yAudit Resupply Finance Review

Review Resources:

- Document describing the protocol

Auditors:

- adriro
- HHK
- Spalen

Table of Contents

1 Review Summary

2 Scope

3 Code Evaluation Matrix

4 Findings Explanation

5 Critical Findings

a 1. Critical - Wrong variable initialization leads to debt write-off and under-collateralization

b 2. Critical - Anyone can steal Merkle claim airdrop

c 3. Critical - Proposals can be replayed in Voter.sol

6 High Findings

a 1. High - `sweepUnclaimed()` could easily overflow, leading to locked tokens

b 2. High - Invalid duration for pre-configured vesting

c 3. High - PermaLocker contract can unstake after migration

7 Medium Findings

a 1. Medium - Incorrect initial supply split in VestManager.sol

b 2. Medium - Multiple unsafe downcasting in `_borrow()`

c 3. Medium - Missing `_addInterest()` inside `redeemCollateral()`

d 4. Medium - Disabling cooldown in GovStaker.sol bricks withdrawals

e 5. Medium - Debt won’t be cleared if collateral balance is zero inside `distributeCollateralAndClearDebt()`

f 6. Medium - Lack of funds in the insurance pool can block liquidations

g 7. Medium - Depositing into the insurance funds doesn’t reset the withdrawal queue

8 Low Findings

a 1. Low - Unsafe cast could be used to drain EmissionsController

b 2. Low - Wrong check in `_calculateInterest()`

c 3. Low - Missing reentrancy protection in `GovStaker::exit()`

d 4. Low - Generic target is not checked in canceler payload

e 5. Low - Impossibility to reset the `pid` back to 0

f 6. Low - Possible wrong transfer amount inside `distributeCollateralAndClearDebt()`

g 7. Low - Second split of creation code is not cleared in ResupplyPairDeployer.sol

h 8. Low - Incorrect implementations in RedemptionHandler.sol

i 9. Low - `claimableFees` is shadowed in `getPairAccounting()`

j 10. Low - Missing debt accrual in `totalDebtAvailable()`

k 11. Low - Add interest before updating the `RateCalculator`

l 12. Low - Possible double allocation in `EmissionController`

m 13. Low - No cooldown period when updating the LTV

n 14. Low - Unsafe swap deadline

o 15. Low - Default swappers cannot be added from the registry contract

9 Gas Saving Findings

a 1. Gas - Gas savings in `ResupplyPairCore.sol`

b 2. Gas - Gas savings in `Voter.sol`

c 3. Gas - Gas savings in `EmissionController.sol`

10 Informational Findings

a 1. Informational - Unused function in `SimpleRewardStreamer.sol`

b 2. Informational - Natspec not updated/incorrect/missing for some functions

c 3. Informational - Division by 0 in `currentUtilization()` when the pair is paused
d 4. Informational - Collateral should not be allowed as a reward
e 5. Informational - Unused variables
f 6. Informational - The protocol assumes the underlying has 18 decimals
g 7. Informational - Missing safe ERC20 wrapping
h 8. Informational - Interest rates are not strictly calculated by epoch
i 9. Informational - Misleading `claimableEmissions()` implementation
j 10. Informational - Voting power can be used after unstake
k 11. Informational - Ensure owner matches caller in InsurancePool.sol `redeem()` and `withdraw()`
l 12. Informational - Limited usage of Protocol pausing in `Core.sol`
m 13. Informational - Repay with collateral will be challenging to estimate off-chain
n 14. Informational - Missing or incorrect events
o 15. Informational - Account argument is ignored in some InsurancePool.sol functions
11 Final remarks

Review Summary

Resupply Finance

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.

The contracts of the Resupply Finance Repo were reviewed over 25 days. Three auditors performed the code review between November 25 and December 27 2024, 2024. The repository was under active development during the review, but the review was limited to the latest commit 075121327cfb92ec2a6b885d517714d9d099f49a for the Resupply Finance repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
├── Constants.sol
├── dao
│   ├── Core.sol
│   ├── GovToken.sol
│   ├── Treasury.sol
│   ├── Voter.sol
│   ├── auth
│   │   └── GuardianAuthHook.sol
│   ├── emissions
│   │   ├── EmissionsController.sol
│   │   └── receivers
│   │       ├── ExampleReceiver.sol
│   │       ├── SimpleReceiver.sol
│   │       └── SimpleReceiverFactory.sol
│   ├── operators
│   │   └── GuardianOperator.sol
│   ├── staking
│   │   ├── GovStaker.sol
│   │   ├── GovStakerEscrow.sol
│   │   └── MultiRewardsDistributor.sol
│   └── tge
│       ├── PermaLocker.sol
│       ├── VestManager.sol
│       └── VestManagerBase.sol
├── dependencies
│   ├── CoreOwnable.sol
│   ├── CorePausable.sol
│   ├── DelegatedOps.sol
│   └── EpochTracker.sol
├── interfaces/*
├── libraries
│   ├── MathUtil.sol
│   ├── SafeERC20.sol
│   └── VaultAccount.sol
└── protocol
```

```
    ├── BasicVaultOracle.sol
    ├── FeeDeposit.sol
    ├── FeeDepositController.sol
    ├── InsurancePool.sol
    ├── InterestRateCalculator.sol
    ├── LiquidationHandler.sol
    ├── RedemptionHandler.sol
    ├── ResupplyPair.sol
    ├── ResupplyPairDeployer.sol
    ├── ResupplyRegistry.sol
    ├── RewardDistributorMultiEpoch.sol
    ├── RewardHandler.sol
    ├── SimpleRewardStreamer.sol
    ├── Stablecoin.sol
    ├── WriteOffToken.sol
    └── pair
        ├── ResupplyPairConstants.sol
        └── ResupplyPairCore.sol
```

After the findings were presented to the Resupply Finance team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Resupply Finance and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Average	Access control modifiers are generally appropriate, with a few exceptions that resulted in findings.
Mathematics	Average	The codebase primarily relies on simple mathematics. However, some unsafe downcast issues were identified, though their impact is minimal.
Complexity	Average	The codebase is relatively straightforward but extensive, comprising numerous small contracts and reward mechanisms that interact, adding to its overall complexity.
Libraries	Good	The project leverages the battle-tested OpenZeppelin library and a forked contract from Frax Finance.
Decentralization	Good	The project will be fully decentralized from day one, featuring on-chain voting mechanisms.
Code stability	Good	Apart from a few small changes, the codebase remained stable during the audit.
Documentation	Good	A document outlining the protocol was provided at the start of the review.
Monitoring	Average	Most functions emit events with a few exceptions that were fixed.
Testing and verification	Low	More thorough testing could have prevented several issues. While fork testing is implemented, fuzzing and invariant testing are not.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.

- Gas savings
    - Findings that can improve the gas efficiency of the contracts.
- Informational
    - Findings including recommendations and best practices.

Critical Findings

1. Critical - Wrong variable initialization leads to debt write-off and under-collateralization

The variable `_collateralForLiquidator` will always be set to 0 if the collateral sent to the insurance pool does not exceed the collateral available. This leads to free liquidation as the borrower will not lose collateral, and the insurance pool will not lose liquidity.

Technical Details

The function `liquidate()` will be called only by the `LiquidationHandler` contract.

Inside the function, it determines how much collateral needs to be sent to the `liquidationHandler`. It does so by calculating the optimistic amount (the maximum amount possible using the liquidation fee). If the optimistic amount exceeds the collateral available, it will only send the available collateral.

However, this is only the theory; the variable `_collateralForLiquidator` is supposed to reflect this logic, but the initialization is wrong.

```
int256 _leftoverCollateral;
...
_collateralForLiquidator = _leftoverCollateral <= 0
                ? _userCollateralBalance
                : _collateralForLiquidator;
```

If the collateral available is smaller than the optimistic amount, then `_collateralForLiquidator` is set to itself, which is 0.

This results in the `LiquidationHandler` receiving zero collateral when liquidating and not adding any debt to its internal accounting.

Later on, the function `processLiquidationDebt()` is called, it is in charge of processing the collateral and burning the debt repaid from the insurance funds.

However in the sub-function `processCollateral()` if the collateral received and redeemed is zero then the function will return early and not burn any stable from the insurance fund.

This results in a debt write-off and, thus, in under-collateralized ReUSD tokens.

POC: can be added inside `LiquidationManagerTest.t.sol` and run with `forge test --mt test_LiquidateBasicZeroForLiquidator -vvv`.

```
function test_LiquidateBasicZeroForLiquidator() public {
        vm.prank(address(core));
        pair.setMaxLTV(90000); //set at 90% instead of 95% so the 5% liquidation bonus doesn't push amount above collateral available
        setOraclePrice(1e18); //set price for collateral

        //build pos
        uint256 amount = 10_000e18;
        uint256 maxBorrow = (
            amount *
            pair.maxLTV() /
            ResupplyPairConstants.LTV_PRECISION
        );
        deal(address(collateral), address(this), amount);
        borrow(pair, maxBorrow - 1e18, amount); // borrow while adding collateral

        //make pos liquidable
        setOraclePrice(0.98e18);

        //save prev balance
        uint256 pairCollateralBalance = collateral.balanceOf(address(pair));
        uint256 ipTotalAssets = insurancePool.totalAssets();

        //try to liquidate
        liquidationHandler.liquidate(address(pair), address(this));

        //assert
        assertEq(insurancePool.totalAssets(), ipTotalAssets, 'total assets should not decrease'); //don't decrease because of early return in processCollateral
        assertEq(collateral.balanceOf(address(pair)), pairCollateralBalance, 'pair collateral balance should not decrease'); //don't decrease because _collateralForLiquidator = 0
    }
```

Impact

Critical. Debts will be written off against zero collateral.

Recommendation

Modify the variable initialization to be equal to the user’s collateral balance.

Developer Response

Fixed in cad853860244462f3856230369461ff9585d1bf1.

2. Critical - Anyone can steal Merkle claim airdrop

`merkleClaim()` function is used to create vesting for a specific account by providing Merkle proof. Attack can frontrun this transaction and change the receiver of the vesting.

Technical Details

Merkle node is generated using following input parameters: `_account`, `_index` and `_amount`. The function has an additional input parameter that is not validated `_recipient`. This enables the attacker to use the claim of others airdrops by providing his address as receiver. The owner transaction would revert, and the attacker would create the vesting with his recipient’s address.

The following PoC shows that the attacker can frontrun the user, use the same proof, and change the `_recipient` parameter. Add the test to the file:
`test/dao/tge/VestManagerHarness.t.sol`

```
function test_AirdropClaim_Attacker() public {
    assertNotEq(address(vestManager), address(0));
    (address[] memory users, uint256[] memory amounts, bytes32[][] memory proofs) =
getSampleMerkleClaimData();
    address attacker = address(0x4774);

    uint256 i = 0;
    // attacker provides his address as receiver
    vestManager.merkleClaim(
        users[i],
        attacker, // attacker has changed `_recipient` param
        amounts[i],
        VestManager.AllocationType.AIRDROP_VICTIMS,
        proofs[i],
        i
    );
    // user claim will revert because the attacker has claimed the vest
    vm.expectRevert("already claimed");
    vestManager.merkleClaim(
        users[i],
        users[i],
        amounts[i],
        VestManager.AllocationType.AIRDROP_VICTIMS,
        proofs[i],
        i
    );
}
```

Impact

Critical. All Merkle claim airdrops can be stolen.

Recommendation

Remove the recipient parameter or restrict who can claim the airdrop for account using the modifier `callerOrDelegated()`.

Developer Response

Fixed with `msg.sender` validation f807106c664b474e9cebf44d8db3974d5b2235c2.

3. Critical - Proposals can be replayed in Voter.sol

Approved proposals can be executed multiple times during the execution window.

Technical Details

The `executeProposal()` function checks if the proposal has not been processed yet, but fails to toggle the flag, allowing anyone to replay an approved proposal during the execution window.

```
    function executeProposal(uint256 id) external {
        require(id < proposalData.length, "Invalid proposalID");

        Proposal memory proposal = proposalData[id];
        require(_canExecute(proposal), "Proposal cannot be executed");

        Action[] storage payload = proposalPayloads[id];
        uint256 payloadLength = payload.length;

        for (uint256 i = 0; i < payloadLength; i++) {
            ICore(core).execute(payload[i].target, payload[i].data);
        }
        emit ProposalExecuted(id);
    }
```

Impact

Critical. Passing proposals can be replayed.

Recommendation

Mark the proposal as processed after checking it can be executed in `executeProposal()`.

```
    function executeProposal(uint256 id) external {
        require(id < proposalData.length, "Invalid proposalID");

        Proposal memory proposal = proposalData[id];
        require(_canExecute(proposal), "Proposal cannot be executed");

+       proposalData[id].processed = true;

        Action[] storage payload = proposalPayloads[id];
        uint256 payloadLength = payload.length;

        for (uint256 i = 0; i < payloadLength; i++) {
            ICore(core).execute(payload[i].target, payload[i].data);
        }
        emit ProposalExecuted(id);
    }
```

Developer Response

Recommendation implemented d178b382750f01724fa02fe72d706b09c4c0a6f3.

High Findings

1. High - `sweepUnclaimed()` could easily overflow, leading to locked tokens

The VestManagerBase.sol contract tracks sweepable balance using the difference between token balance and vesting allocations, which could eventually lead to an overflow and cause a denial of service.

Technical Details

The sweepable amount of tokens after the deadline is reached is defined by the `getUnallocatedBalance()` function as the token balance minus the vesting allocations.

```
    function getUnallocatedBalance() public view returns (uint256) {
        return token.balanceOf(address(this)) - totalAllocated;
    }
```

As vests are created, the `totalAllocated` variable is incremented but never decremented, while the token balance will be decremented as vests are claimed. Assuming there is a surplus of tokens in relation to the final vesting allocations, this difference will shorten as users claim their vest and eventually overflow. For example, the contract starts with 100 tokens, and 75 are allocated to vests. After more than 25 tokens from vests are claimed, the subtraction will overflow, causing a denial of service in `sweepUnclaimed()`.

Impact

High. Tokens could get permanently locked in the vesting contract.

Recommendation

Decrement `totalAllocated` when claims are executed or use a new variable to track the amount of tokens owed to vests.

Developer Response

Fixed by removing this function and the concept of a "deadline" altogether, done in multiple commits:

- remove function d1c5fb330e0c483087453 93d4accb0bf32844627
- remove deadline 2be14ec6dffbcfd804ec25aecd85c7628ebb2663
- cleanup unused vars 2822fe4f73aa0273b5697f567ef5814decbaf1ec

2. High - Invalid duration for pre-configured vesting

During initialization, the VestManager.sol contract creates the vests for the non-user entities by incorrectly setting the vesting duration as the current timestamp instead of using the intended vesting duration for the associated type.

Technical Details

The `setInitializationParams()` is used to prepare all the configuration for the vesting contract. During this initialization, the implementation pre-configures the vests for the non-user entities (`TREASURY`, `PERMA_LOCKER1`, and `PERMA_LOCKER2` types).

```
            // Create vest for non-user targets
            if (i < _nonUserTargets.length) {
                _createVest(
                    _nonUserTargets[i],
                    uint32(block.timestamp),
                    uint112(allocation)
                );
            }
```

The second argument to the `_createVest()` function is the duration of the vest. The implementation is incorrectly setting the duration as the current timestamp.

Impact

High. Vesting for non-users will last for many years instead of the intended duration.

Recommendation

Use the proper duration parameter.

```
    // Create vest for non-user targets
    if (i < _nonUserTargets.length) {
        _createVest(
            _nonUserTargets[i],
-           uint32(block.timestamp),
+           uint32(_vestDurations[i]),
            uint112(allocation)
        );
    }
```

Developer Response

Fixed by passing correct value 246a0b0f23c75f4ab998758dc3e7c701ee869c10.

3. High - PermaLocker contract can unstake after migration

The mechanism to prevent unstaking in PermaLocker.sol is ineffective after a staking contract migration.

Technical Details

The PermaLocker.sol contract includes functions to allow arbitrary execution. To prevent these functions from being used to unstake tokens, they are guarded by the `noUnstaking` modifier, which checks that the balance of staked tokens doesn’t decrease.

```
    modifier noUnstaking {
        bool shouldCheck = !unstakingAllowed;
        uint256 pre;
        if (shouldCheck) {
            pre = staker.balanceOf(address(this));
        }
        _;
        if (shouldCheck) {
            require(
                staker.balanceOf(address(this)) >= pre,
                "UnstakingForbidden"
            );
        }
    }
```

The implementation also supports upgrading the staking contract via the `migrateStaker()` function, which queries the registry and updates the `staker` reference if this has changed.

```
    function migrateStaker() external onlyOwner {
        address _newStaker = registry.staker();
        require(_newStaker != address(0), "Staker not set");
        address _oldStaker = address(staker);
        require(_oldStaker != _newStaker, "No change");

        govToken.approve(_newStaker, type(uint256).max);
        govToken.approve(_oldStaker, 0);
        staker = IGovStaker(_newStaker);

        emit StakerMigrated(_oldStaker, _newStaker);
    }
```

Once the `staker` variable has changed, a malicious owner or operator can simply unstake from the previous staker since the modifier will apply the checks using the new reference.

Impact

High. Stake intended to be permanently locked can be removed after a migration.

Recommendation

Ideally the migration process should unstake first from the previous staker and move those tokens to the new contract atomically. Given that the current implementation of GovStaker.sol lacks this functionality, the process should be split between multiple steps to first cooldown, then unstake, and finally stake in the new contract. Alternatively, the staker interface could implement a generic migration function to move tokens without any delay so that `migrateStaker()` can call this function as part of the process.

Developer Response

Staker updated for trustless perma-staking and migration PR#28.

Medium Findings

1. Medium - Incorrect initial supply split in VestManager.sol

The VestManager.sol contract reserves a portion of the initial token supply intended for emissions, which may remain unused and potentially locked within the contract.

Technical Details

The `setInitializationParams()` function takes an `_allocPercentages` array parameter with the chosen split for each of the allocation types. This array holds an extra element with the percentage allocated to emissions, as indicated by the documentation:

> @param _allocPercentages Percentages of the initial supply allocated to each type, with the final value being the total percentage allocated for emissions.

This percentage is considered in the checks as being part of the initial supply sent to the vesting contract but never assigned or distributed. The EmissionsController.sol contract, in charge of continuous emissions, will mint the required RSUP tokens at each epoch.

Impact

Medium. Any reserved amount of tokens in VestManager.sol intended for emissions will remain locked in the contract, as emissions are handled elsewhere.

Recommendation

Remove the extra element in the `_allocPercentages` array and adjust the `setInitializationParams()` function accordingly.

Developer Response

Fixed in 2be14ec6dffbcfd804ec25aecd85c7628ebb2663.

2. Medium - Multiple unsafe downcasting in `_borrow()`

Technical Details

The function `totalDebtAvailable()` is in charge of determining how much more debt can be minted.

- This function returns an `uint256`, its value can be greater than `type(uint128).max` but when it is called inside `_borrow()` the value is then downcasted to `uint128` since the `totalBorrow` variable is an `uint128`.

This means the value used inside the `_borrow()` might differ from what was initially returned by `totalDebtAvailable()` if the value returned was greater than `type(uint128).max`.

- The variable `debtForMint` is also unsafely downcasted when initialized. The risk of a harmful result is very low as this would require a very high `borrowAmount` and a `mintFee`, but it is still not advised.

- The `_sharesAdded` computation is downcasted to `uint128`; if the share ratio increases too much because of redemptions, borrowing a high amount could lead `_sharesAdded` to be higher than `uint128`, which would result in the `_totalBorrow.shares` to be smaller than it should have been.

Impact

Low. Unsafe downcasting.

Recommendation

- Check if the value returned is greater than `type(uint128).max` inside `totalDebtAvailable()` and if so then return `type(uint128).max`.
- Initialize `debtForMint` as `uint256` and then use safeCast library to cast as `uint128()` when incrementing `totalBorrow`.
- Use safeCast library to cast as `uint128()` for `_sharesAdded` and throughout the whole codebase for extra caution.

Developer Response

Fixed in 430242f45eb15d8eb86dce5605a177027aacd91d, 5bd625e2698b53b0e15b033f812923e4ddb944af and 9ee5a2e26974d4fba5ba0a0b6bcf20afeec98054.

3. Medium - Missing `_addInterest()` inside `redeemCollateral()`

Technical Details

The function `redeemCollateral()` allows a user to redeem collateral in exchange for debt assets (ReUSD). When calling the function, the `totalBorrow` is reduced by the amount of debt reimbursed minus a small fee.

Since the `totalBorrow` is reduced, saving the awaiting interests before the update is important as the ongoing interest rate needs to be applied on the initial `totalBorrow`. By reducing the variable without accounting for ongoing interests, when the interests are finally added on a future call, the `totalBorrow` used to determine the amount will be smaller than it should have been, and thus less interest will be added.

Impact

Medium. Interests added on the next call will be smaller than they should have been.

Recommendation

Add a call to `_addInterest()` at the beginning of the function.

Developer Response

Fixed in eac48b7b0a04e8a80abb9bc5cd1a5f8a41c5cbe1.

4. Medium - Disabling cooldown in GovStaker.sol bricks withdrawals

An incorrect condition would prevent withdrawals when the cooldown is disabled.

Technical Details

Unstaking in GovStaker.sol undergoes a cooldown period before tokens can be effectively withdrawn. The duration is controlled by the `cooldownEpochs` setting, which defines the `end` timestamp when queuing a new withdrawal.

```
        UserCooldown memory userCooldown = cooldowns[_account];
        userCooldown.end = uint104(block.timestamp + (cooldownEpochs * epochLength));
        userCooldown.amount += uint152(_amount);
        cooldowns[_account] = userCooldown;
```

The `unstake()` function checks this timestamp has been reached before sending the tokens to the user.

```
        if(block.timestamp < userCooldown.end || cooldownEpochs == 0) revert InvalidCooldown();
```

Note also that the condition requires `cooldownEpochs` to be different from zero; otherwise, the check will always revert.

Impact

Medium. Cooldown cannot be disabled without creating a denial of service in the unstaking process.

Recommendation

Likely, the intended condition would be `block.timestamp < userCooldown.end && cooldownEpochs != 0` so that setting `cooldownEpochs` to zero bypasses the timestamp check.

Developer Response

Recommendation implemented in 8d3637410905a1bdcaa0a0d68e62ebda010f1e24.

5. Medium - Debt won't be cleared if collateral balance is zero inside `distributeCollateralAndClearDebt()`

The function `distributeCollateralAndClearDebt()` will not clear the debt if there is no collateral available even though its purpose is to clear debt at the expense of insurance funds so the protocol doesn't have unbacked reUSD.

Technical Details

The function `distributeCollateralAndClearDebt()` is in charge of liquidating bad debt. This can happen if one of the vaults was exploited or lost funds for one reason or another; thus, its underlying value is less than the debt that was taken out from it.

Inside the `LiquidationHandler.sol` contract, the function `processCollateral()` can be called to redeem underlying and reduce the `debtByCollateral` mapping.

In the case of bad debt, it might have a `debtByCollateral` mapping greater than 0 for no collateral left since all have been redeemed already. In this case, if the owner tries to call the `distributeCollateralAndClearDebt()` function it will return early as there is a condition to return if no balance of collateral is found.

This is an issue as it means the bad debt cannot be socialized even though it's essentially the point of this function as it fully clears the debt and burns reUSD from insurance funds no matter the amount of collateral as long as it's greater than `1 wei`.

Impact

Medium. The bad debt will not be cleared if no collateral is available.

Recommendation

Don’t return if the collateral balance is zero; process usually instead, and don’t try to transfer collateral since there is none.

Developer Response

Fixed in 5a55dd9244f7b3897be4bd81e8ba9e8d58df8174.

6. Medium - Lack of funds in the insurance pool can block liquidations

The liquidation flow expects the insurance pool to have enough reUSD to cover the liquidated loan, or else the transaction will revert.

Technical Details

After a loan has been liquidated, LiquidationHandler.sol will process the seized collateral using `processCollateral()`.

```
    function processCollateral(address _collateral) public{
        require(IResupplyRegistry(registry).liquidationHandler() == address(this), "!liq handler");

        //get underlying
        address underlying = IERC4626(_collateral).asset();

        //try to max redeem
        uint256 withdrawnAmount;
        try IERC4626(_collateral).redeem(
            IERC4626(_collateral).maxRedeem(address(this)),
            insurancePool,
            address(this)
        ) returns (uint256 _withdrawnAmount){
            withdrawnAmount = _withdrawnAmount;
        } catch{}

        if(withdrawnAmount == 0) return;

        //debt to burn (clamp to debtByCollateral)
        uint256 toBurn = withdrawnAmount > debtByCollateral[_collateral] ? debtByCollateral[_collateral] : withdrawnAmount;
        IInsurancePool(insurancePool).burnAssets(toBurn);

        //update remaining debt (toBurn should not be greater than debtByCollateral as its adjusted above)
        debtByCollateral[_collateral] -= toBurn;

        emit CollateralProccessed(_collateral, toBurn, withdrawnAmount - toBurn);
    }
```

While a failed call to `redeem()` is handled by the surrounding `try/catch`, the burning of reUSD in the insurance pool might fail in line 115 if the funds held by the contract are lower than `toBurn`.

Impact

Medium. Liquidations cannot be executed if the insurance pool lacks funds.

Recommendation

Check if the insurance pool has enough tokens to cover the required burning amount and avoid the denial of service in the liquidation path. Debt will be saved in `debtByCollateral` and can be processed later when the pool is refilled.

Another option is to cap the debt decrease to the available amount of reUSD in the insurance pool to allow partial processing. However, additional logic will be required to determine the amount of collateral that should be sent.

Developer Response

Fixed in:

- df19a70e014af6597e551d5affd3d8ee6505d8e6
- 13c446fa1146131213b8e744b0703da98bde7a03
- d860a19b8ef4b0ba75ff2e5dae0df0424fd6d163

7. Medium - Depositing into the insurance funds doesn’t reset the withdrawal queue

Technical Details

When a user wants to exit the insurance pool they have to call the `exit()` function and wait 7 days before they can withdraw. During that period their liquidity can still be used for liquidations; they still earn liquidation fees but stop earning emissions of RSUP.

The issue is that when depositing, this cooldown is not reset, a user could start the cooldown period before depositing and then deposit during the withdrawal window so he can earn liquidation rewards without being locked in. In case of bad debt, he could withdraw before the bad debt is applied to the insurance pool while other users will still be locked.

Since the withdrawal window is only one day, the user could set up seven accounts/contracts that would start to cool down each day of the week and switch between them every day.

An advanced user could set up more accounts and then try to sandwich profitable liquidations by depositing and withdrawing at the beginning and end of the bundle. He would receive some of the profits of the liquidation fee while not risking being caught in a bad debt situation.

Impact

Medium. Users can deposit and withdraw without being locked; the downside is that they get no emissions.

Recommendation

Consider calling `_clearWithdrawQueue()` during deposits to clear the withdraw cooldown for the user.

Developer Response

Added require check: dbbeaed96fdd66efc0e8be47298dca66d219f441.

Low Findings

1. Low - Unsafe cast could be used to drain EmissionsController

A malicious actor can drain the EmissionsController.sol contract by using an unsafe type cast.

Technical Details

The `transferFromAllocation()` function allows a registered receiver to claim their portion of the allocated rewards.

```
209:     function transferFromAllocation(address _recipient, uint256 _amount) external returns (uint256) {
210:         if (_amount > 0) {
211:             allocated[msg.sender].amount -= uint200(_amount);
212:             govToken.transfer(_recipient, _amount);
213:         }
214:         return _amount;
215:     }
```

Before reducing the allocation, the implementation does an unsafe cast to `uint200` in line 211. An attacker could submit an `amount` such that `uint200(amount)` overflows to zero, bypassing the checks and allowing them to drain the contract.

Impact

Low. The issue would require the emissions contract to hold a large amount of tokens to trigger the overflow.

Recommendation

Use a safe cast library to sanitize the input argument.

Developer Response

Custom safe cast function implemented 83a1fc0cc3120a6f7a49f19280f47290dbb24125.

2. Low - Wrong check in `_calculateInterest()`

Technical Details

The function `_calculateInterest()` is in charge of computing the interests since the last contract call.

When adding the interests to the `totalBorrow`, it checks if there is no overflow risk by ensuring it doesn't go over `type(uint128).max`. However, it also checks that `interestEarned + borrowLimit <= type(uint128).max`, which is not needed as the `borrowLimit` is here only to limit borrowings and will not overflow because of the interests.

The risk is that if the `borrowLimit` was set to a value close to or greater than `type(uint128).max`, then interests will not accumulate.

Impact

Low. Useless check that might lead to interest not accumulating if the `borrowLimit` was set to a value close to or greater than `type(uint128).max`.

Recommendation

Remove this check.

Developer Response

Removed unnecessary check: 85052a1c24a1bbf5eeec845ac8be4d426669d728.

3. Low - Missing reentrancy protection in `GovStaker::exit()`

The `exit()` function, which handles reward distribution, is missing the `nonReentrant` modifier.

Technical Details

While the rewards claiming functions in the base MultiRewardsDistributor.sol contract are protected for reentrancy, the `exit()` function, which internally calls `_getRewardFor()`, is missing the `nonReentrant` modifier.

Impact

Low.

Recommendation

Add the `nonReentrant` modifier to the `exit()` function.

Developer Response

Recommendation implemented d34e0f6873066d97cf79bdd14b8217a8290fbd9a.

4. Low - Generic target is not checked in canceler payload

When checking if a proposal contains a payload to modify the operator permissions, the implementation of Voter.sol fails to account for the generic target `address(0)`.

Technical Details

The implementation of `_containsProposalCancelerPaylod()` inspects the proposal payload to determine if there is any call to the `Core::cancelProposal()` function. The intention here is to disallow `cancelProposal()` when the proposal includes a change over the permissions of this same function.

```
            if (action.target == address(core) && selector == ICore.setOperatorPermissions.selector) {
                bytes memory slicedData = new bytes(data.length - 4); // create new byte array that excludes the selector
                // copy the data to slicedData byte by byte, excluding the selector
                for (uint256 j = 0; j < slicedData.length; j++) {
                    slicedData[j] = data[j + 4];
                }
                (, address target, bytes4 permissionSelector, , ) = abi.decode(slicedData, (address, address, bytes4, bool, address));
                if (target == address(this) && permissionSelector == ICore.cancelProposal.selector) {
                    require(payloadLength == 1, "Payload with canceler must be single action");
                    return true;
                }
            }
```

After decoding the arguments, the implementation checks that the target is the voter contract and that the selector matches the cancel proposal function. However, operator permissions in `execute()` also support a generic way of targeting any account by using `address(0)`.

```
    function execute(address target, bytes calldata data) external returns (bytes memory) {
        if (msg.sender == voter) return target.functionCall(data);
        bytes4 selector = bytes4(data[:4]);
        OperatorAuth memory auth = operatorPermissions[msg.sender][address(0)][selector];
        if (!auth.authorized) {
            auth = operatorPermissions[msg.sender][target][selector];
        }
```

Impact

Low. If the approval was initially set for a generic target, the operator can override any change by canceling the proposal.

Recommendation

Check also for the generic target (`address(0)`).

```diff
    (, address target, bytes4 permissionSelector, , ) = abi.decode(slicedData, (address, address, bytes4, bool, address));
-   if (target == address(this) && permissionSelector == ICore.cancelProposal.selector) {
+   if ((target == address(this) || target == address(0)) && permissionSelector == ICore.cancelProposal.selector) {
        require(payloadLength == 1, "Payload with canceler must be single action");
        return true;
    }
```

Developer Response

Recommendation implemented 379814969c06d5d531e5fdeb33010bdaf055a9de.

5. Low - Impossibility to reset the `pid` back to 0

Technical Details

When initialized, the `ResupplyPair.sol` contract can take a `pid`. This allows the contract to deposit the collateral into a Convex vault to receive CRV and CVX tokens.

It is possible to put zero as `pid` to ask the contract not to deposit/withdraw collateral; however, this is only at initialization. After that the `pid` can only be changed to a valid `pid` (different than 0) as the contract will always try to deposit the collateral into the new `pid` when calling `setConvexPool()` which will likely revert if the `pid` is zero as the vault connected to it is for cDai/cUsdc curve pool only.

Impact

Low. Impossibility to reset the `pid` to 0 and turn off deposit/withdraw of collateral into a Convex pool.

Recommendation

Add a condition if `pid == 0` then don't try to deposit into the Convex vault inside `_updateConvexPool()`.

Developer Response

Acknowledged.

6. Low - Possible wrong transfer amount inside `distributeCollateralAndClearDebt()`

Technical Details

The function `distributeCollateralAndClearDebt()` allows the owner of the `LiquidationHandler.sol` contract to force distribute the collateral available to the insurance fund and burn the outstanding debt even if it's more than the collateral distributed.

This allows the protocol to socialize the debt at the expense of insurance fund depositors and burn the debt so the protocol doesn't have unbacked debt standing.

However, when doing so the function first computes the balance of collateral available on the contract then tries to process the collateral available by calling `processCollateral()` which is going to redeem some of that collateral for underlying and reduce the debt by the amount redeemed. Then, finally, it sends the collateral balance to the insurance fund.

The issue is that the balance computed before the call to `processCollateral()` might be incorrect when used to transfer the collateral at the end of the function since some of it may have been redeemed inside `processCollateral()`.

This will lead the function to revert and thus might block the contract owner from clearing the debt. One way to not get into that issue is for the owner to first call `processCollateral()` then call `distributeCollateralAndClearDebt()` so the balance doesn't get reduced during the second call.

Impact

Low. If there is collateral to be redeemed, then the function will revert and not clear the debt.

Recommendation

Compute the balance after the internal call to `processCollateral()`.

Developer Response

Fixed in 39035d6ee4e23be95e52b861429e04a47f39adf5.

7. Low - Second split of creation code is not cleared in ResupplyPairDeployer.sol

The second part of the creation code split is not cleared in the deployer contract, causing a potential conflict if the updated version is shorter.

Technical Details

In `setCreationCode()`, the pair's creation code is split in two if it's longer than 13000 bytes.

```
    function setCreationCode(bytes calldata _creationCode) external onlyOwner{
        bytes memory _firstHalf = BytesLib.slice(_creationCode, 0, 13_000);
        contractAddress1 = SSTORE2.write(_firstHalf);
        if (_creationCode.length > 13_000) {
            bytes memory _secondHalf = BytesLib.slice(_creationCode, 13_000,
_creationCode.length - 13_000);
            contractAddress2 = SSTORE2.write(_secondHalf);
        }
    }
```

The second part of the split is stored using SSTORE2 in `contractAddress2`. However, this variable remains uncleared if the code is smaller than the threshold. This could cause an accidental mixture of different code versions.

Impact

Low. The issue manifests only when the code's length is less than ~13kb.

Recommendation

Clear the `contractAddress2` variable when `_creationCode` is not split.

```
    function setCreationCode(bytes calldata _creationCode) external onlyOwner{
        bytes memory _firstHalf = BytesLib.slice(_creationCode, 0, 13_000);
        contractAddress1 = SSTORE2.write(_firstHalf);
        if (_creationCode.length > 13_000) {
            bytes memory _secondHalf = BytesLib.slice(_creationCode, 13_000,
_creationCode.length - 13_000);
            contractAddress2 = SSTORE2.write(_secondHalf);
-       }
+       } else {
+           contractAddress2 = address(0);
+       }
    }
```

Also, make sure to skip the second part if it's null.

```
    function _deploy(
        bytes memory _configData,
        bytes memory _immutables,
        bytes memory _customConfigData
    ) private returns (address _pairAddress) {
        // Get creation code
-       bytes memory _creationCode = BytesLib.concat(SSTORE2.read(contractAddress1),
SSTORE2.read(contractAddress2));
+       bytes memory _creationCode = SSTORE2.read(contractAddress1);
+       if (contractAddress2 != address(0)) {
+           _creationCode = BytesLib.concat(_creationCode,
SSTORE2.read(contractAddress2));
+       }
```

Developer Response

Fixed in 024c4b144ee49ff6761d7bf51a68ef202fce462b.

8. Low - Incorrect implementations in RedemptionHandler.sol

The `getMaxRedeemableCollateral()` and `getMaxRedeemableUnderlying()` functions return incorrect values.

Technical Details

The implementation of `getMaxRedeemableCollateral()` fetches the pair's exchange rate to calculate the redeemable amount in collateral (vault shares).

```
    function getMaxRedeemableCollateral(address _pair) public view returns(uint256){
        (,,uint256 exchangeRate) = IResupplyPair(_pair).exchangeRateInfo();
        if (exchangeRate == 0) return 0;
        return getMaxRedeemableValue(_pair) * PRECISION / exchangeRate;
    }
```

There are two issues here. First, `exchangeRateInfo()` returns the cached version of the exchange rate, which could be stale. Second, the conversion is incorrect since `exchangeRate` is the factor to convert from underlying to shares; it should multiply and not divide.

Similarly, the intention in `getMaxRedeemableUnderlying()` seems to inform the amount of underlying when redeeming the collateral shares through the vault (i.e., when `_redeemToUnderlying = true`), which is missing a call to `previewRedeem()` to project shares to underlying assets.

```
    function getMaxRedeemableUnderlying(address _pair) public view returns(uint256){
        uint256 maxCollat = getMaxRedeemableCollateral(_pair);
        address vault = IResupplyPair(_pair).collateral();
        uint256 maxWithdraw = IERC4626(vault).maxWithdraw(_pair);

        return maxWithdraw > maxCollat ? maxCollat : maxWithdraw;
    }
```

Impact

Low.

Recommendation

In the case of `getMaxRedeemableCollateral()`, the implementation can call `updateExchangeRate()`, but it needs to be transformed to mutable, else it would need some kind of non-mutable variant to preview the exchange rate.

```
-   function getMaxRedeemableCollateral(address _pair) public view returns(uint256){
+   function getMaxRedeemableCollateral(address _pair) public returns(uint256){
-       (,,uint256 exchangeRate) = IResupplyPair(_pair).exchangeRateInfo()
+       uint256 exchangeRate = IResupplyPair(_pair).updateExchangeRate();
        if (exchangeRate == 0) return 0;
-       return getMaxRedeemableValue(_pair) * PRECISION / exchangeRate;
+       return getMaxRedeemableValue(_pair) * exchangeRate / PRECISION;
    }
```

For `getMaxRedeemableUnderlying()`:

```
    function getMaxRedeemableUnderlying(address _pair) public view returns(uint256){
        uint256 maxCollat = getMaxRedeemableCollateral(_pair);
        address vault = IResupplyPair(_pair).collateral();
        uint256 maxWithdraw = IERC4626(vault).maxWithdraw(_pair);
+       uint256 maxCollatUnderlying = IERC4626(vault).previewRedeem(maxCollat);

-       return maxWithdraw > maxCollat ? maxCollat : maxWithdraw;
+       return maxWithdraw > maxCollatUnderlying ? maxCollatUnderlying : maxWithdraw;
    }
```

Developer Response

Fixed in 2ffacb5993e3b1ded700b26f7268f59f3c7decbd.

9. Low - `claimableFees` is shadowed in `getPairAccounting()`

An overlooked variable shadowing makes the return value of `_claimableFees` to be always zero in `getPairAccounting()`.

Technical Details

```
127:     function getPairAccounting()
128:         external
129:         view
130:         returns (
131:             uint256 _claimableFees,
132:             uint128 _totalBorrowAmount,
133:             uint128 _totalBorrowShares,
134:             uint256 _totalCollateral
135:         )
136:     {
137:         (, , uint256 _claimableFees, VaultAccount memory _totalBorrow) = previewAddInterest();
138:         _totalBorrowAmount = _totalBorrow.amount;
139:         _totalBorrowShares = _totalBorrow.shares;
140:         _totalCollateral = totalCollateral();
141:     }
```

The `_claimableFees` variable declaration in line 137 shadows the return value with the same name.

Impact

Low. The return value of `_claimableFees` is incorrect.

Recommendation

Remove the declaration that shadows the return variable.

```diff
    function getPairAccounting()
        external
        view
        returns (
            uint256 _claimableFees,
            uint128 _totalBorrowAmount,
            uint128 _totalBorrowShares,
            uint256 _totalCollateral
        )
    {
-       (, , uint256 _claimableFees, VaultAccount memory _totalBorrow) =
previewAddInterest();
+       VaultAccount memory _totalBorrow;
+       (, , _claimableFees, _totalBorrow) = previewAddInterest();
        _totalBorrowAmount = _totalBorrow.amount;
        _totalBorrowShares = _totalBorrow.shares;
        _totalCollateral = totalCollateral();
    }
```

Developer Response

Fixed in 05d0765b12687cbfd6f6eea65a382cbf92d8a188.

10. Low - Missing debt accrual in `totalDebtAvailable()`

The total amount of debt is used in the `totalDebtAvailable()` calculation without considering pending interests.

Technical Details

The implementation of `totalDebtAvailable()` uses `totalBorrow` without first checkpointing the debt.

```
    function totalDebtAvailable(
    ) public view returns (uint256) {
        //check for max mintable. on mainnet this shouldnt be limited but on l2 there could
        //be a limited amount of stables that have been bridged and available
        uint256 mintable = block.chainid == 1 ? type(uint256).max : IResupplyRegistry(registry).getMaxMintable(address(this));
        uint256 borrowable = borrowLimit > totalBorrow.amount ? borrowLimit - totalBorrow.amount : 0;
        //take minimum of mintable and the difference of borrowlimit and current borrowed
        return borrowable < mintable ? borrowable : mintable;
    }
```

Impact

Low. Results can be slightly incorrect due to pending interests.

Recommendation

Call `previewAddInterest()` to get an updated version of `totalBorrow`. Note that `totalDebtAvailable()` is used internally in `_borrow()`, which already accrues the debt using a previous call to `_addInterest()`. Considering this, it would be convenient to refactor the implementation using an internal variant that receives the updated `totalBorrow` structure.

```
function totalDebtAvailable() external view returns (uint256) {
    (,,, VaultAccount memory _totalBorrow) = previewAddInterest();

    return _totalDebtAvailable(_totalBorrow);
}

function _totalDebtAvailable(VaultAccount memory _totalBorrow) internal view returns (uint256) {
    //check for max mintable. on mainnet this shouldnt be limited but on l2 there could
    //be a limited amount of stables that have been bridged and available
    uint256 mintable = block.chainid == 1 ? type(uint256).max : IResupplyRegistry(registry).getMaxMintable(address(this));
    uint256 borrowable = borrowLimit > _totalBorrow.amount ? borrowLimit - _totalBorrow.amount : 0;
    //take minimum of mintable and the difference of borrowlimit and current borrowed
    return borrowable < mintable ? borrowable : mintable;
}
```

Developer Response

Updated at e1e55181f07ad255ff50d7990fa6aedea585585b.

11. Low - Add interest before updating the `RateCalculator`

Technical Details

The function `setRateCalculator()` can be called by the owner to change the `RateCalculator` contract in charge of returning the interest rate.

If the logic of the new contract is going to return a different interest rate then it would be more fair for users to save the interests accumulated before the update by calling `_addInterest()`.

Impact

Low.

Recommendation

Call `_addInterest()` at the beginning of the function before the variable update.

Developer Response

Fixed in [aa877ae1c51e3d985b8c8f97b80e74ecf6154029](aa877ae1c51e3d985b8c8f97b80e74ecf6154029).

12. Low - Possible double allocation in `EmissionController`

Technical Details

The `EmissionController` is in charge of distributing emissions. Receivers can be set by calling the function `registerReceiver()`.

When the first receiver is added, his id will be 0. There is a special case in the function for when the id is 0:

- The receiver weight will be set to 100%.
- The receiver `lastAllocEpoch` will be set to 0 instead of the current epoch.

By setting `lastAllocEpoch` to 0, all past and current epoch distributions will distribute 100% of the emissions to the newly added receiver.

The function `_mintEmissions()` is in charge of minting tokens and setting unallocated tokens. When there are no receivers, all rewards are set to the variable `unallocated`, which can then be claimed by the owner.

This can lead to double distribution if the contract is deployed and the first receiver is not set before `_mintEmissions()` is called by a call to `setEmissionsSchedule()`. then the `unallocated` variable will receive the emissions, later on when the first receiver is added it will also receive all past and current emissions. Resulting in emissions being allocated twice.

Impact

Low. Emissions could be allocated twice if no receiver is set before emissions start.

Recommendation

Consider removing the `unallocated` variable since the first receiver will receive all past emissions or not setting `lastAllocEpoch` to 0 but to the current epoch instead for the first receiver.

Developer Response

Decided to make the call to `_mintEmissions()` conditional on whether > 0 receivers exists. Decided against the suggestion to remove unallocated as it would lead to lost emissions when a receiver becomes disabled following epoch(s) in which it did not get pinged.

Fixed in 8d86c1a1bd549b291a9a9096bf7b3fc195353e1f and b29e609e31d5545772fde23e50da75655859bbeb.

13. Low - No cooldown period when updating the LTV

Technical Details

The function `setMaxLTV()` can be called by the owner to change the current max LTV.

If the TVL is reduced, some positions could become liquidable. If the users are not given proper notice before the change, it could lead to unexpected liquidations.

Most lending markets implement a cooldown for changing LTVs, with the new LTV directly taking effect for new borrowers while older borrowers have a given duration to update their positions before the new LTV takes effect for them.

Impact

Low.

Recommendation

Implement a `pendingLTV` variable that will be used for new borrows, then slowly transition `maxLTV` to it over a given duration (e.g. 7 days).

Developer Response

Acknowledged.

There is a voting period to warn in the government layer. We can also do something gradual on the gov side if some pairs were to have their LTV updated. There are other possible options as well, such as temporarily setting the liquidation fee to zero and atomically closing insolvent positions.

14. Low - Unsafe swap deadline

Swaps in `leveragedPosition()` and `repayWithCollateral()` use an unsafe argument for the deadline.

Technical Details

Both functions swap tokens using `block.timestamp` as the deadline, which would allow execution at any time, defeating its purpose.

Impact

Low.

Recommendation

Move this argument as user input and forward it to `swapExactTokensForTokens()`.

Developer Response

Deadline is removed as it is not used in curve swaps. Changed interface of swapper as well as directly mint reUSD to the given swapper to skip on the approve. PR#34.

15. Low - Default swappers cannot be added from the registry contract

The ResupplyRegistry.sol contract doesn't permit setting swappers on pairs.

Technical Details

The `addPair()` function sets a predefined list of swappers when Pairs are registered.

```
        // Set additional values for ResupplyPair
        IResupplyPair _pair = IResupplyPair(_pairAddress);
        address[] memory _defaultSwappers = defaultSwappers;
        for (uint256 i = 0; i < _defaultSwappers.length; i++) {
            _pair.setSwapper(_defaultSwappers[i], true);
        }
```

However, the `setSwapper()` function is restricted to the owner of the Pair, which is the Core.sol contract.

Impact

Low. Default swappers functionality cannot be used.

Recommendation

Allow the registry to access the `setSwapper()` function.

Developer Response

Added at 8a381e39d3100e864205ef748ad22efbbe8f02e1.

Gas Saving Findings

1. Gas - Gas savings in `ResupplyPairCore.sol`

Technical Details

There are some gas savings possible inside `ResupplyPairCore.sol`:

- No need to set `currentRateInfo.lastTimestamp = uint64(0);` on line 181 as it will be 0 by default.
- `_isInterestUpdated` can be set to `true` directly instead of `_isInterestUpdated = _results.isInterestUpdated;` as it will always be true inside the condition on line 523.
- `_syncUserRedemptions()` could return the `_userCollateralBalance` to save extra storage loads when the collateral balance needs to be synced then used like in `userCollateralBalance()`.

- Use `_userBorrowShares` instead of `userBorrowShares()` in both place it's called inside the codebase which are `_isSolvent()` and `isSolvent()` since in both case there is a checkpoint prior earlier in the function.
- `liquidate()` could be called directly by the user since there is no extra logic inside LiquidationManager's `liquidate()` function.

Impact

Gas.

Recommendation

Apply the changes suggested.

Developer Response

Added some at a2fea7554dd06ad916d80df6a3534bffa2f87503. Will just acknowledge the sync one. Liquidate is extrapolated so we can add things in the middle if needed such a stipend to call.

2. Gas - Gas savings in `Voter.sol`

Technical Details

There are some gas savings possible inside `Voter.sol`:

- Could remove `if (epoch == 0) return 0;` as `createNewProposal()` has a `require(epoch > 0)`.
- `_containsProposalCancelerPaylod` has a typo.
- The payload can be sliced easily with `slicedData = data[4:data.length]` instead of using a loop on line 279.
- `require(pctYes <= MAX_PCT && pctNo <= MAX_PCT, "Pct must not exceed MAX_PCT");` is not needed as the second `require` will ensure the sum is equal to `MAX_PCT`.

Impact

Gas.

Recommendation

Apply the changes suggested.

Developer Response

Fixed in:

- 23b632fbef6d01969e8d612d71d178544621a26d
- f6112084adec33667ac1a68e9647abf2fd4f6d39
- 5286a842506f90cbf9200d02276b7e52db465678
- 593026f943ed87456e069ba3b156eb2368a3bd0a

3. Gas - Gas savings in `EmissionController.sol`

Technical Details

There are some gas savings possible inside `EmissionController.sol`:

- Call `_fetchEmissions()` instead of `IReceiver(receiver.receiver).allocateEmissions();` as the result is the same on line 141.
- Check `if (i == _rates.length - 1) break;` inside the `for` statement above it directly on line 320.

Impact

Gas.

Recommendation

Apply the changes suggested.

Developer Response

Fixed in:

- b3758c67bb5b5ee843e0dbc842faf245be62e60f
- aa1706d9a9940cc787333727d009cffbd878a1f6

Informational Findings

1. Informational - Unused function in `SimpleRewardStreamer.sol`

Technical Details

The contract `SimpleRewardStreamer.sol` has an empty function `user_checkpoint()` that is not used anywhere and can't be used since its body is empty.

Impact

Informational.

Recommendation

Remove the function.

Developer Response

Acknowledged.

2. Informational - Natspec not updated/incorrect/missing for some functions

Technical Details

Some functions from the original Frax Lend codebase were modified, but their natspec wasn't.

- The `constructor()` is incorrect in the ResupplyPairCore.sol.
- `RemoveCollateral` event is missing `_borrower`.
- Some new contracts like InsurancePool.sol don't have natspec.

Impact

Informational.

Recommendation

Update natspec.

Developer Response

Fixed in 0f0559d160d3e2b22aab5cafefd44323a2fd969b and 3914081768dfd0ea4a3e0b0dc39b71306a50ac2b.

3. Informational - Division by 0 in `currentUtilization()` when the pair is paused

Technical Details

The function `currentUtilization()` returns the current pair utilization in comparison to the `borrowLimit`. When the protocol is paused the `borrowLimit` is set to 0.

Because of that, when calculating the utilization, the function will divide by 0, which will make it revert.

Impact

Informational.

Recommendation

Consider early returning 100% when `borrowLimit == 0`.

Developer Response

Added in 739322a317c991501645f5903e982ecaee77de7d.

4. Informational - Collateral should not be allowed as a reward

The collateral token should be restricted when rewards are added to the pairs.

Technical Details

The implementation of `_checkAddToken()` allows any token to be added as a reward. If enabled, this could result in users' deposited collateral being distributed as rewards.

```
    function _checkAddToken(address _address) internal view override returns(bool){
        return true;
    }
```

Impact

Informational.

Recommendation

Filter the collateral token in `_checkAddToken()`.

Developer Response

Added collateral check at c61eaf459aa1e741d61c50ed682dc43856817582.

5. Informational - Unused variables

Technical Details

Multiple variables are not used throughout the codebase:

- `currentRateInfo.ratePerSec` in `ResupplyPairCore` is not used and shouldn't be used to display rates on the UI as it may be outdated. The UI should use `IRateCalculator(rateCalculator).getNewRate()`.
- `owner` parameter is not used in `redeem()` and `withdraw()` inside `InsurancePool`; since the parameter is needed to match the interface, it is possible to keep `address` and remove the name.
- `_pair` and `_amount` are not used in `getRedemptionFeePct()` inside `RedemptionHandler`. Similar to the above, consider removing the name of the parameters but keep the types in case these are used in future versions.
- `splits.platform` is not used inside `feeDepositController`.
- `underlying` is not used in `processCollateral()` inside `LiquidationHandler`.
- `oldStaker` storage variable in PermaLocker
- `receiverPlatform` and `receiverInsurance` in FeeDeposit
- `defaultSwappers` in ResupplyPairDeployer
- `circuitBreakerAddress` in ResupplyRegistry

Impact

Informational.

Recommendation

Consider removing variables and removing parameter names.

Developer Response

Clean up at 0f0559d160d3e2b22aab5cafefd44323a2fd969b and 593f874d7f186d0b99173de8bc192d9fc1656737.

6. Informational - The protocol assumes the underlying has 18 decimals

There are several occurrences in which the protocol defaults the underlying asset decimals to 18. Even though this is aligned with FRAX and crvUSD, it would not work for the general case.

Technical Details

While the implementation of ResupplyPairCore.sol checks that the collateral has 18 decimals, there is no assurance that the underlying asset also uses the same number of decimals.

Several places in the codebase assume the underlying asset has 18 decimals, such as the BasicVaultOracle.sol contract that expects the result of `IERC4626(_vault).convertToAssets(1e18)` to be scaled to 18 decimals, or the calculation of the exchange rate.

Impact

Informational.

Recommendation

Consider adding an explicit check or documenting that underlying assets with decimals different than 18 are not supported.

Developer Response

Not sure is really needed but went ahead and added check to underlying to be decimals of 18 at 8d13393b7ad6843819901cf7e58d033ba25cffa2.

7. Informational - Missing safe ERC20 wrapping

Several instances of ERC20 approvals or transfers are not wrapped with SafeERC20.

Technical Details

- ResupplyPair.sol#L83
- ResupplyPairCore.sol#L179
- ResupplyPairCore.sol#L1234
- RewardDistributorMultiEpoch.sol#L103

Impact

Informational.

Recommendation

Ensure all instances of `approve()`, `transfer()` and `transferFrom()` are wrapped with SafeERC20.

Developer Response

Acknowledged.

8. Informational - Interest rates are not strictly calculated by epoch

Interest rates from Pairs used to determine their weight in the emission distribution depend on when fees are flushed and are not strictly bucketed by epoch.

Technical Details

The `withdrawFees()` function mints pending fees and calls the `incrementPairRevenue()` function, which calculates the new interest rate and updates the weight in the pair emission stream.

The only guarantee during the process is that `withdrawFees()` can be called at most once per epoch, which means the pending fees are not truly aligned with epochs and could either span multiple epochs or include portions of different epochs.

Impact

Informational.

Recommendation

Ensure `withdrawFees()` is called synchronously at the start of each epoch for all pairs so that weights in the emission stream are relatively fair.

Developer Response

Acknowledged. This is not an issue for us.

9. Informational - Misleading `claimableEmissions()` implementation

The implementation of SimpleReceiver.sol returns the currently allocated amount to the receiver, without considering any distributed amounts that are technically claimable but pending allocation.

Technical Details

```
    function claimableEmissions() external view returns (uint256) {
        (, uint256 allocated) = emissionsController.allocated(address(this));
        return allocated;
    }
```

Impact

Informational.

Recommendation

Consider renaming the function or documenting the behavior.

Developer Response

Docs added here: 921c2f9ecbd057499a16e4f9a6c7a6c7de2a4f7e.

10. Informational - Voting power can be used after unstake

Depending on the cooldown settings, voting power can still be exercised after tokens have been unstaked.

Technical Details

Voter.sol works by looking up voting power at the epoch before the current one. Once tokens are realized in the GovStaker.sol contract, these can be used to propose governance actions and vote on those proposals.

A user can exit the staker at a certain epoch while still holding the corresponding voting power to exercise actions in the voter. Depending on the cooldown setting, the user can even withdraw these tokens, allowing them to propose and vote on negative actions against the DAO without being exposed to their consequences.

Impact

Informational.

Recommendation

Consider keeping a cooldown value that at least covers the duration of any active proposals associated with the voting power being removed, ensuring stakers cannot withdraw their tokens before these proposals conclude.

Developer Response

Acknowledged. This is expected behavior. Agree with your point about cooldown.

11. Informational - Ensure owner matches caller in InsurancePool.sol `redeem()` and `withdraw()`

The contract implements an interface similar to ERC4626 but ignores the `owner` argument in `redeem()` and `withdraw()`.

Technical Details

The implementation of these functions follows the same interface as the ERC4626 specification but ignores the `owner` argument and assumes the redemption will always be executed on the caller, creating a potential confusion with the standard behavior due to its similarities.

Impact

Informational.

Recommendation

Since the contract doesn't follow the ERC4626 standard, the `owner` argument can be safely removed. Alternatively, consider adding a check to avoid any potential confusion.

```
     function withdraw(uint256 _amount, address _receiver, address _owner) public
nonReentrant returns(uint256 shares){
+        require(_owner == msg.sender);
         _checkWithdrawReady(msg.sender);
```

Developer Response

Added in a1aa0250d763708b4a6a9e01904c2f2f00a053c4.

12. Informational - Limited usage of Protocol pausing in `Core.sol`

Technical Details

The contract `Core.sol` has a storage variable `paused` used by `CorePausable.sol`, `GuardianAuthHook.sol`, and in `GuardianOperator.sol`.

However, these contracts use this variable in a very limited way:

- `CorePausable.sol` has modifiers `whenProtocolNotPaused()` and `whenProtocolPaused()` but they are not used in the codebase.
- `GuardianAuthHook.sol` just uses it to block unpause of the protocol if it was paused in the `preHook`.
- `GuardianOperator.sol` just uses it to block the pausing of the protocol if it's already paused.

Impact

Informational. The protocol pausing doesn't affect the DAO or pairs.

Recommendation

Remove this functionality or check if the protocol is paused at the DAO and pair level.

Developer Response

We have opt'ed against having an additional protocol level pause functionality, in favor of pause locally in pairs. Thus, I have:

- removed protocol-level pausing
- removed `GuardianOperator.sol`
- removed `GuardianAuthHook.sol`
- remove `CorePausable.sol`
- added tests on `Core.t.sol`

Commit 2fb1203e8a726f08f7c1638278bd5012e773af7a.

13. Informational - Repay with collateral will be challenging to estimate off-chain

The repayment must be calculated with precision, or else the operation could overflow or fail due to the required minimum debt.

Technical Details

The `repayWithCollateral()` function swaps user collateral for stablecoins to repay the loan.

The resulting output amount from the swap is translated to borrow shares, and the user's debt is reduced based on this number of shares. As the implementation uses `swapExactTokensForTokens()`, if the user wants to clear their debt, it will be very difficult to estimate off-chain the required input amount so that the resulting output does not overflow the shares (i.e. results in more shares than owed by the user) or falls short to fully cover the debt, reverting due to the required minimum amount of debt.

Impact

Informational.

Recommendation

The implementation of `repayWithCollateral()` could take a number of shares to repay, calculate the amount of stablecoins, and then swap collateral using `swapTokensForExactTokens()`.

Developer Response

Added a clamp to user borrow shares and send back leftover stables in efacb0d32bcee57e54eafa137afdc0e2d25bff4a and 7b12a79ce488e7aa7d609844e29079cfd926334e.

14. Informational - Missing or incorrect events

Technical Details

Missing events:

- EmissionsController.sol#L317
- MultiRewardsDistributor.sol#L168
- VestManager.sol#L119
- FeeDepositController.sol#L79

Incorrect events:

- The shares and assets arguments are swapped in the `Withdraw` event in `redeem()` and `withdraw()`
- In `_addCollateral()` the sender argument is incorrect when called from `addCollateral()` or `leveragedPosition()`

- In `_removeCollateral()` the receiver argument is incorrect when called from `removeCollateral()` or `repayWithCollateral()`

Impact

Informational.

Recommendation

Add or fix the listed occurrences.

Developer Response

Fixed in:

- a1aa0250d763708b4a6a9e01904c2f2f00a053c4
- 0eec47e8e1b23e8711a573822a6969ef8c4a6a29
- VestManager.sol#L126

15. Informational - Account argument is ignored in some InsurancePool.sol functions

Technical Details

The implementation of `_clearWithdrawQueue()` mixes the `_account` argument with the caller (lines 213 and 225).

```
212:     function _clearWithdrawQueue(address _account) internal{
213:         if(withdrawQueue[msg.sender] != 0){
214:             //checkpoint rewards
215:             _checkpoint(_account);
216:             //get reward 0 info
217:             RewardType storage reward = rewards[0];
218:             //note how much is claimable
219:             uint256 reward0 = claimable_reward[reward.reward_token][_account];
220:             //reset claimable
221:             claimable_reward[reward.reward_token][_account] = 0;
222:             //redistribute back to pool
223:             reward.reward_remaining -= reward0;
224:
225:             withdrawQueue[msg.sender] = 0; //flag as not waiting for withdraw
226:         }
227:     }
```

Similarly, `_checkWithdrawReady()` ignores the `_account` argument and uses `msg.sender`.

Impact

Informational.

Recommendation

Replace `msg.sender` with the `_account` argument.

Developer Response

Updated at d5a0db1e29ef0f7207c0ac68284fda9c7e5a6026.

Final remarks

The Resupply Protocol is an innovative solution designed to enhance Frax and Curve USD liquidity, potentially extending its benefits to other protocols in the future. While straightforward in its design, the codebase introduces complexity due to its scale, reward mechanisms, and lending components. Although the lending functionality is forked from Frax, modifications, including redemptions and the insurance pool, required a comprehensive assessment.

Although it is a sizeable codebase, its modularity and decentralization features are notable strengths, particularly in the DAO aspect. The protocol implements a fully fledged governance solution that provides clear tokenomics and voting mechanisms for protocol management.

Given the number and severity of findings, auditors strongly recommend conducting extensive testing, including fuzzing and invariant testing, and securing an additional review before moving to production. The team demonstrated exceptional efficiency in addressing identified issues and responding to auditor inquiries, which made the collaboration a positive experience.