are missing.
 - `InsurancePool.withdraw()`: the `_owner` param and return are missing.
 - `InterestRateCalculator.constructor()`: the `_minimumRate` and `_rateRatio` params are missing.
 - `InterestRateCalculator.getNewRate()`: the `_vault` and `_previousShares` params and part of the return are missing.
 - `ResupplyPair.constructor()`: the `_core` param is missing.
 - `ResupplyPair.setRateCalculator()`: the `_updateInterest` param is missing.
 - `ResupplyPairCore.constructor()`: the `_core` param is missing.
 - `ResupplyPairCore._totalDebtAvailable()`: the `_totalBorrow` param is missing.
 - `ResupplyPairCore.addInterest()`: the `_returnAccounting` param is missing.
 - `ResupplyPairCore.Borrow`: the `_mintFees` param is missing.
 - `ResupplyPairCore.redeemCollateral()`: the `_caller` param is missing.
 - `ResupplyPairCore.Liquidate`: the `_sharesLiquidated` and `_amountLiquidatorToRepay` params are missing.
 - As of `MultiRewardsDistributor.getOneReward()`: the param `_account` is missing.
-

Code corrected:

All the listed inconsistencies were corrected in the codebase.

6.33 Order of Events When Adding Reward Token

CS-RESUPPLY-056

In `RewardDistributorMultiEpoch`, the function `_insertRewardToken()` might emit events in the following order if the reward token is invalidated at insertion:

1. `RewardInvalidated`
2. `RewardAdded`

This order could be misleading to event listeners, as the reward token is already invalidated before it is added.

Code corrected:

The order of events has been corrected.

6.34 Protected Keys Should Be Constants

CS-RESUPPLY-018

In the `ResupplyRegistry`, all protected keys should be constants. This would avoid any potential typo that could lead to a security issue.

Code corrected:

The `ResupplyRegistry` now defines constants for all protected key strings and uses them in the code to set the respective addresses.

6.35 Selector Taken From the Wrong Interface

CS-RESUPPLY-020

In `Voter._containsProposalCancelerPayload()`, it is checked that:

```
permissionSelector == ICore.cancelProposal.selector
```

However, this check intends to check if the `permissionSelector` is the selector of the `Voter`'s `cancelProposal` function. Although the current behaviour is correct, the `Core` does not define a `cancelProposal`, and the signature of the function in the voter might diverge from the `ICore` interface in the future.

Code corrected:

The selector is now obtained directly from `this`.

6.36 Solidity Version Not Fixed

CS-RESUPPLY-021

The Solidity version is not fixed in the contracts or in the `foundry.toml` file. This could lead to issues if the contracts are compiled with a different version of Solidity than the one they were developed with. For that reason, it is recommended to fix the Solidity version for the project.

Code corrected:

The Solidity version was fixed to `0.8.28` in the contracts.

6.37 Unreachable Code

CS-RESUPPLY-022

In the Treasury, the `FailedETHSend` event will never be emitted given that the execution will revert if the `success` is false.

```
if(!success) {emit FailedETHSend(returnData);}
require(success, "Sending ETH failed");
```

Code corrected:

The unreachable code has been removed accordingly.

6.38 Unused Errors

CS-RESUPPLY-061

The following errors are defined but never used in the codebase:

- In `GovStaker`: `InvalidEpoch`.
 - In `ResupplyRegistry`: `CircuitBreakerOnly`.
 - In `ResupplyPairConstants`: `SetterRevoked`.
-

Code corrected:

All unused errors were removed from the codebase.

6.39 Unused Events

CS-RESUPPLY-062



The following events are defined in the codebase but are currently unused:

- In Voter: OperatorExecuted.
 - In PermaStaker: StakerMigrated and UnstakingAllowed.
 - In ResupplyPairCore: WarnOracleData.
-

Code corrected:

All unused events have been removed from the codebase.

6.40 Unused Imports

CS-RESUPPLY-063

The following unused imports were found in the codebase:

- In Voter: GovStaker.
 - In MultiRewardsDistributor: ICore.
 - In VestManager: CoreOwnable.
 - In Swapper: SafeERC20.
 - In FeeDepositController: IERC20Metadata.
 - In ResupplyPairDeployer: IResupplyPair.
 - In ResupplyPairDeployer: IFraxlendWhitelist.
 - In SimpleRewardStreamer: ERC20.
 - In SimpleRewardStreamer: ReentrancyGuard.
 - In InsurancePool: IERC4626.
 - In Rewardhandler: IRewardHandler.
 - In Stablecoin: IERC20.
 - In ResupplyPairCore and InsurancePool: ReentrancyGuard (already inherited from RewardDistributorMultiEpoch)
-

Code corrected:

All unused imports were either removed from the codebase or used.

6.41 Unused Parameters

CS-RESUPPLY-065

The following functions have unused parameters:

1. VestManagerBase.claimWithCallback(), _recipient is not used.
2. WriteOffToken.balanceOf(), account is not used and could be commented out (as the parameter should be kept for ERC20 compatibility).

3. `Swapper.swap()`, `account`, `amountIn` and `amountOutMin` are not used.
 4. `LiquidationHandler.processLiquidationDebt()`, `_collateralAmount` is unused.
-

Code corrected:

1. Removed.
2. Commented out.
3. Partially changed, keeping `account` and `amountIn` for future proofing.
4. No change for future proofing.

6.42 Unused Struct

CS-RESUPPLY-064

In the `Core` the struct `Action` is unused.

Code corrected:

The struct `Action` was removed from the contract.

6.43 Variable Shadowing

CS-RESUPPLY-024

Across the codebase, several variable shadowings are present. Although they are not necessarily a problem, they can lead to confusion and should be avoided.

The following variable shadowings were found in the codebase:

- | | |
|---|---------|
| • <code>InsurancePool._checkWithdrawReady(address).withdrawQueue</code>
<code>InsurancePool.withdrawQueue</code> | shadows |
| • <code>ResupplyPair.getUserSnapshot(address)._userBorrowShares</code>
<code>ResupplyPairCore._userBorrowShares</code> | shadows |
| • <code>ResupplyPair.getUserSnapshot(address)._userCollateralBalance</code>
<code>ResupplyPairCore._userCollateralBalance</code> | shadows |
| • <code>ResupplyPair._updateConvexPool(uint256).rewards</code>
<code>RewardDistributorMultiEpoch.rewards</code> | shadows |
| • <code>ResupplyPair._unstakeUnderlying(uint256).rewards</code>
<code>RewardDistributorMultiEpoch.rewards</code> | shadows |
| • <code>ResupplyPair.totalCollateral().rewards</code>
<code>RewardDistributorMultiEpoch.rewards</code> | shadows |
| • <code>ResupplyPairCore.liquidate(address)._userCollateralBalance</code>
<code>ResupplyPairCore._userCollateralBalance</code> | shadows |
-

Code corrected:

All the above mentioned shadowing variables have been renamed to avoid confusion.

6.44 Vesting Duration for Sub-DAOs

CS-RESUPPLY-025

In `VestManager.setInitializationParams`, the following is done:

```
durationByType[allocType] = uint32(_vestDurations[i]);
```

Given that both Sub-DAO vests share the same allocation type, this means that both Sub-DAOs should have the same vesting duration, which is not enforced at the contract level.

Code corrected:

The vest duration of the two Sub-DAOs is now required to be the same.

6.45 SimpleReceiver Implementation Can Be Initialized

CS-RESUPPLY-026

The implementation of the `SimpleReceiver` class can be initialized by anyone by calling `initialize()`. While it is not a security issue at the moment since the contract does not contain `delegatecalls` or `selfdestruct` instructions, it is a good practice to prevent the initialization of the implementation in the constructor.

Code corrected:

A state variable `initialized` is set to `true` in the constructor, and the `initialize()` function is modified to check if the contract has already been initialized. If it has, the function will revert.

6.46 getUnstakableAmount Does Not Account for cooldownEpochs

CS-RESUPPLY-027

In `GovStaker`, `getUnstakableAmount()` returns 0 if `block.timestamp < userCooldown.end`. However, if the user started a cooldown period, and afterwards `cooldownEpochs` was set to 0, they could immediately unstake their tokens. This is not reflected in the view function's logic.

Code corrected:



If `cooldownEpochs` is set to 0, the function will return the full amount of tokens staked by the user, regardless of the cooldown period.

6.47 `previewAddInterest()` Does Not Set `lastTimestamp`

CS-RESUPPLY-066

In `ResupplyPairCore`, `previewAddInterest()` does not set the `lastTimestamp` field in the returned `CurrentRateInfo` struct.

Code corrected:

The `lastTimestamp` field is now set.

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 Convex Pool Might Have Wrong Token

CS-RESUPPLY-015

In `ResupplyPair._updateConvexPool()`, it is not checked that the new `pid` corresponds to a convex pool whose token is matching the collateral of the pair. In general `ICurveStaking(convexBooster).deposit(_pid, stakedBalance, true)` would however fail if the `stakedBalance > 0` as the pair did not give approval to the booster for the pool token. However, if `stakedBalance == 0`, the deposit would be successful and the pair would have a convex `pid` that does not correspond to its collateral token.

Acknowledged:

Resupply acknowledged this behaviour.

7.2 Curve Pool Interface

CS-RESUPPLY-043

The Swapper uses the following interface to call curve pools:

```
interface ICurveExchange {
    function exchange(
        int128 i,
        int128 j,
        uint256 dx,
        uint256 min_dy,
        address receiver
    ) external returns (uint256);
}
```

However, this interface is not shared by all curve pools, for example:

- Some pools define `i` and `j` as `uint256` instead of `int128`.
 - Some pools define an additional parameter `use_eth` to specify whether the swap should be done with ETH or WETH.
-

Acknowledged:

Resupply answered that if other pool interfaces are used, the Swapper will be updated to support them.

7.3 Event Reentrancy

CS-RESUPPLY-030

In `VestManagerBase.claimWithCallback()`, if the callback reenters the contract, the events would be emitted out of order.

Acknowledged

Resupply acknowledges the finding.

7.4 Function Selector Collision

CS-RESUPPLY-017

The `Core` contract implements access control based on function selectors. It is possible that two function selectors collide, which would allow a user to call a function that they should not have access to. Special care should be taken to ensure that function selectors are unique across the different contracts in the system.

Acknowledged:

Resupply acknowledged the behavior and answered:

We should maintain an SDK for the protocol which, among other things, aids in analyzing and crafting new governance proposals. It should include scripts to check proposals for dangerous payloads - including adding new contracts to the registry which introduce a selector collision.

7.5 Gas Savings

CS-RESUPPLY-047

The following potential gas savings were found in the codebase:

ResupplyPairCore:

1. In `_totalDebtAvailable()`, `borrowLimit` is read twice from storage, it could be read once and stored in a local variable.
2. In `currentUtilization()`, `borrowLimit` is read twice from storage, it could be read once and stored in a local variable.
3. In `_isSolvent()`, `maxLTV` is read twice from storage, it could be read once and stored in a local variable.
4. In `isSolvent()`, `exchangeRateInfo` is read three times from storage and the cached value `_exchangeRateInfo` is not used.
5. In `_syncUserRedemptions()`, `_userCollateralBalance[_account]` is read twice from storage, it could be cached.
6. In `_repay()`, `_userBorrowShares[_borrower]` is read three times from storage, it could be cached.

7. In `repayWithCollateral()`, the call to `_removeCollateral()` could directly be passed the `_swapperAddress` as `_receiver` instead of later transferring the collateral.
8. In `repayWithCollateral()`, `_userBorrowShares` is read twice from storage, it could be cached.

ResupplyPair:

1. In `getUserSnapshot()`, the calls to `userBorrowShares()` and `userCollateralBalance()` could be swapped, this would make `userBorrowShares()` more gas efficient as the user would already be up-to-date with the latest epoch.
2. In `_updateConvexPool()`, `_stakeUnderlying()`, `_unstakeUnderlying()` and `totalCollateral()`, `convexPid` is read twice from storage, it could be read once and stored in a local variable.

RewardDistributorMultiEpoch:

1. In `_calcRewardIntegral()`, `reward.reward_token` can be read up to 7 times from storage; it could be cached.
2. Similarly, `reward.reward_remaining` can be read up to 4 times and could also be cached.
3. In `_calcRewardIntegral()`, if `address(0)` is passed as `_account`, its user integral will be updated, although it is not necessary. This could be avoided by checking if `_account` is not `address(0)`.
4. In `_increaseRewardEpoch()`, `currentRewardEpoch` is read twice from storage, it could be read once and cached.
5. In `_checkpoint()`, `rewards.length` could be read once outside the loop instead of at each iteration.
6. In `getReward()`, `rewardRedirect[_account]` is read twice from storage if it is set, this could be avoided.

RewardHandler:

1. Both `queueInsuranceRewards()` and `queueStakingRewards()` call their respective simple receiver 3 times, which itself calls the `EmissionsController` 5 times during these calls. The flow could be optimized to reduce the number of calls.

SimpleRewardStreamer:

1. In `updateReward()`, `rewardPerTokenStored` is written to storage and immediately read after, it could be cached.
2. In `_setWeight()`, `_balances[_account]` is read twice, and could be cached.
3. The external function with no parameters `getReward()` calls `updateReward()` twice for the message sender; it could be optimized to only call it once.
4. The public function `getReward()` reads the reward for the account with `earned(_account)`, however, as the reward is already updated in `updateReward()`, it would be enough to read it from `rewards[_account]`. The same applies to the external function with parameters `getReward()`.
5. In `getReward()`, `rewardRedirect[_account]` is read twice from storage if it is set, this could be avoided.

ResupplyPairDeployer:

1. In `_deploy()`, `contractAddress2` is read twice from storage, it could be read once and stored in a local variable.

ResupplyRegistry:

1. In `getAllAddresses()`, `keys.length` is read at each iteration of the loop, it could be cached.

RedemptionHandler:

1. In `redeemFromPair()`, if `_redeemToUnderlying == False`, `_receiver` could be directly passed to `redeemCollateral()` to remove one token transfer.

BasicOracleVault:

1. `oracleType` could be constant.

FeeDeposit:

1. In `distributeFees()`, `operator` is read twice from storage, it could be read once and stored in a local variable.

FeeDepositController:

1. In the constructor and in `setSplits()`, the insurance, treasury, and platform attributes of the split are written to storage before being read again; storage reads could be avoided. Moreover, it is cheaper to write the entire struct at once as all fields are packed in one slot.

LiquidationHandler:

1. In `distributeCollateralAndClearDebt()`, `debtByCollateral[_collateral]` is read three times to storage, the last one in the event will always read 0. The variable could be cached.
2. In `processCollateral()`, `debtByCollateral[_collateral]` is read three times from storage. Its value could be cached.

InsurancePool:

1. `maxBurnableAssets()` loads `minimumHeldAssets` twice from storage, it could be cached.
2. `burnAssets()` loads `_totalSupply` twice from storage, it could be cached.
3. `_clearWithdrawQueue()` loads `reward.reward_token` twice from storage, it could be cached.

Voter:

1. In `executeProposal()`, `payload.length` is read twice from storage, it could be read once and stored in a local variable.
2. In both functions `voteForProposal`, the modifier `callerOrDelegated` is called twice as it is also called in `_voteForProposal`.

GovToken:

1. In the constructor `globalSupply += _initialSupply` could be replaced with `globalSupply = _initialSupply` as it is the first assignment to `globalSupply`.

GovStaker:

1. In `_checkpointTotal()`, `totalPending` can be cached after the early return.
2. In `_unstake()`, `cooldowns[_account]` could be loaded to memory once to avoid one storage load since all fields fit in one word.

VestManager:

1. In `setInitializationParams`, the storage `redemptionRatio` is written to before being read, it could be cached to avoid the storage read.

VestManagerBase:

1. `token` could be immutable.

MultiRewardsDistributor:



1. In `rewardPerToken()`, `_totalSupply` is read twice, it could be read once and stored in a local variable.
2. `_getRewardFor()` reads the length of `rewardTokens` at each iteration from storage it could be read once and stored in a local variable.
3. `setRewardsDuration()` load the entire `Reward` struct to storage but only reads two of its fields, it could be optimized to only read the necessary fields.
4. In the `updateReward` modifier, if a non-zero account was provided, `rewardData[token].rewardPerTokenStored` is written to storage and read immediately after, it could be cached to avoid the storage read.
5. The `updateReward` modifier reads the length of `rewardTokens` at each iteration from storage it could be read once and stored in a local variable.
6. As of , `_getRewardFor()` and `getOneReward()` read `rewardRedirect[_account]` twice from storage, moreover, `getOneReward()` reads it at every iteration of the loop, it could be cached.

EmissionController:

1. In `setReceiverWeights()`, duplicates are checked with:

```
address[] memory receivers = new address[](_receiverIds.length);
for (uint256 i; i < _receiverIds.length; ++i) {
    ...
    for (uint256 j; j < receivers.length; ++j) {
        require(receivers[j] != receiver.receiver, "Duplicate receiver id");
    }
    receivers[i] = receiver.receiver;
    ...
}
```

This could be optimized by having the inner loop stops at `i` as `receivers` is written to in order.

2. In `_calcEmissionsForEpoch()`, `tailRate` is read twice from storage, it could be read once and stored in a local variable.
3. `isRegisteredReceiver()` could return `True` instead of `id != 0` as if `id == 0`, the function returns earlier.

SimpleReceiverFactory:

1. In `deployNewReceiver()`, `implementation` is read from storage twice, it could be read once and stored in a local variable.

Code partially corrected:

All gas savings were implemented except for:

- **RewardHandler:** 1.
- **Voter:** 1.
- **GovStaker:** 1. and 2.
- **EmissionController:** 2.
- **LiquidationHandler:** 1. (the event still reads from storage a variable that is always 0), 2. (one storage read could still be avoided).
- **FeeDepositController:** 1. (only partially implemented).

7.6 Incorrect Interfaces

CS-RESUPPLY-049

- The interface `IGovToken` contains functions that are not present in the implementation `GovToken`.
 - As of `1.0.0`, `getProposalData()` and `createNewProposal()` signatures in `IVoter` do not match the implementation in the `Voter`.
 - As of `1.0.0`, `getOneReward()` and `RewardPaid` signatures in `IGovStaker` does not match the implementation in the `MultiRewardsDistributor`.
-

Code partially corrected:

The interface `IGovToken` now matches the implementation `GovToken`.

7.7 Insurance Pool Is Not Compliant With ERC-4626 Standard

CS-RESUPPLY-050

The `InsurancePool` is not compliant with the ERC-4626 standard:

- `previewRedeem()` and `previewWithdraw()` do not take into account the exit window.
 - Several ERC20 functions such as `transfer()`, `transferFrom()`, `name()`, `symbol()`, `approve()` or `allowance()` are not implemented.
-

Acknowledged:

Resupply added that the pool is not meant to be ERC-4626 compliant nor to be a transferable token.

7.8 Locked Tokens

CS-RESUPPLY-052

Various ERC20 tokens could be accidentally sent to the contracts of the codebase. If no recovery function is implemented, the tokens will be locked, with no way to recover them. Incidents in the past showed this is a real issue as there always will be users sending tokens to contracts not designed to receive them.

Acknowledged:

Resupply acknowledges the finding.

7.9 Missing Inheritance

CS-RESUPPLY-054

- GovToken does not inherit from IGovToken.
 - SimpleReceiver does not inherit from ISimpleReceiver.
 - GovStakerEscrow does not inherit from IGovStakerEscrow.
 - BasicVaultOracle does not inherit from IOracle.
 - ResupplyRegistry does not inherit from IResupplyRegistry.
 - RewardHandler does not inherit from IRewardHandler.
 - Stablecoin does not inherit from IMintable.
 - Swapper does not inherit from ISwapper.
-

Acknowledged:

Resupply acknowledges the finding.

7.10 Price Difference Between Underlying Tokens

CS-RESUPPLY-057

Across the protocol, the assumption that the price of the underlying tokens is equal to the price of reUSD is made as it is expected that both frxUSD and crvUSD are stable. However, in case there is a price difference between the two tokens, this would incentivize redeeming reUSD in pairs with collateral tied to the more expensive token. As a result, redemptions would consistently target specific pairs.

Acknowledged:

Resupply acknowledges and states:

Will be partially addressed with redemption calculator and decaying redemption fee. Further adjustments can be made in the future with the redemption handler.

7.11 Reentrancy Guard Constructor

CS-RESUPPLY-019

Across the codebase, the ReentrancyGuard library is inherited from in multiple contracts but never called its constructor. With the current implementation of the library, this means that until anyone calls a protected function, 0 will be stored in `_status` instead of `NOT_ENTERED == 1`. Looking at the implementation of `_nonReentrantBefore()`, this does not have any security implications as it is checked that status is equal to `ENTERED`, and not whether it is equal to `NOT_ENTERED`.

Acknowledged:

Resupply acknowledges this finding.

7.12 Refactoring Introduces Accounting Inconsistencies

CS-RESUPPLY-058

When refactoring the shares of the pairs or of the insurance pool, it might be that rounding errors lead to the total amount of shares becoming greater than the sum of the shares of the individual shareholders.

- In the case of the insurance pool, this means that some dead shares are created, and cover some part of the reUSD assets. This is a small loss for other shareholders, but not for the system.
- In the case of the pairs, this means that some of the debt is now covered by some dead shares, in other words, the system. That is, if every shareholder were to repay their debt, the system could still be in debt. Which is considered as "bad" debt.

In the following, we call $assets_0$ the total amount of assets and shares right before refactoring and after the redemption. We call $assets_1$ and $shares_1$ the total amount of assets and shares right after refactoring.

By definition of refactoring, it is known that:

$$shares_1 = shares_0 \cdot 10^{12}$$

$$assets_0 \cdot 10^{12} < shares_0$$

$$assets_0 = assets_1$$

After refactoring, the amount of dead shares, that we will call *loss* can be upper bounded by the sum of the individual user's refactoring rounding errors (1 wei of share), that is $loss_{share} \leq N$, where N is the number of users.

Using the equations above, we can upper bound the loss in terms of assets as:

$$loss_{asset} = loss_{shares} \cdot \frac{assets_1}{share_1} = N \cdot \frac{assets_0 \cdot 10^{12}}{share_0} < N \cdot \frac{assets_0 \cdot 10^{12}}{assets_0 \cdot 10^{12}} = N$$

This demonstrates that the asset loss is limited by the number of users, which is expected to be small relative to the total assets. Therefore, the loss should be minimal.

Despite the small asset loss, it is important to recognize that the system depends on shares for other mechanisms.

1. Writeoff token distribution:

As the sum of user shares is smaller than the total amount of shares, reward distribution will not distribute all the rewards to users, but also some to the dead shares. For regular reward tokens, this is not an issue, as the users would just receive slightly less reward and some dust would be stuck in the system. However, for `writeOff` tokens in the pair, this means that some collateral to be written off might not be accounted for. Given the amount of loss in share shown above, these are dust amounts, but it could be problematic for the last user in the system to withdraw their collateral as described in [Pair might be insolvent for collateral](#) which shows that reward distribution rounding errors can also lead to the system being insolvent for collateral.

2. Reinitialization of the system:

In case every user in the system exits, the system could end up in one of the following three states:

1. `total_shares == total_assets == 0.`
2. `total_shares > 0 && total_assets > 0.`
3. `total_shares > total_assets == 0.`

In the first case, the system is in a state where no shares are minted, and no assets are held. This is a correct state as the first user entering the system would then mint the first shares one to one with the first assets provided.

In the second case, the system has some bad debt, that is, the system is in debt even if all the users repay their individual debt. As shown above, this bad debt should be dust, and new users would not be able to profit from it as they would mint according to the price per share and not one to one.

In the third case, the system is in a state where shares are minted, but no assets are held. This means that all the debt has been repaid, however some "dead" shares created with refactoring remain in the system. When the first user enters the system, the system would mint shares to that user one to one given the special condition (`total.amount == 0`) in `VaultAccountingLibrary.toShares()`. Repaying would however, allow the user to share their debt with the "dead shares" and profit from the system. They would leave the system in a state where `total_shares == total_assets > 0`.

Acknowledged:

Resupply acknowledged the informational note.

7.13 Safe Cast Too Restrictive

CS-RESUPPLY-059

In `EmissionsController`, the function `_safeCastToUint200()` is too restrictive as it reverts when passed `type(uint200).max` although it is a valid value for `uint200`.

Acknowledged:

The function was not modified to keep the gas savings associated with equality check avoidance.

7.14 Sandwiching Voter Proposal Execution

CS-RESUPPLY-060

Governance proposals can be executed in a permissionless manner. This also means that anyone can sandwich such a proposal in order for it to be executed while the system is in an unstable state. While no such attack vector has been observed in the current implementation of the system, it is important to note that this is a possibility and should be considered if the system is upgraded.

Acknowledged:

Resupply acknowledges the finding and decided to ignore it in favor of keeping the system permissionless.

7.15 Unsafe Transfer

CS-RESUPPLY-023

In `VestManager.redeem()`, tokens are transferred with `IERC20(_token).transferFrom(msg.sender, BURN_ADDRESS, _amount)`. As long as the token is compliant with the ERC20 standard (return a boolean and revert on failure), the transfer is correctly handled. Otherwise, the transaction would either always revert if the token does not return a value, or the transfer would be misinterpreted as successful if the token does not revert on failure but instead returns false.

In the current implementation, the token can be PRISMA, yPRISMA or cvxPRISMA, which are all ERC20 compliant, but if new tokens are added, it is important to ensure that the token is compliant with the ERC20 standard.

Acknowledged

Resupply acknowledged this behaviour and answered:

No chance of introducing a non-compliant token as no new tokens can/will be added.

7.16 platform Unused in FeeDepositController Split

CS-RESUPPLY-028

In the `FeeDepositController`, the `platform` part of the split is not needed as it can be recomputed from the `insurance` and `treasury` parts. Moreover, due to rounding error, it might be that `balance - ipAmount - treasuryAmount != balance * _splits.platform / BPS`.

Acknowledged

Resupply acknowledges the finding and adds that `platform` is never used in calculation and is there for information purposes only.

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 Access Control `VestManager`

In the `VestManager`, the function `claimWithCallback()` can be called by trusted third parties. However, to do so, calling `setDelegateApproval()` is not enough. It is also required that `allowPermissionlessClaims` is set to `true` in the account's permissions.

8.2 Non-finalized Minter Can Mint Tokens

The minter of the `GovToken` can be changed as long as it is not finalized using `finalizeMinter()`. It should be noted that even if it is not finalized, the minter can mint tokens.

8.3 Pair Utilization

Although new borrows can be limited when the `borrowLimit` is reached, the pair utilization might still go above 100% due to interest accruing.

8.4 Partial Liquidations Not Supported

All liquidations in `Resupply` cover the outstanding debt of an insolvent position fully. The amount of collateral required to settle the borrowers full debt is transferred to the `LiquidationHandler` and then transferred to the insurance pool and distributed to insurance pool participants. Note that all liquidations must be executed through the `LiquidationHandler` which does not allow for partial liquidations.

8.5 Pending Stake During Migration

When migrating from a `GovStaker` to a new one, perma-stakers can call `OldGovStaker.migrateStake()` to have their realized stake migrated. In the case they had a pending stake at that time, they will need to call `migrateStake()` again in a later epoch by using `PermaStaker.execute(oldStaker.migrateStake())`.

8.6 Redemptions Will Target Specific Pairs

For this review, it is assumed that fraxUSD and crvUSD are pegged to exactly one dollar. In reality, this might not always be the case, and there could be slight deviations. To maximize profit, redemptions will always target pairs whose underlying asset is the more expensive of the two tokens.

8.7 Redemptions of PRISMA

In the `VestManager`, `allocationByType[REDEMPTIONS]` is expected to include the allocation for both:

- Redemptions of PRISMA, YPRISMA and CVXPRISMA.
- Airdrop for vePRISMA lock penalties.

This means that `_maxRedeemable` should be greater than or equal to the circulating supply of PRISMA, YPRISMA, and CVXPRISMA combined, including the vePRISMA past penalties and potential future penalties (balance of the vePRISMA locker).

It is expected that penalties already collected by the locker's fee receiver are accounted for in `_maxRedeemable`, but will be burned and not redeemable as they are accounted for in the penalty airdrop.

`allocationByType[AIRDROP_LOCK_PENALTY]` will only represent a subset of `allocationByType[REDEMPTIONS]`.

Any penalty from breaking a lock after the penalty airdrop merkle root is set using `setLockPenaltyMerkleRoot()` will not be claimable, although it was accounted for in the allocation. This means that the corresponding allocation will be locked in the `VestManager`.

8.8 Reward Claim Is Permissionless

In `SimpleRewardStreamer`, `MultiRewardsDistributor` and `RewardDistributorMultiEpoch`, reward claims are permissionless. Users should be aware that if they cannot handle receiving rewards and did not set a redirect, anyone could claim rewards on their behalf, effectively locking the rewards.

8.9 Underlying Can Be Not Redeemable

In case all underlying have been borrowed in the underlying lending market (Curve lend or Fraxlend), it could be that the collateral is not redeemable at a given moment. In the context of liquidations, this means that the collateral will stay in the liquidation handler, and can be later redeemed once enough underlying is available in the lending vault by calling `processCollateral()`.

8.10 Underlying Stablecoin Depegging

reUSD is not strictly pegged to 1\$ but rather to crvUSD and fraxUSD, which are both assumed to be pegged to 1\$ in the system and for this review. Shall one of the underlying stablecoins depeg, reUSD will also depeg. This is a risk that should be considered when using reUSD as a stablecoin.

8.11 Withdrawing From the InsurancePool Clears All RSUP Reward

When redeeming or withdrawing from the InsurancePool after waiting for the withdrawing period, all accrued RSUP rewards are redistributed to the shareholders. It should be noted that this is the case for any amount of tokens withdrawn. It might hence be more interesting to open multiple positions with different accounts to be able to withdraw small amounts while still maximizing the RSUP rewards.