PUBLIC

Code Assessment

of the sreUSD Smart Contracts

August 19, 2025

Produced for

RESUPPLY

by

CHAINSECURITY

Contents

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 sreUSD according to Scope to support you in forming an opinion on their security risks.

The assessment was performed on a very limited scope that is an addition to the existing Resupply codebase. The main addition is a staking vault (`sreUSD`) and an adopted fee distribution mechanism. `sreUSD` offers cross-chain support via LayerZero and linear reward distribution. The provided commits contain code that was not audited by us. The scope is described in detail in our Scope section.

The team was always professional and available to answer questions.

The most critical subjects covered in our audit are mathematical operations like incorrect reward calculation described in Interest for sreUSD deducted twice or issues with the weight calculation described in Current Weight Might exceed 100%, All issues were addressed and/or resolved accordingly.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	1
• Specification Changed	1
Low-Severity Findings	5
• Code Corrected	3
• Risk Accepted	1
• Acknowledged	1

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	21 Jul 25	9a3de5b4a3ced4eb993c87cc19cf85c99bf3e6a2	Initial Version
2	18 Aug 25	c88c0c8ed90c1121bcf3d183f6d6cbddcb81bc32	After Intermediate Report

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

The scope of this assessment is limited to the changes and additions in:

```
src/protocol/
    sreusd/
        LinearRewardsErc4626.sol
        sreUSD.sol
    protocol/
        PriceWatcher.sol
        InterestRateCalculatorV2.sol
        FeeLogger.sol
        RewardHandler.sol
        FeeDepositController.sol
```

2.1.1 Excluded from scope

Note that for the below contracts, the scope was limited to a diff. More precisely, the diffed commit is `0beb774a9669869e86422ba3fbb5a2054fbb2aaa` which corresponds to the 5th version of our core review. Below is a list of detailed elaborations:

- `InterestRateCalculatorV2.sol`: The scope is limited to the differences with `InterestRateCalculator.sol`. It is assumed that the previous version is correct. The scope consists of only the boosted interest mechanics and potential consequences.
- `RewardHandler.sol`: The scope is limited to the differences with the previous version. It is assumed that the previous version is correct. The scope consists only of the addition of the hooks for the intended purposes.
- `FeeDepositController.sol`: The scope is limited to the differences with the previous version. It is assumed that the previous version is correct. The scope consists only of the addition of the boost computations and relevant parts.

Additionally, the following is out of scope:

- All changes in other files than the ones listed above.

- Library and third party code.
- Deployment scripts and procedures including configuration.
- Behavior of third party protocols and libraries.
- LayerZero is assumed to be correct and working as intended. The OFT on other chains is expected to be correct and sound.
- The reUSD oracle is assumed to be safe, please consider Oracle considerations.
- The only token relevant for the diff is reUSD. Any forks must ensure that tokens are similar to reUSD as otherwise issue might occur (e.g. reentrancies in sreUSD).

2.2 System Overview

This system overview describes the latest received version of the contracts as defined in the Assessment Overview.

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

Resupply implement sreUSD a yield bearing token that aims to stabilize the system in case of depegs. Several contracts have been adjusted, and helper contracts have been introduced to supported boosted interest rates and its distribution. Please consider our previous core audit report for a complete picture of the system.

2.2.1 sreUSD

sreUSD is an ERC-4626 vault that supports LayerZero's cross-chain OFT bridging functionality and a linear streaming of rewards. The rewards are new assets streamed into the vault, thereby increasing the share price.

LinearRewardsErc4626.sol serves as the base contract for sreUSD and implements both the ERC-4626 functionality and the linear reward streaming.

The contract operates in reward cycles (e.g., 6 hours). At the beginning of each cycle, rewards are registered (i.e., any surplus balance not yet accounted for in the total assets) and the linear distribution is configured via `_syncRewards`. This reward synchronization can be simulated using `previewSyncRewards`, which is called internally by `_syncRewards`. These rewards are then distributed linearly, proportional to the time elapsed within the cycle, via `_distributeRewards`.

The `syncRewardsAndDistribution` function first distributes any pending rewards and then synchronizes the reward cycle (i.e., it checks if a new cycle has started and computes the new reward cycle data accordingly). Each state-modifying ERC-4626 entry point (`deposit`, `mint`, `withdraw`, and `redeem`) is overridden to ensure the exchange rate is updated correctly. Thus, `syncRewardsAndDistribution` is called before the parent contract's implementation (`super`).

Similarly, read-only functions must account for this linear increase by simulating the current exchange rate. This is achieved by overriding the `totalAssets` function. As outlined above, the current balance (`underlying.balanceOf(address(this))`) is used to compute the surplus rewards. The `storedAssets` value represents the total assets managed by the vault at the time of the last state-changing operation. Hence, `totalAssets` is the sum of `storedAssets` and the simulated rewards that have accrued since then. These simulated rewards are calculated with `previewDistributeRewards`, which internally uses `calculateRewardsToDistribute` to compute a linear distribution of the current cycle's rewards.

Furthermore, `storedTotalAssets` is increased during deposits (in `afterDeposit`) and decreased during withdrawals (in `beforeWithdraw`).

Finally, for rewards to be distributed fairly, new cycles must be initiated as soon as a previous cycle ends and new rewards are available.

sreUSD.sol implements the actual sreUSD token.

First, the `core` contract is intended to be the owner; this is enforced by having the `owner` function return the `core` contract's address. The owner can only be set once.

Second, it overrides the `LinearRewardsErc4626` contract to introduce a cap that effectively limits the share price growth. This is achieved through a configurable maximum distribution per second per asset, which can be set by the administrator.

Last, it adds LayerZero support via the `token`, `approvalRequired`, `_debit`, and `_credit` functions. The vault itself acts as an escrow for outbound bridged shares. The OFT on other chains is expected to be a standard ERC-20 OFT. If exchange rates are required on other chains, a separate, currently unimplemented, mechanism would be needed.

2.2.2 InterestRateCalculatorV2

The `InterestRateCalculatorV2` is a refactored version of the `InterestRateCalculator` contract. The main change is the implementation of boosted interest rates.

The `InterestRateCalculatorV2` contract is responsible for calculating the interest rate for the pair contracts. Its most important function is `getNewRate`, which is queried by pairs to calculate the interest accrual for the preceding period.

`getNewRate` returns the multiplication of a rate called `rateRatio` with the maximum of the sfrxUSD rate, the interest per second generated by the collateral token and the minimum rate multiplied. `rateRatio` is calculated as the sum of a base share (`rateRatioBase`) and a boosted share (`rateRatioAdditional * priceweight / 1e6`). The boosted share's weight is retrieved by querying `PriceWatcher.findPairPriceWeight`, which provides a time-weighted depeg indicator (see PriceWatcher). Ultimately, the boosted portion of the rate cannot exceed `rateRatioAdditional`.

For additional information, please read Boosted distribution considerations.

2.2.3 FeeDepositController

The assessment's scope was limited to the changes in the `FeeDepositController` contract. The main changes relate to the boosted interest mechanism and include the addition of the `EpochTracker` contract, integration of the new `PriceWatcher` contract, a major adjustment to the `distribute` function, and the addition of the `setAdditionalFeeRatio` function. This last function sets a base ratio that is scaled by the average weight.

The `FeeDepositController` allows anyone to call `distribute` once per epoch to distribute the rewards accumulated in the `FeeDeposit` contract. The function can only be called once per epoch, as the `FeeDeposit` contract is designed to revert on subsequent calls within the same epoch. It is assumed that the operator of the `FeeDeposit` contract is the `FeeDepositController` contract.

The high-level overview of the `distribute` function is as follows:

1. Fees are collected from the `FeeDeposit` contract. These correspond to the fees accumulated two epochs prior (i.e., in `currentEpoch - 2`).
2. The total fees (i.e., the held balance) are logged in the fee logger for `currentEpoch - 2` via `FeeLogger.logTotalFees`.
3. A weight checkpoint is performed by calling `PriceWatcher.updatePriceData`. This new index serves as the final price watcher checkpoint for `currentEpoch - 1` and the first checkpoint for `currentEpoch`.
4. Since the final checkpoint for `currentEpoch - 1` is now created, an average weight for `currentEpoch - 1` is computed. This weight may be used in the next `distribute` call. This

value, along with the final index (which serves as the starting index for `currentEpoch`), is stored in `epochWeighting`.

5. Finally, the fee distribution begins. The process uses the average weight for `currentEpoch - 2` (cached in `epochWeighting` during the previous epoch) to determine the boosting factor. The boost is based only on the interest generated. Therefore, the fee logger is queried with `FeeLogger.epochInterestFees` to report the interest generated for `currentEpoch - 2`. Using this boost and the total interest, the computation described in InterestRateCalculatorV2 is reversed to calculate the boosted interest amount. For additional information, please read Boosted distribution considerations.
6. Last, the remaining fees are split according to the configured allocation among the treasury, sreUSD, the insurance pool, and governance stakers. The rewards are sent to their respective destinations. For the insurance pool and governance stakers, the rewards are routed through the `RewardHandler`.

To summarize, checkpointing for `currentEpoch - 1` is performed, and rewards for `currentEpoch - 2` are distributed by inverting the boosted interest logic, while the remainder is distributed according to the split configuration.

2.2.4 PriceWatcher

The price watcher records weights and depeg indicators (based on the reUSD price) in a `priceData` array. Its data is used by `InterestCalculatorV2` and `FeeDepositController` to calculate the magnitude of depeg-related boosts, which provide extra incentives to sreUSD holders.

`priceData` is updated via `PriceWatcher.updatePriceData`, which can be called by anyone (but is expected to be called by `FeeDepositController.distribute`). The array will only be updated if at least `UPDATE_INTERVAL` (6 hours) has passed since the last update. Thus, the function can be called multiple times per block but will only execute an update when sufficient time has elapsed.

The data pushed to the array includes the update timestamp, the cumulative sum of weights, and the current weight. The cumulative sum of weights corresponds to the previous entry's sum plus the product of the previous entry's weight and the elapsed time delta. The current weight is computed with `getCurrentWeight`.

`getCurrentWeight` returns a reUSD depeg indicator used as a weight, defined as `(1e18 - price) / 1e10`. If the reUSD price is above `1e18`, it will return a weight of zero.

Furthermore, a mapping from a floored timestamp (`timestamp / UPDATE_INTERVAL * UPDATE_INTERVAL`) to an array index is maintained, allowing for efficient lookups of weights based on arbitrary timestamps.

This is leveraged in `findPairPriceWeight`, which calculates the average depeg indicator for a given pair since its latest interest rate update. By using the `timeMap`, a suitable starting index in the array is found, while the latest index in the array serves as the final index for the calculation. Both the start and end values are extrapolated to reflect the pair's last update timestamp and the current timestamp, respectively. The average is computed by dividing the change in the cumulative weight sum by the time delta.

2.2.5 Fee Logger

The fee logger manages the state of fees collected by the `FeeDeposit` contract. The contract has two relevant functions:

`logTotalFees`: Allows the `FeeDepositController` to log the *total fees* distributed in a given epoch in `epochTotalFees`.

`logInterestFees`: Allows the `RewardHandler` contract to log the amount of *interest*:

- `pairEpochWeightings`: Stores the amount of interest collected for a given pair per epoch.

- `epochInterestFees`: Stores the amount of interest collected per epoch.

The `FeeLogger` serves as a helper contract, as interest fees are not distinguished from other fees by other contracts. However, the interest generated per epoch (`epochInterestFees`) is required by the `FeeDepositController` to invert the boosted interest rate computation.

The `RewardHandler` will invoke `logInterestFees`, as it is a suitable place (that does not require modifying the fee deposit or pairs) where interest can still be distinguished from other fees.

`epochTotalFees` and `pairEpochWeightings` are unused in the current implementation, and future use cases were not considered.

2.2.6 Reward Handler

The assessment's scope was limited to the changes in the `RewardHandler` contract. The changes include only the added hooks to the helper contracts and the migration from the old handler.

In general, the `RewardHandler` is responsible for distributing revenue, rewards, and emissions to various parties.

The added hook calls are:

- Call to `FeeLogger.logInterestFees` in `setPairWeight`: This is a suitable place to invoke the hook, as it can separate the interest generated from other fees within the `withdrawFees` execution flow of the pair.
- Call to `PriceWatcher.updatePriceData` in `claimRewards`: This hook is invoked to reduce the burden on bots that are required to update the price watcher. Its addition forces most pool interactions to update the price watcher.

The state migration function `migrateState` allows migrating the state from the old `RewardHandler` contract to the new one. Governance can specify which state it wants to migrate.

2.2.7 Changelog

In (Version 2), the most significant changes are:

- The computation of the interest rate now also multiplies the minimum rate with `rateRatio`. Before, that was not the case.
- The boost within the `FeeDepositController` now defines a `additionalFeeRatio` instead of a `maxAdditionalFeeRatio`. With that also the specification changed, allowing weights to exceed 100%.
- The price watcher now relies only on the reUSD price in crvUSD and is not affected by crvUSD market prices anymore.

2.3 Trust Model

- **Governance**: Fully trusted. Can manipulate the system (e.g., malicious configurations, manipulating the fee logger). In the core, it could whitelist malicious collateral to mint arbitrary tokens. For the items in scope, it could configure malicious delegators for LayerZero or set malicious configurations to allow draining of sreUSD.
- **Governance bots**: Expected to update the `PriceWatcher` when needed.
- **Users**: Untrusted.
- **LayerZero**: Trust depends on the configuration. If the configuration is maintained by other parties, it is fully trusted. Otherwise, LayerZero is only expected to work correctly according to the configuration set by governance (trusted configuration).

- **LayerZero delegator**: The sreUSD delegator is fully trusted. It could drain sreUSD by creating malicious configurations.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 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:

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

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

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

- Infinite Loop in Price Search Risk Accepted
- Withholding Rewards Leads to Unnecessary Growth Slowdown Acknowledged

5.1 Infinite Loop in Price Search

Correctness Low Version 1 Risk Accepted

CS-RESUPPLY-sreUSD-005

`PriceWatcher.findPairPriceWeight` may run into an infinite loop in some cases.

Consider the following code:

```
uint256 ftime = _getTimestampFloor(lastPairUpdate);
uint256 currentIndex = timeMap[ftime];
while(currentIndex == 0){
    ftime -= UPDATE_INTERVAL;
    currentIndex = timeMap[ftime];
```

Assume that `ftime < index1Time` where `index1Time` satisfies `timeMap[index1Time] == 1`. Then, `currentIndex` will remain `0` as no index `>=1` will be found. The loop's condition will always remain true.

Hence, if a pool has had its last update time before the deployment of the price watcher, an infinite loop might occur which could lead to failing interest rate updates and thus failing pair operations.

Risk accepted:

Resupply is aware of the behavior and specified that `setRateCalculator` will only be called with `_updateInterest == true` and only once the price watcher is set up and live.

5.2 Withholding Rewards Leads to Unnecessary Growth Slowdown

Design | Low | Version 1 | Acknowledged

CS-RESUPPLY-sreUSD-007

The limitation mechanism withholds cycle rewards if they exceed the limit. These rewards will not be claimable within the same cycle even if the maximum constraints could allow it. That ultimately leads to followings:

- The growth of the exchange rate is unnecessarily slowed down.
- Distributing rewards as often as possible leads to higher reward limits.

Note that the rewards distributable are a fraction of total reward of the cycle (proportion to the time). The maximum distribution, in addition to the time delta depends on `storedAssets` and `maxDistributionPerSecondPerAsset`. Any change in one of those values, ultimately leads to a change in the maximum which could allow withheld rewards to still be distributable in the same cycle.

Consider the following (simplified) example:

1. Assume that `REWARDS_CYCLE_LENGTH = 1000`, `storedAssets = 1e18` and `maxDistributionPerSecondPerAsset = 1`.
2. The previous cycle ends and a new cycle starts with `rewards = 10000`.
3. After 500 seconds, Alice deposits `99e18-500`. The rewards distributable are `5000`. However, the maximum distribution limits the reward to be `500`. Ultimately, `storedAssets` becomes `100e18`. Note that `4500` fees are undistributed and withheld.
4. After another 500 seconds, the cycle ends and fees are again distributed. The rewards distributable are `5000`. The maximum distribution has no effect as it would be `50000`.

Note that the withheld rewards in step 3 will be distributed in the coming cycle. However, they could have been distributed in step 4 while respecting the maximum distribution constraints. Ultimately, the growth is slowed down unnecessarily.

Note that the following cases exist where withheld rewards could be distributed:

- Increase of `storedAssets`. Note that this can be achieved through deposits or reward distribution. Interestingly, the fact that reward distribution could lead to a sufficient increase of `storedAssets` creates a scenario where the distributing rewards multiple times within a cycle leads to better yields than doing it once.
- Increase of `maxDistributionPerSecondPerAsset` may allow for distributing withheld rewards.

To summarize, withheld rewards are only considered in the next cycle. However, due to changes in values, the withheld amounts could still be considered within the same cycle.

Acknowledged:

Resupply is aware of the behaviour.

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.

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

- Current Weight Might Exceed 100% Specification Changed
- Interest for sreUSD Deducted Twice Code Corrected

Low-Severity Findings	3

- Fee Splits Setting Problems Code Corrected
- No Consideration of Floor Rates Code Corrected
- Price Watcher Weight Affected by crvUSD Depeg Code Corrected

Informational Findings	4

- Gas Optimization Code Corrected
- Interest Rate Calculator Version Code Corrected
- Missing Events Code Corrected
- Non-indexed Events Code Corrected

6.1 Current Weight Might Exceed 100%

Correctness Medium Version 1 Specification Changed

CS-RESUPPLY-sreUSD-001

`FeeDepositController` defines the following:

```
uint256 additionalFeeRatio = maxAdditionalFeeRatio * distroWeight.avgWeighting / 1e6;
```

The naming suggests that `additionalFeeRatio <= maxAdditionalFeeRatio` and that thus `avgWeighting <= 1e6` should hold. If not, 50-100% of the interest could be used as extra incentives for sreUSD which violates the documentation that specifies that up to 50% can be paid out as extra stabilization incentives.

Similarly, the `rateRatio` in `InterestRateCalculatorV2.getNewRate` could be unbound if the weights are not limited to 100%.

However, `PriceWatcher.getCurrentWeight` does not enforce such limits:

```
function getCurrentWeight() public view returns (uint64) {
    uint256 price = IReusdOracle(oracle).price();
    uint256 weight = price > 1e18 ? 0 : 1e18 - price;
    //our oracle has a floor that matches redemption fee
    //e.g. it returns a minimum price of 0.9900 when there is a 1% redemption fee
```

```
    //at this point a price of 0.99000 has a weight of 0.010000 or 1e16
    //reduce precision to 1e6
    return uint64(weight / 1e10);
```

While reUSD is above the peg, the weight will be 0. For values below the peg, the comment suggests that the weight at the floor price should be 100%. However, note that multiple problems exist:

- The floor price is assumed to be always 0.99 (redemption fee is 1%). However, the redemption fee could change, leading to another floor price of for example 0.98. As a consequence, the weight could be 2*1e16, leading to a weight of 200%.
- Additionally, due to `price()` being used, the price could be still below the floor price due to crvUSD being below the peg.

To summarize, the weight is indicated to be at most 100% by the usage of the weight. However, `getCurrentWeight` does not normalize the weight based on the floor price.

Specification Changed:

Weights are now allowed to exceed 100%.

6.2 Interest for sreUSD Deducted Twice

Correctness | Medium | Version 1 | Code Corrected

CS-RESUPPLY-sreUSD-002

`FeeDepositController.deposit` deducts the fee boost for sreUSD twice from the tracked balance which may lead to underflows (resulting in DoS of fee distribution) or too low fee distributions to the `GovStaker`.

Consider the following:

1. The amount to distribute is `x`.
2. Assume that an amount `boost` is computed as a boost for sreUSD.
3. Assume that the sum of the amounts to send to treasury, insurance and sreUSD is `splitAmount <= x - boost`.
4. The total distributed amount corresponds to `splitAmount + boost`.
5. The remainder is computed as follows `(x - boost) - (splitAmount + boost)`.
6. As a consequence, `boost` is deducted twice from `x`.

The consequences of the double deduction are:

- Always: Not enough revenue shared with governance staker (e.g. 100 to distribute, 10 to boost, 40 to others, then 40 instead of 50 would go to stakers).
- If the governance staker share of the payout is smaller than the boost: reverts and DoS of the system (e.g. 100 to distribute, 50 to boost, 50 to others, then -50 is computed which leads to a revert).

Code corrected:

The accounting for `stakedStableAmount` remains to be the same. However, the amount from the split for the staked stable token is tracked additionally in a separate variable `stakedStableSplitAmount` which is used for the delta computation for the governance staker amount. Hence, point 5. above would compute `(x - boost) - splitAmount` now.

6.3 Fee Splits Setting Problems

Correctness | Low | Version 1 | Code Corrected

CS-RESUPPLY-sreUSD-004

Multiple related problems and inconsistencies exist for the fee setting in `FeeDepositController`:

1. `constructor`: Ensures that `_insuranceSplit + _treasurySplit <= BPS`. However, adding `_stakedStableSplit` could exceed `BPS`. That is only implicitly validated as part of the computation of `splits.platform`. If successful, `BPS` is guaranteed to be the sum of splits. However, the error messages are inconsistent.
2. `setSplits`: Ensures that `_insuranceSplit + _treasurySplit + _platformSplit == BPS`. However, `splits.stakedStable` may be set to an arbitrary value. Thus, the sum of splits may overshoot BPS.
3. `setSplits`: Lacks NatSpec for `_stakedStableSplit`.
4. `splits.platform`: The storage variable is unused as `distribute` computes the remainder of fees which it sends to the governance staker.

Code corrected:

The code has been adjusted and most logic has been defined in a new internal function `_setSplits`. While `splits.platform` still exists and still is unused it can be helpful to directly be able to see all percentages sent out.

6.4 No Consideration of Floor Rates

Design | Low | Version 1 | Code Corrected

CS-RESUPPLY-sreUSD-017

The floor for the interest rate, the `minimumRate`, in `InterestRateCalculatorV2` is not considered in the `FeeDepositController`. Boosting may be inaccurate if the floor rates are used.

Code corrected:

Now, the minimum rate is also scaled with `rateRatio` so that the computations can be properly inverted in the `FeeDepositController`.

6.5 Price Watcher Weight Affected by crvUSD Depeg

Design | Low | Version 1 | Code Corrected

CS-RESUPPLY-sreUSD-016

`PriceWatcher.getCurrentWeight` computes the weight based on the reUSD price that is retrieved with `IReusdOracle(oracle).price()`. However, that function relies on Curve's aggregate oracle that may reflect market conditions. Thus, a crvUSD depeg might affect the weights set by the price watcher. As a result, sreUSD deposits will be incentivized which might not be necessary.

Code corrected:

Now, `IReusdOracle(oracle).priceAsCrvusd()` instead of `IReusdOracle(oracle).price()` is used.

6.6 Gas Optimization

Informational | Version 1 | Code Corrected

CS-RESUPPLY-sreUSD-008

1. In `FeeDepositController.distribute` the following calculation is done:

```
uint64 dt = prevWeight.timestamp - prevData.timestamp;
prevData.timestamp = prevData.timestamp + dt;
```

   Hence, `prevData.timestamp` is `prevWeight.timestamp` and must not be calculated.

2. `LinearRewardsErc4626.previewDistributeRewards` could return early for `block.timestamp` equal `_lastRewardsDistribution` as `_deltaTime` would be zero.
3. The state variable `suffix` in `InterestRateCalculatorV2` is only set in the constructor and could be declared immutable.
4. The struct `RewardsCycleData` uses an uncommon `uint216` for the `rewardCycleAmount`. This could be either optimized or kept in two slots. In two slots a `uint256` would save the gas for the type conversions. For one slot it must be evaluated if `rewardCycleAmount` never exceeds the lower type.
5. The state `epochWeighting[currentEpoch - 2]` is loaded redundantly in `FeeDepositController.distribute`. First time it is stored in the memory variable `prevWeight` and then again in `distroWeight`.

Code corrected:

1. Simplified the computation.
2. Early return implemented.
3. Not implemented.
4. Implemented that the amount is `uint256`.
5. Removed `distroWeight`.

Note that most of the suggestions have been implemented and that 3 would have required additional changes. Ultimately, the gas efficiency has been optimized.

6.7 Interest Rate Calculator Version

Informational | Version 1 | Code Corrected

CS-RESUPPLY-sreUSD-018

`InterestRateCalculatorV2.version()` returns `(1,0,0)` which is the same as `InterestRateCalculator.version()`. However, it is the second version and should reflect that.

Code corrected:

`(2,0,0)` is now returned.

6.8 Missing Events

Informational Version 1 Code Corrected

CS-RESUPPLY-sreUSD-010

Events help off-chain applications and users to retrieve data from smart contracts. Missing events might make it harder to retrieve information. Below is a non-exhaustive list of missing event emissions:

- `PriceWatcher.constructor`: Does not emit the `OracleSet` event as `PriceWatcher.setOracle` does.
- `SavingsReUSD.constructor`: Does not emit the `SetMaxDistributionPerSecondPerAsset` event as `SavingsReUSD.setMaxDistributionPerSecondPerAsset` does.

Code corrected:

The listed events are now emitted.

6.9 Non-indexed Events

Informational Version 1 Code Corrected

CS-RESUPPLY-sreUSD-011

1. In `FeeDeposit` the `SetOperator` event has both address fields not indexed.

Code corrected:

The fields are now indexed.

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 Donation Before Deployment

Informational **Version 1** **Risk Accepted**

CS-RESUPPLY-sreUSD-003

`LinearRewardsErc4626` performs incorrect computations if donations have been made to the address before deployment. Note that sreUSD is not affected due to the override of `calculateRewardsToDistribute`. However, future contracts inheriting from `LinearRewardsErc4626` could be affected. Thus, for the below, consider only the code of `LinearRewardsErc4626`.

Namely, the constructor syncs and distributes rewards to initialize the state:

```
_syncRewards();
_distributeRewards();
```

As part of `_syncRewards` the reward cycle data will be written in the constructor. In case a donation was made, that reward will be registered in `rewardCycleData.rewardCycleAmount`. In `_distributeRewards`, the reward to distribute will be computed and added to the `storedTotalAssets`. `previewDistributeRewards` however will compute an incorrect result:

```
uint256 _deltaTime = _timestamp > _rewardsCycleData.cycleEnd
    ? _rewardsCycleData.cycleEnd - _lastRewardsDistribution
    : _timestamp - _lastRewardsDistribution;

// Calculate the rewards to distribute
_rewardToDistribute = calculateRewardsToDistribute({
    _rewardsCycleData: _rewardsCycleData,
    _deltaTime: _deltaTime
});
```

More specifically, `_deltaTime` will be computed based on the `_lastRewardsDistribution` which is still 0 (as it has never been set). As a consequence, `calculateRewardsToDistribute` will receive a too high time delta (i.e. exceeding the maximum cycle length). Thus, the amount to distribute will be too high.

Consider the following example:

1. A donation of `1e18` has been made.
2. A contract inheriting from `LinearRewardsErc4626` is deployed.
3. The reward to distribute will be at the time of writing ~`1.7*1e9 * 1e18 = 1.7*1e27`.

Note that this does not affect the first deposit since the minting in those cases is 1:1. However, later withdrawals will use a higher `storedTotalAssets` which will lead to incorrect accounting thereafter.

Note that `sreUSD` is not affected since both `storedTotalAssets` and `maxDistributionPerSecondPerAsset` will be 0 at the time of the code execution in that context. That leads to a limit of 0 reward distribution.

Risk accepted:

Resupply is aware and accepts the risk.

7.2 Incorrect or Incomplete Interface Definitions

Informational | Version 1 | Code Partially Corrected

CS-RESUPPLY-sreUSD-009

Many of the contracts do not implement their interfaces. As a result several problems arise. Below is a non-exhaustive list of potential problems.

- `IFeeDepositController`:
  - `stakedStable` is missing from the struct `Splits`.
  - The type `uint80` is incorrect. It should be `uint40` in the struct `Splits`.
  - The event and event signature are wrong for `SplitsSet` as it is missing the `stakedStable` parameter and has the wrong types (`uint80` instead of `uint40`).
  - Multiple functions are not properly defined like `setMaxAdditionalFeeRatio` or certain getters for the state variables (e.g., `priceWatcher`, `feeLogger`, `maxAdditionalFeeRatio`).
  - The event `MaxAdditionalFeeRatioSet` is missing in the interface definition.
- `IFeeLogger`:
  - Events are missing in the interface definition.
  - Non-implemented functions are included in the interface definition (`updateTotalFees` and `updateInterestFees`).
  - Getters for the state variables are missing in the interface definition.
- `IPriceWatcher`:
  - Event definitions are missing in the interface definition.
  - Getters for the state variables are missing in the interface definition.
  - Functions are missing in the interface definition (`getCurrentWeight`, `canUpdatePriceData` and `setOracle`).
- `RewardHandler`:
  - Events are missing in the interface definition.
  - Getters for the state variables are missing in the interface definition.
  - Functions are missing in the interface definition (`setBaseMinimumWeight`, `setPairMinimumWeight` `distribute` and `getPairRate`).
- `InterestRateCalculatorV2` does not have an interface definition.
- `sreUSD` does not have an interface definition.
- `LinearRewardsErc4626` does not have an interface definition.

Code partially corrected:

Not all interfaces were corrected:

- `FeeDepositController` & `IFeeDepositController`:
    - The interface defines `treasury()` and `feeDeposit()` which the contract does not.
    - The contract defines getters for `priceWatcher`, `feeLogger`, `epochWeighting` which the interface does not declare.
- `FeeLogger` & `IFeeLogger`:
    - The contract defines and uses events `LoggedEpochTotalFees`, `LoggedEpochInterestFees` and `LoggedPairEpochFees` whereas the interfaces define the events `LogTotalFees` and `LogInterestFees`.
- `InterestRateCaluclatorV2` & `IInterestRateCaluclatorV2`:
    - The contract declares the automatically generated getter `priceWatcher()` while the interface does not declare that function.
- `PriceWatcher` & `IPriceWatcher`:
    - The contract declares the automatically generated getters for `priceData` and `timeMap` which the interface does not declare.
- `RewardHandler` & `IRewardHandler`:
    - The contract declares the automatically generated getters for `registry`, `revenueToken`, `insurancePool`, `govStaker` and `emissionToken` which the interface does not declare.
- `sreUSD` and `LinearRewardsErc4626` continue to not have an interface declaration.

7.3 Partially Missing NatSpec

Informational **Version 1** **Acknowledged**

CS-RESUPPLY-sreUSD-012

Some contracts have extensive NatSpec and others have very sparse to no NatSpec. Using consistent NatSpec across all contracts would improve the quality and readability of the code.

Acknowledged:

Resupply acknowledges the issue and a few changes were applied.

7.4 Permit Griefing

Informational **Version 1** **Acknowledged**

CS-RESUPPLY-sreUSD-013

`LinearRewardsErc4626.depositWithSignature` allows users to deposit in one transaction by leveraging reUSD's `permit` functionality. However, calls to the function can be frontrun with regular calls to `reUSD.permit` which will consume the signature and give approval to sreUSD. However, the deposit will revert, and the user will need to call `deposit` again. Note that it is best practice to wrap `permit` calls in a `try/catch` to handle such griefing cases (see OpenZeppelin).

Acknowledged:

Resupply accepts the risk.

7.5 Unsanitized Values and Unsafe Casts

Informational Version 1 Code Partially Corrected

CS-RESUPPLY-sreUSD-014

Some variables are not sanitized and some instances could have unsafe casts. Below is a list of such occurrences:

1. `FeeDepositController.distribute`: `uint128(dw / dt)` could technically overflow. In such cases it might be more reasonable to migrate to a new controller.
2. `InterestRateCalculatorV2.getNewRate`: `uint128(IERC4626(_vault).convertToShares(1e18))` may overflow in hypothetical scenarios. The interest may be computed incorrectly in the next iteration. However, such scenarios could be handled gracefully.
3. `InterestRateCalculatorV2.getNewRate`: Both casts when computing `_newRatePerSec` could hypothetically overflow, leading to an incorrect rate. However, such scenarios could be handled gracefully.

We recommend to also check all constructors. Even if it can be assumed to be trusted, best practice would be to sanitize the constructor arguments as this could be problematic. E.g.,

1. `FeeDepositController.constructor`: `_stakedStableSplit` is cast to `uint40` without any additional checks. Note that `_insuranceSplit` and `_treasurySplit` are not safely cast but their size is constrained. Ultimately, a too high `_stakedStableSplit` may lead to unexpected results and incorrect events.
2. `sreUSD.constructor`: Does not force the same type limits like the setter for `maxDistributionPerSecondPerAsset`.
3. `LinearRewardsErc4626.constructor`: `_rewardsCycleLength` has no bounds validation. Could be set to 0 or very large values.
4. `InterestRateCalculatorV2.constructor`: Rate parameters (`_minimumRate`, `_rateRatioBase`, `_rateRatioAdditional`) lack validation.

Code partially corrected:

The following two constructor checks were implemented. The remaining risk was accepted by Resupply.

- In `FeeDepositController.constructor` the cast has been resolved.
- In `sreUSD.constructor` the limits are enforced consistently

7.6 Unused Imports

Informational Version 1 Code Partially Corrected

CS-RESUPPLY-sreUSD-015

Some imports are never used. Below is a non-exhaustive list:

- PriceWatcher.sol: `IERC4626`
- sreUSD.sol: `IERC20` and `ERC20`

Code partially corrected:

sreUSD still imports `ERC20`.

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 Arbitrary Cycle Prolongation

Note Version 1

If the time until the next cycle would end is too short, the cycle end is moved further to the future. Note that the divisor 40 is chosen arbitrarily:

```
// Calculate the next cycle end, this keeps cycles at the same time regardless of when sync is called
uint40 _cycleEnd = (((_timestamp + REWARDS_CYCLE_LENGTH) / REWARDS_CYCLE_LENGTH) * REWARDS_CYCLE_LENGTH)
    .safeCastTo40();

// This block prevents big jumps in rewards rate in case the sync happens near the end of the cycle
if (_cycleEnd - _timestamp < REWARDS_CYCLE_LENGTH / 40) {
    _cycleEnd += REWARDS_CYCLE_LENGTH.safeCastTo40();
}
```

However, Resupply confirmed that the value is the expected value. Additionally, Resupply confirmed that cycles are expected to be defined as implemented.

8.2 Boosted Distribution Considerations

Note Version 1

The boosted interest fees rely on many assumptions for them to be accurate which, for example, includes the configuration of `InterestRateCalculatorV2` and the `FeeDepositController`. Below is a list of considerations for governance, developers, and users.

Interest Rate Updates and Distribution Timing. The distribution of fees should be optimally at the very beginning of every epoch and should optimally be immediately followed by fee withdrawals on all pairs. Otherwise, the average weights for a pair might be unsuitable to use. Additionally, not distributing may lead to a lack of checkpointing which leads to forfeiting the boost.

InterestRateCalculatorV2 shared. `InterestRateCalculatorV2` must be shared or at least have the same configuration among all pairs. Otherwise, boosting may be inaccurate as outlined in the last item of this note.

InterestRateCalculatorV2 changes. `InterestRateCalculatorV2` should optimally not be changed. Otherwise, boosting may be inaccurate as outlined in the last item of this note.

Configuration of Parameters. The parameters `rateRatioAdditional` and `maxAdditionalFeeRatio` must be correctly configured for the computations to be meaningful.

Note that this part of the note, for simplicity, interprets most variables as floating point numbers and writes them as such for simplicity and readability.

`FeeDepositController` computes interest generated as part of the boosted interest as the difference of the total interest generated and the base interest. From the computations, one can conclude that the fees in interest can thus be computed from the base interest from a multiplicative factor as follows:

```
feesInInterest = baseInterest * (1 + maxAdditionalFeeRatio * avgWeight)
```

The `InterestRateCalculatorV2` and the pair will compute the interest fee typically as (simplified):

```
feesInInterest = x * collateralInterest * (rateRatioBase + (rateRatioAdditional * priceweight))
```

However, for the computation to be matching the one in `FeeDepositController`, `rateRatioAdditional` must be defined as `rateRatioBase * maxAdditionalFeeRatio` so that:

```
feesInInterest = x * collateralInterest * rateRatioBase * (1 + maxAdditionalFeeRatio * priceweight)
               = baseInterest * (1 + maxAdditionalFeeRatio * priceweight)
```

To summarize, the *configuration must ensure that* `rateRatioAdditional = rateRatioBase * maxAdditionalFeeRatio` holds. Otherwise, inaccurate boosting may occur.

Note that as of Version 2, `maxAdditionalFeeRatio` has been changed to `additionalFeeRatio`. Also, note that initial issue *No consideration of floor rates* was presented within this note.

8.3 FeeLogger Owner Can Log

Note **Version 1**

Note that the owner can call the log function in the `FeeLogger` contract. Resupply suggests that this is for future-proofing in case something else should tie into it

8.4 FeeLogger Will Log Donations

Note **Version 1**

Currently, no usage for `FeeLogger.epochTotalFees` has been implemented. Developers and governance should be aware that the logged total fees might include donations since `FeeDepositController.distribute` simply logs its balance.

8.5 Frequent Reward And Fee Updates

Note **Version 1**

Governance should be aware that some operations might need monitoring to ensure that they are triggered frequently enough. While Manual Price Watcher Updates puts focus on bot operations on the price watcher, this note aims to outline the importance of frequent fee and reward claiming.

- `FeeDepositController.distribute`: Should be called at the very beginning of an epoch to ensure accurate fee distributions.
- `Pair.withdrawFees`: Should be called as soon as possible after `distribute` to ensure accurate interest accounting within the fee logger.
- `sreUSD.syncRewardsAndDistribution`: Should be called as soon as possible when the rewards arrive or when the cycle ends to ensure fees are distributed in the optimal time frame.

While deviations from this are handled, governance should run bots to ensure that the fee mechanism is as accurate as possible (e.g. if no actions within first hour of an epoch, run the bots accordingly).

8.6 Manual Price Watcher Updates

Note Version 1

The price watcher serves as a source for the interest rate computation. To ensure that the interest rate is adapting accordingly, the update should be triggered frequently. Note that this happens automatically:

- During fee distribution as part of `FeeDepositController.distribute` (optimally, this happens at the very beginning of every epoch).
- When fetching incentives during pool operations such as borrowing or repaying or other operations.

However, if activity is low and if no rewards are distributed, the interest rate may remain without updates.

That may lead to the following problems:

- Mechanism for incentivizing sreUSD deposits might not work as expected.
- The search for the price weight index in `PriceWatcher.findPairPriceWeight` might become too expensive for some pools. For example, assume that a suitable update timestamp requires 1000 iterations (250 days without updates before the last pool update), then it might become too expensive for users to use the protocol (e.g. liquidations might be too expensive as a search would require at least 1000 SLOADs leading to a minimum of 2.1M gas cost). Similarly, the loop might fill the full block. Assuming a block gas limit of 45M, the block gas limit could be exceeded if for ~14 years prior to the pool's last update no update has occurred.

Governance and developers should monitor the frequency of the price watcher updates and update manually if necessary (e.g. depeg but no price update for a certain period of time).

8.7 Oracle Considerations

Note Version 1

The reUSD oracle must be safe and sound. Governance should monitor its suitability.

Namely, the following properties of highest importance (holds mainly for the underlying oracle):

- Correctness: The source code must be correct and sound.
- Staleness: The oracle must be updated frequently to give meaningful values.
- Safe: The oracle should be safe to use (e.g. no reverts).
- Non-manipulateable: The oracle should not be manipulateable. While minimal manipulations might be fine, more severe manipulations might lead to profitable scenarios in terms of interest generated for sreUSD. Thus, users might be earlier liquidated if interest is unnecessarily high.
- No zero returns: Returning 0 will return to a DoS of the oracle which might impact regular operations.

8.8 Resupply Registry Considerations

Note Version 1

Changes of core contracts in the registry might lead to severe problems. Any such change must ensure that nothing breaks. Below is a list of considerations (assuming upgrades are non-malicious as this could lead to many unforeseen implications):

- Reverts (due to e.g. gas, interface mismatches and similar)
- Undistributed rewards for `sreUSD`.

- Changing the `FEE_LOGGER`, `PRICE_WATCHER` has no impact on `FeeDepositController` as they are stored as immutables. However, the contracts should not be replaced to ensure information is easy to retrieve.
- Changing the `feeDeposit` may lead to critical accounting problems due to fee distribution being multiple times possible in the same epoch.
- Changing `SREUSD` allows sending funds accordingly. However, changing the sreUSD token is pointless in the sense that sreUSD's address should not change.
- Changing `feeDeposit`, `feeDeposit.operator` or `rewardHandler` may lead to incorrect logging within the `FeeLogger`.
- Changing `REUSD_ORACLE` must be followed by `PriceWatcher.setOracle` to ensure that the price watcher can track prices as desired.

Note that the list above is non-exhaustive. Any updates to the registry must be performed with the highest diligence and carefulness.

8.9 Reward Handler Migration Enforcement

Note **Version 1**

Note that the `RewardHandler`'s functionality will work even if the state has not been migrated at all with `migrateState`.

Resupply specified that it will be ensured that the above will be part of the proposal.

8.10 Zero-Weight for Price Watcher Weight Prior to Index 1

Note **Version 1**

Governance should be aware that the price watcher assumes 0 weight for the time prior to the first weight update. As a consequence, the average taken might be inaccurate.

8.11 sreUSD Rewards Are Delayed

Note **Version 1**

sreUSD holders will be incentivized to deposit reUSD into sreUSD in case of a depeg as part of the economic model implemented by Resupply.

Note that the interest rate is immediately reactive and will accrue interest immediately. However, note that the sreUSD holders will only start receiving the benefits from stabilizing the system after 2 epochs.

Thus, arbitrary users could do the following:

1. Wait for the peg to be restored again (e.g. 1.99 epochs).
2. Deposit into sreUSD to receive the boosted yield.

Ultimately, sreUSD depositors might not deposit into sreUSD when needed but rather when the peg is restored but the boosted yield can be enjoyed.

For example, that could lead to additional borrowing to allow for depositing into sreUSD to have fully self-repaying loans.