yAudit

November 24, 2025

Prepared for
Resupply Finance

Audited by
HHK
adriro

Resupply sreUSD

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Resupply Finance is a CDP-based lending protocol that allows simple, low-risk, leveraged yield farming while encouraging the use of value-added ecosystem protocols' underlying stables like Curve's crvUSD and Frax's FRAX.

1.2 Audit Scope

This audit covers 7 smart contracts totaling approximately 750 lines of code across 4.5 days of review.

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

1.3 Risk Assessment Framework

1.3.1 Severity Classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	3
Informational	1

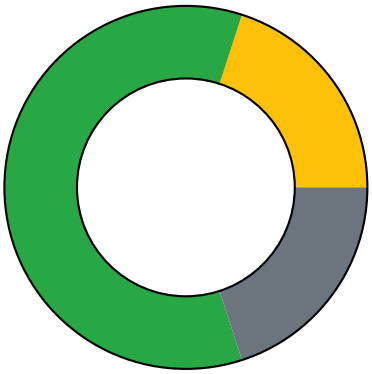

Figure 1: Distribution of security findings by impact level

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

Table 1: severity classification

1.5 Overall Assessment

The update includes the new ERC-4626 vault for savings reUSD, along with changes to the interest rate calculator to boost fees depending on the stable peg.

2 Audit Overview

2.1 Project Information

Protocol Name: Resupply Finance
Repository: https://github.com/resupplyfi/resupply
Commit Hash: 9a3de5b4a3ced4eb993c87cc19cf85c99bf3e6a2
Commit URL:
https://github.com/resupplyfi/resupply/commit/9a3de5b4a3ced4eb993c87cc19cf85c99bf3e6a2

2.2 Audit Team

HHK, adriro

2.3 Audit Resources

sreUSD Auditor Roadmap

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 PriceWatcher weights can exceed the `1e6` scale

Weights can potentially fall outside the expected scale, affecting the PriceWatcher consumers.

Technical Details

The implementation of `getCurrentWeight()` fetches the reUSD price from the Oracle and adjusts the scale to return a value within `1e6` precision.

```
153:     function getCurrentWeight() public view returns (uint64) {
154:         uint256 price = IReusdOracle(oracle).price();
155:         uint256 weight = price > 1e18 ? 0 : 1e18 - price;
156:         //our oracle has a floor that matches redemption fee
157:         //e.g. it returns a minimum price of 0.9900 when there is a 1% redemption
fee
158:         //at this point a price of 0.99000 has a weight of 0.010000 or 1e16
159:         //reduce precision to 1e6
160:         return uint64(weight / 1e10);
161:     }
```

The expected behavior here is that the Oracle price is clamped to $1 minus the redemption fee (1%), making the resulting weight fall within the `[0, 0.01]` range, which is then scaled down to the `1e6` region
However, two dependencies could potentially break this invariant.

- The ReusdOracle `price()` function multiplies the reUSD price (floored to the redemption fee) by the crvUSD price, potentially allowing the resulting value to be less than $0.99
- The redemption fee can be updated, breaking the floored price assumption.

Impact

Medium. Users of the PriceWatcher contract consume the weight as a rate while assuming the returned scale is `1e6`, potentially causing downstream calculations to exceed 100%.

Recommendation

The weight calculation should use `priceAsCrvusd()`, which returns the floored price without the crvUSD price scaling. Additionally, ensure that the redemption fee is not changed, as this could also disrupt the calculations.

Developer Response

Fixed in 6eeae6b.

2.7 Low Findings

2.7.1 Ensure `_updateInterest = true` when setting the new interest rate contract in `setRateCalculator()`

Technical Details

The function `setRateCalculator()` updates the contract used to calculate the interest rate. When called, the param `_updateInterest` can be set to true to flush interests before changing the contract.
The new contract `InterestRateCalculatorV2` and `PriceWatcher` use the pair's `lastPairUpdate` variable to determine the new interest rate. This variable is only updated when interests are flushed/added to the pair.
When the `PriceWatcher` is deployed, it will save two new `priceData`, one at timestamp zero and one at the deployment timestamp's floor.
If the `lastPairUpdate` is smaller than the deployment's timestamp floor, the `findPairPriceWeight()` function called by the `InterestRateCalculatorV2` will loop forever looking for a valid `priceData` all the way to timestamp zero, which will likely result in the transaction reverting out of gas.

Impact

Low. The pair may not be able to determine their interests again until a new interest rate contract is established.

Recommendation

Ensure that the param `_updateInterest` is set to true inside the deployment script when calling `setRateCalculator()` (already the case inside `LaunchSreUsd`) and document this in the natspec of the `InterestRateCalculatorV2` contract.

Developer Response

Included in the draft governance proposal: LaunchSreUsd.s.sol#L126.

2.7.2 Incorrect split update implementation

The new staked reUSD split is not considered in the update logic of FeeDepositController.

Technical Details

The implementation of `setSplits()` takes all four split arguments but requires only three of them to equal the total BPS.

```
181:     function setSplits(uint256 _insuranceSplit, uint256 _treasurySplit, uint256 _platformSplit, uint256 _stakedStableSplit) external onlyOwner {
182:         require(_insuranceSplit + _treasurySplit + _platformSplit == BPS, "invalid splits");
183:         splits.insurance = uint40(_insuranceSplit);
184:         splits.treasury = uint40(_treasurySplit);
185:         splits.platform = uint40(_platformSplit);
186:         splits.stakedStable = uint40(_stakedStableSplit);
187:         emit SplitsSet(uint40(_insuranceSplit), uint40(_treasurySplit), uint40(_platformSplit), uint40(_stakedStableSplit));
188:     }
```

Additionally, during initialization, the `stakedStable` variable is missing from the check on line 62. However, in this case, the split can still be initialized properly and would overflow on line 66 if the splits exceed 100%.

```
62:         require(_insuranceSplit + _treasurySplit <= BPS, "invalid splits");
```

Impact

Low.

Recommendation

Change the condition in `setSplits()` to account for the new staked split.

```
-   require(_insuranceSplit + _treasurySplit + _platformSplit == BPS, "invalid splits");
+   require(_insuranceSplit + _treasurySplit + _platformSplit + _stakedStableSplit == BPS, "invalid splits");
```

The condition in the contract's constructor can be adjusted as well.

```
-   require(_insuranceSplit + _treasurySplit <= BPS, "invalid splits");
+   require(_insuranceSplit + _treasurySplit + _stakedStableSplit <= BPS, "invalid splits");
```

Alternatively, the platform split can be removed and interpreted as the difference between 100% and the sum of the other three splits, aligning the behavior with the actual split implementation.

Developer Response

Fixed the summation check in 2f9510b.
We think it's preferable to leave the struct as-is for the following reasons:

1. Explicitness.
2. Convenient for users or indexers to read from.

2.7.3 Preview sync rewards doesn't account for fee distribution

The public-facing variant of `previewSyncRewards()` doesn't account for fee distribution and will return incorrect results.

Technical Details

The implementation of `_syncRewards()` has been updated so that `_distributeFees()` is called before `previewSyncRewards()` but only if the current cycle has elapsed (`block.timestamp <= rewardsCycleData.cycleEnd`).

```
157:     function _syncRewards() internal virtual {
158:         if (block.timestamp <= rewardsCycleData.cycleEnd) return;
159:         _distributeFees();
160:         RewardsCycleData memory _rewardsCycleData = previewSyncRewards();
```

This is necessary to ensure fees are incorporated into the contract and accounted for as rewards in the upcoming cycle.
However, if `previewSyncRewards()` is called externally, the fee distribution isn't executed, and rewards are not accounted for in the preview simulation.

Impact

Low. `previewSyncRewards()` returns incorrect results.

Recommendation

The implementation should be refactored so that `previewSyncRewards()` simulates the effects of fee distribution.

Developer Response

Fixed in 48d2b16.

2.8 Gas Savings Findings

2.8.1 Return early inside `previewDistributeRewards()`

Technical Details

Inside the function `previewDistributeRewards()` return early if `lastRewardsDistribution == block.timestamp`.
This will save gas on deposits and withdrawals as the contract currently `_distributeRewards()` first, which will update `lastRewardsDistribution` but then call the function `previewDistributeRewards()` again inside `totalAssets()` to determine the amount of shares to mint/burn.

Impact

Gas.

Recommendation

Inside the function `previewDistributeRewards()`, return early if `lastRewardsDistribution == block.timestamp`.

Developer Response

Updated in b2139c2.

2.9 Informational Findings

2.9.1 Ensure no distribution is triggered when deploying sreUSD

The initialization of LinearRewardsErc4626 triggers a dummy distribution to bootstrap to the state, which could create a conflict if actual rewards are being distributed.

Technical Details

The constructor of LinearRewardsErc4626 calls `_syncRewards()`

```
67:         // initialize rewardsCycleEnd value
68:         // NOTE: normally distribution of rewards should be done prior to
_syncRewards but in this case we know there are no users or rewards yet.
69:         _syncRewards();
70:
71:         // initialize lastRewardsDistribution value
72:         _distributeRewards();
```

As the comment indicates, it is expected that no users or rewards are present at this point. However, this may not be true if the call to `_distributeFees()` ends up doing an actual distribution of tokens.

Impact

Informational.

Recommendation

Ensure no reward distribution happens as part of the initialization of the staked reUSD vault.

Developer Response

Added an extra sanity check in constructor: c41e8ae.

2.10 Final Remarks

The Resupply Protocol's staked reUSD implementation represents a new extension to the existing ecosystem, introducing a yield-bearing vault that leverages a reward distribution mechanism based on collected fees from existing markets.
The audit revealed a well-structured implementation with no critical or high-severity vulnerabilities identified. The medium-severity findings were primarily related to edge cases in the PriceWatcher weight calculations, which could potentially affect downstream consumers but do not pose immediate security risks to user funds.
The development team demonstrated exceptional responsiveness throughout the audit process, promptly addressing all identified issues, reflecting their strong commitment to security and code quality.