SB SECURITY

RootsFi

Security Review

April 15, 2025

Contents

1. About SBSecurity

SBSecurity is a duo of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

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

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

- **High** - almost **certain** to happen, easy to perform, or highly incentivized.
- **Medium** - only **conditionally possible**, but still relatively likely.
- **Low** - requires specific state or **little-to-no incentive**.

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

RootsFi contracts have been audited through the Hyacinth platform.

Audit Disclaimer: Our review found that, although much of the code is forked and familiar, a number of issues still persist. We recommend a follow-up audit and thorough testing prior to deployment.

Overview

Project	RootsFi
Repository	Private
Commit Hash	3709b32bf1a80e119350ac54291386196f17ccce
Resolution	ebecf4a8a8635479444f0b2584f78ac2ad2b79b1
Timeline	Audit: March 27, 2025 - April 4, 2025 Mitigation: April 4, 2025 - April 15, 2025

Scope

BeraAdapter.sol
BorrowerOperations.sol
DebtToken.sol
EmissionScheduler.sol
Factory.sol
LiquidationManager.sol
PriceFeed.sol
RootsCore.sol
SortedTroves.sol

StabilityPool.sol

Staker.sol

TroveManager.sol

BexPriceFeed.sol

Issues Found

Critical Risk	0
High Risk	3
Medium Risk	7
Low/Info Risk	10

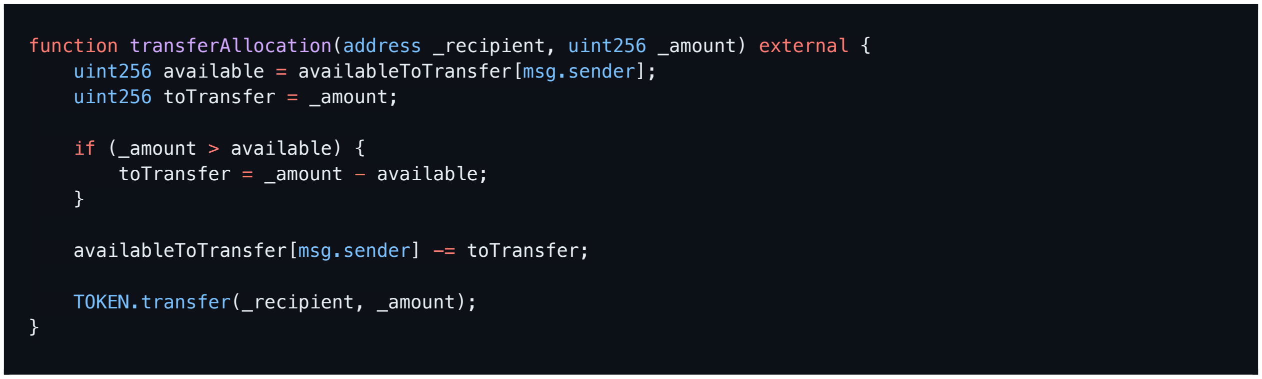
5. Findings

5.1. High severity

5.1.1. Wrong allocation transfer will block reward claims

Severity: High Risk

Description: When the allocation for a specific week is transferred to the recipient in `EmissionScheduler.transferAllocation()`, when the amount passed is different from `availableToTransfer`, the calculation will mess `availableToTransfer` and will transfer a different amount of tokens.



```
function transferAllocation(address _recipient, uint256 _amount) external {
    uint256 available = availableToTransfer[msg.sender];
    uint256 toTransfer = _amount;

    if (_amount > available) {
        toTransfer = _amount - available;
    }

    availableToTransfer[msg.sender] -= toTransfer;

    TOKEN.transfer(_recipient, _amount);
}
```

Simple example is:

- `amount` = 60

- `available` = 50

- `toTransfer` = 60 - 50 = 10

Then 10 will be removed from the `availableToTransfer`, but 60 will be transferred.

Recommendation: If the idea is to transfer only the available amount, `available` should be reassigned to `_amount` in the if clause and then used to subtract from `availableToTransfer` and transfer tokens.

Resolution: Fixed

5.1.2. Wrong index clears collateral mappings of the next collateral token instead

Severity: High Risk

Description: When collateral is enabled via enableCollateral(), the collateral index (indexByCollateral mapping) is set to 1 index after the actual index in the collateralTokens array. The first collateral has index = 1, despite being in the 0th index.

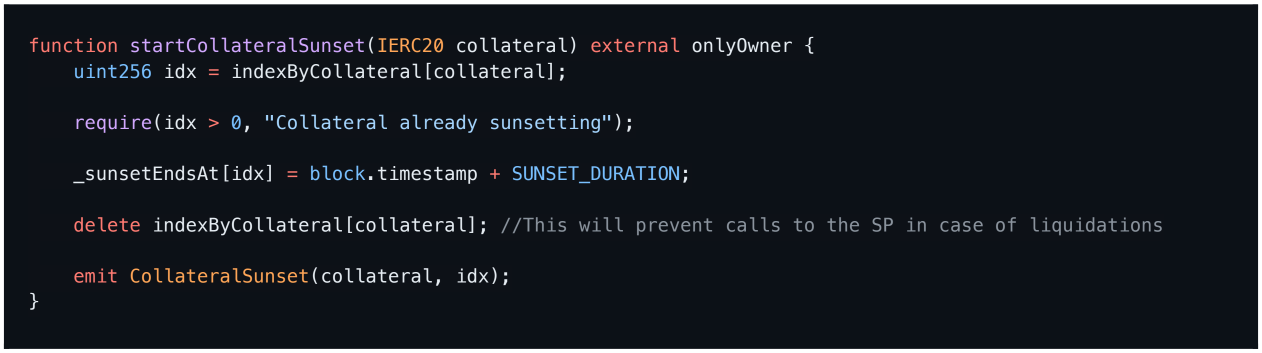
Then, when offset() is called for the collateral. The index will be obtained from indexByCollateral, reduced by 1 to get the true index in the collateralTokens array, and then update the rewards inside _updateRewardSumAndProduct() for it, specifically setting data to the epochToScaleToSums mapping.

```
function _offset(IERC20 collateral, uint256 _debtToOffset, uint256 _collToAdd) internal {
    require(msg.sender == liquidationManager, "StabilityPool: Caller is not Liquidation Manager");
    uint256 idx = indexByCollateral[collateral];
    idx -= 1;

    uint256 totalDebt = totalDebtTokenDeposits; // cached to save an SLOAD
    if (totalDebt == 0 || _debtToOffset == 0) {
        return;
    }

    _triggerRewardIssuance();

    (uint256 collateralGainPerUnitStaked, uint256 debtLossPerUnitStaked) =
        _computeRewardsPerUnitStaked(_collToAdd, _debtToOffset, totalDebt, idx);

    _updateRewardSumAndProduct(collateralGainPerUnitStaked, debtLossPerUnitStaked, idx); // updates S and P

    // Cancel the liquidated Debt debt with the Debt in the stability pool
    _decreaseDebt(_debtToOffset);
}
```

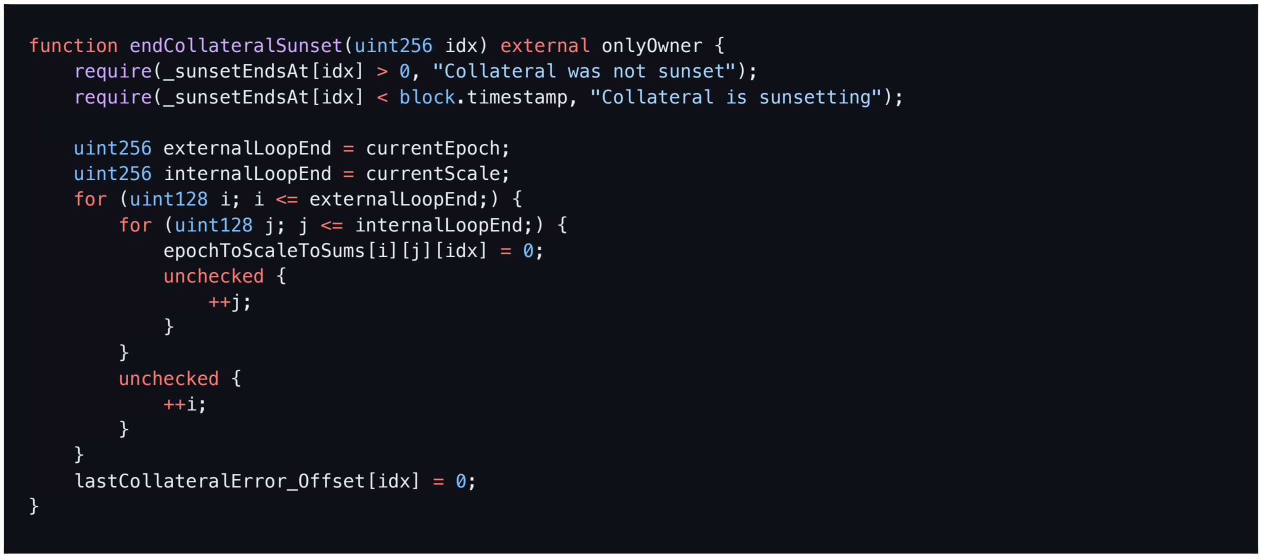
```
function _updateRewardSumAndProduct(
    uint256 _collateralGainPerUnitStaked,
    uint256 _debtLossPerUnitStaked,
    uint256 idx
) internal {
    uint256 currentP = P;
    uint256 newP;

    /*
     * The newProductFactor is the factor by which to change all deposits, due to the depletion of
Stability Pool Debt in the liquidation.
     * We make the product factor 0 if there was a pool-emptying. Otherwise, it is (1 -
DebtLossPerUnitStaked)
     */
    uint256 newProductFactor = uint256(DECIMAL_PRECISION) - _debtLossPerUnitStaked;

    uint128 currentScaleCached = currentScale;
    uint128 currentEpochCached = currentEpoch;
    uint256 currentS = epochToScaleToSums[currentEpochCached][currentScaleCached][idx];

    /*
     * Calculate the new S first, before we update P.
     * The collateral gain for any given depositor from a liquidation depends on the value of their deposit
     * (and the value of totalDeposits) prior to the Stability being depleted by the debt in the
liquidation.
     *
     * Since S corresponds to collateral gain, and P to deposit loss, we update S first.
     */
    uint256 marginalCollateralGain = _collateralGainPerUnitStaked * currentP;
    uint256 newS = currentS + marginalCollateralGain;
    epochToScaleToSums[currentEpochCached][currentScaleCached][idx] = newS;
    emit S_Updated(idx, newS, currentEpochCached, currentScaleCached);

    // If the Stability Pool was emptied, increment the epoch, and reset the scale and product P
    if (newProductFactor == 0) {
        currentEpoch = currentEpochCached + 1;
        emit EpochUpdated(currentEpoch);
        currentScale = 0;
        emit ScaleUpdated(currentScale);
        newP = DECIMAL_PRECISION;

        // If multiplying P by a non-zero product factor would reduce P below the scale boundary, increment
the scale
    } else if ((currentP * newProductFactor) / DECIMAL_PRECISION < SCALE_FACTOR) {
        newP = (currentP * newProductFactor * SCALE_FACTOR) / DECIMAL_PRECISION;
        currentScale = currentScaleCached + 1;
        emit ScaleUpdated(currentScale);
    } else {
        newP = (currentP * newProductFactor) / DECIMAL_PRECISION;
    }

    require(newP > 0, "NewP");
    P = newP;
    emit P_Updated(newP);
}
```

The problem is that later, when `startCollateralSunset()` is called for the collateral, the index will be retrieved from the `indexByCollateral` mapping again, but it will not be decremented. In this case, 1 will be set for index, not 0, inside `_sunsetEndsAt`. `endCollateralSunset()` is the next function that should reset `epochToScaleToSums` for this collateral, but it will use 1 index above compared to the one used when calling `offset()`.

This will delete data for the wrong collateral and mess up the entire rewards system.



```solidity
function startCollateralSunset(IERC20 collateral) external onlyOwner {
    uint256 idx = indexByCollateral[collateral];

    require(idx > 0, "Collateral already sunsetting");

    _sunsetEndsAt[idx] = block.timestamp + SUNSET_DURATION;

    delete indexByCollateral[collateral]; //This will prevent calls to the SP in case of liquidations

    emit CollateralSunset(collateral, idx);
}
```



```solidity
function endCollateralSunset(uint256 idx) external onlyOwner {
    require(_sunsetEndsAt[idx] > 0, "Collateral was not sunset");
    require(_sunsetEndsAt[idx] < block.timestamp, "Collateral is sunsetting");

    uint256 externalLoopEnd = currentEpoch;
    uint256 internalLoopEnd = currentScale;
    for (uint128 i; i <= externalLoopEnd;) {
        for (uint128 j; j <= internalLoopEnd;) {
            epochToScaleToSums[i][j][idx] = 0;
            unchecked {
                ++j;
            }
        }
        unchecked {
            ++i;
        }
    }
    lastCollateralError_Offset[idx] = 0;
}
```

Recommendation: Inside `startCollateralSunset()` lower the index at the beginning and then use it.

Resolution: Fixed

5.1.3. BPT oracle will return wrong and easy-manipulated price

Severity: High Risk

Description: `BexPriceFeed` is relying on the spot balances of the pool asset, thus making it easy to manipulate and also to return the wrong price. Here are the major mistakes that were observed:

1. Usage of the `totalSupply` - BEX (forked from BalancerV2) uses `getActualSupply` as an indicator of the net BPT tokens in circulation, `totalSupply` will return lower value since it doesn't include the unminted BPT that are being accumulated and will be added after join/exit operation.

2. Using `(token0Tvl + token1Tvl).divWadDown(BEX_POOL.totalSupply())` to find the BPT price - this is an spot price calculation that can be manipulated easily.

3. Not including the token weights in the price calculation - value weight ratio of the token must be fetched in order to get to the right price.

4. Not using the weighted pool invariant - a BPT weighted pool allows adding and removing liquidity not only proportionally, but also in non-proportional or "unbalanced" ways. An unbalanced add or remove liquidity can be considered a combination of a proportional add or remove plus a swap. Thus, simply summing the pool token TVLs will be returning wrong price.

All these inconsistencies will lead to miscalculation of the BPT token price, which can lead to all sorts of issues across the codebase, from unfair liquidation to opening Troves under their “real” CR below MCR.

Recommendation: Consider rewriting the entire price feed functionality, by following these resources:

1. Example implementation: https://etherscan.io/address/0x00463c21f4fad709717879e4bc13e4e2ca80e7ac#code

2. Balancer pricing guides:

- https://hackmd.io/@re73/SJHmQaCFq

- https://docs-v2.balancer.fi/concepts/advanced/valuing-bpt/valuing-bpt.html#informational-price-evaluation

Resolution: Fixed

5.2. Medium severity

5.2.1. rewardRate reset if weekly amount scheduled is smaller than 604800

Severity: Medium Risk

Description: Inside `_triggerRewardIssuance`, the rewardRate for the week is calculated based on the amount allocated. If the amount allocated is less than 1 week is seconds as a number (604800), the reward rate will be 0. This is a possible case for an expensive token with 6 decimals.

```
function _triggerRewardIssuance() internal {
    _updateG(_vestedEmissions());

    uint256 _periodFinish = periodFinish;
    uint256 lastUpdateWeek = (_periodFinish - startTime) / 1 weeks;
    // If the last claim was a week earlier we reclaim
    if (getWeek() >= lastUpdateWeek) {
        uint256 amount = scheduler.claimAllocation();
        if (amount > 0) {
            // If the previous period is not finished we combine new and pending old rewards
            if (block.timestamp < _periodFinish) {
                uint256 remaining = _periodFinish - block.timestamp;
                amount += remaining * rewardRate;
            }
            rewardRate = uint128(amount / REWARD_DURATION);
            periodFinish = uint32(block.timestamp + REWARD_DURATION);
        }
    }
    lastUpdate = uint32(block.timestamp);
}
```

Recommendation: For reward tokens with less decimals and extremely expensive tokens, make sure each weekly allocation is above 604800.

Resolution: Acknowledged

5.2.2. emissions for the week are locked if `_updateG` returns early

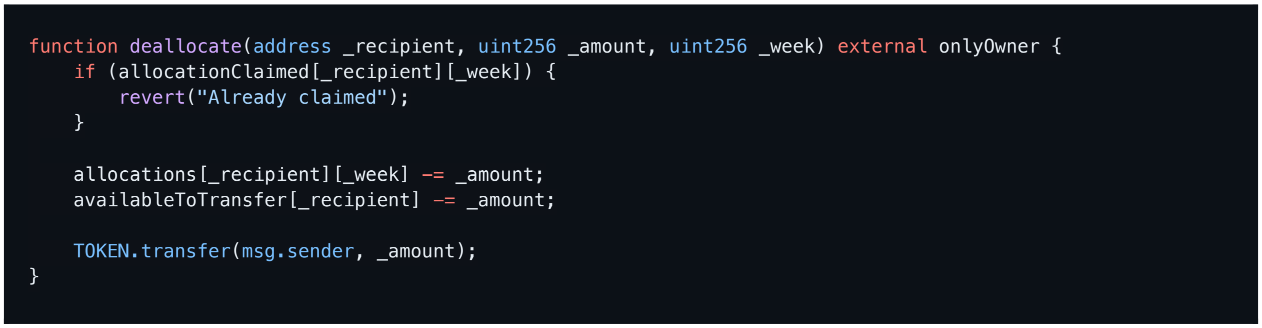
Severity: Medium Risk

Description: `_triggerRewardIssuance` is relying on the `_updateG` to distribute the new gain based on the updated P, but the problem is it can return early if `totalDebt` or `_rewardsIssuance` are 0.

```
function _updateG(uint256 _rewardsIssuance) internal {
    uint256 totalDebt = totalDebtTokenDeposits; // cached to save an SLOAD
    /*
     * When total deposits is 0, G is not updated. In this case, the Prisma issued can not be obtained by
later
     * depositors - it is missed out on, and remains in the balanceof the Treasury contract.
     *
     */
    if (totalDebt == 0 || _rewardsIssuance == 0) {
        return;
    }

    uint256 rewardsPerUnitStaked;
    rewardsPerUnitStaked = _computeRewardsPerUnitStaked(_rewardsIssuance, totalDebt);
    uint128 currentEpochCached = currentEpoch;
    uint128 currentScaleCached = currentScale;
    uint256 marginalPrismaGain = rewardsPerUnitStaked * P;
    uint256 newG = epochToScaleToG[currentEpochCached][currentScaleCached] + marginalPrismaGain;
    epochToScaleToG[currentEpochCached][currentScaleCached] = newG;

    emit G_Updated(newG, currentEpochCached, currentScaleCached);
}
```

```
function _triggerRewardIssuance() internal {
    _updateG(_vestedEmissions());

    uint256 _periodFinish = periodFinish;
    uint256 lastUpdateWeek = (_periodFinish - startTime) / 1 weeks;
    // If the last claim was a week earlier we reclaim
    if (getWeek() >= lastUpdateWeek) {
        uint256 amount = scheduler.claimAllocation();
        if (amount > 0) {
            // If the previous period is not finished we combine new and pending old rewards
            if (block.timestamp < _periodFinish) {
                uint256 remaining = _periodFinish - block.timestamp;
                amount += remaining * rewardRate;
            }
            rewardRate = uint128(amount / REWARD_DURATION);
            periodFinish = uint32(block.timestamp + REWARD_DURATION);
        }
    }
    lastUpdate = uint32(block.timestamp);
}
```

The problem is when new week has started and allocations are marked as claimed in `EmissionScheduler::claimAllocation`, then these tokens will be locked forever, since they're added towards the `StabilityPool` gains and also cannot be deallocated, because there's a check which prevents `deallocating` weeks that are already marked as claimed.



```
function deallocate(address _recipient, uint256 _amount, uint256 _week) external onlyOwner {
    if (allocationClaimed[_recipient][_week]) {
        revert("Already claimed");
    }

    allocations[_recipient][_week] -= _amount;
    availableToTransfer[_recipient] -= _amount;

    TOKEN.transfer(msg.sender, _amount);
}
```

Recommendation: In the `if` statement, consider returning to the owner the weekly allocation or just adding it to the next week. If you chose the second approach, still there should be a function that makes it possible to claim the tokens as in the worst case the stability pool is not used anymore.

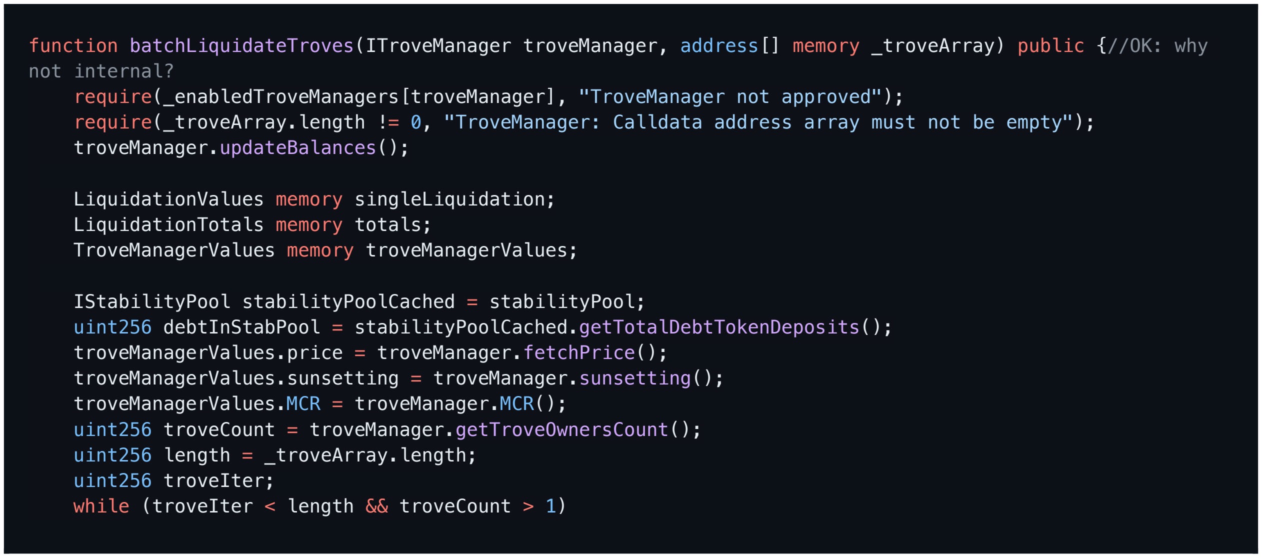
Resolution: Acknowledged

5.2.3. Last, 0th indexed trove cannot be liquidated

Severity: Medium Risk

Description: Liquidation mechanism, which is a modified version of `Liquity`, makes it impossible to remove the last, 0th indexed trove. That means, unless he is not willing to close his trove no one else can. Indeed someone can open a trove himself with a higher CR, liquidate the last one, and close. In no other way, the last liquidation can happen, due to the following while conditions:

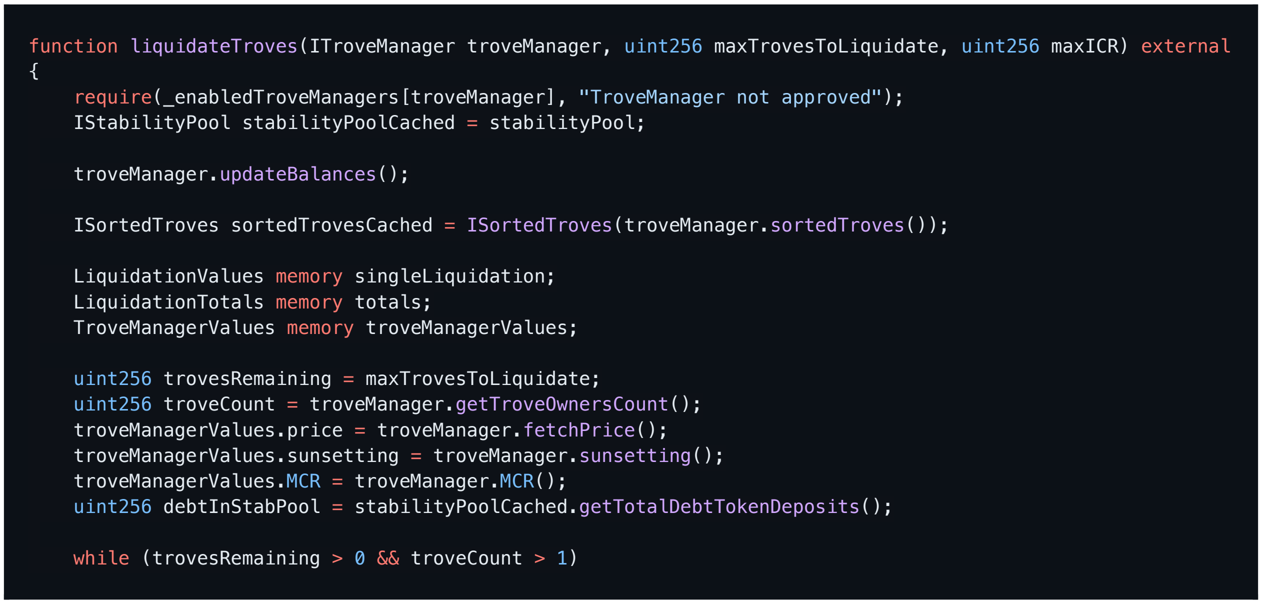
- In `batchLiquidateTroves`, `troveIter` is the internal `while` tracking variable, `troveCount` is the total count of Troves, 1st one is incremented, 2nd one is decremented at the end of the loop



```
function batchLiquidateTroves(ITroveManager troveManager, address[] memory _troveArray) public {//OK: why
not internal?
    require(_enabledTroveManagers[troveManager], "TroveManager not approved");
    require(_troveArray.length != 0, "TroveManager: Calldata address array must not be empty");
    troveManager.updateBalances();

    LiquidationValues memory singleLiquidation;
    LiquidationTotals memory totals;
    TroveManagerValues memory troveManagerValues;

    IStabilityPool stabilityPoolCached = stabilityPool;
    uint256 debtInStabPool = stabilityPoolCached.getTotalDebtTokenDeposits();
    troveManagerValues.price = troveManager.fetchPrice();
    troveManagerValues.sunsetting = troveManager.sunsetting();
    troveManagerValues.MCR = troveManager.MCR();
    uint256 troveCount = troveManager.getTroveOwnersCount();
    uint256 length = _troveArray.length;
    uint256 troveIter;
    while (troveIter < length && troveCount > 1)
```

For the last trove, `troveCount` will be equal to 1, which will evaluate the `while` condition to false and won't lead to liquidation.

- In `liquidateTroves`, `trovesRemaining` is liquidator-controlled variable, `troveCount` is the total count of Troves, here both are decremented at the end of the loop.



```
function liquidateTroves(ITroveManager troveManager, uint256 maxTrovesToLiquidate, uint256 maxICR) external
{
    require(_enabledTroveManagers[troveManager], "TroveManager not approved");
    IStabilityPool stabilityPoolCached = stabilityPool;

    troveManager.updateBalances();

    ISortedTroves sortedTrovesCached = ISortedTroves(troveManager.sortedTroves());

    LiquidationValues memory singleLiquidation;
    LiquidationTotals memory totals;
    TroveManagerValues memory troveManagerValues;

    uint256 trovesRemaining = maxTrovesToLiquidate;
    uint256 troveCount = troveManager.getTroveOwnersCount();
    troveManagerValues.price = troveManager.fetchPrice();
    troveManagerValues.sunsetting = troveManager.sunsetting();
    troveManagerValues.MCR = troveManager.MCR();
    uint256 debtInStabPool = stabilityPoolCached.getTotalDebtTokenDeposits();

    while (trovesRemaining > 0 && troveCount > 1)
```

For the last trove, `trovesRemaining` will be equal to X, as it can be controlled, while `troveCount` will be equal to 1 which will evaluate the `while` condition to false and won't lead to liquidation.

Recommendation: Either add restrictions in `closeTrove` not to allow closing the last trove, as it's done in Liquity, or make it possible the liquidation to be executed for it, by modifying the `while` loop conditions.

Resolution: Acknowledged

5.2.4. Sunsetting will enable the interestRate

Severity: Medium Risk

Description: Roots is intended to work without any interest rate, implementation which originates from LiquityV1. The problem is that the additional sunsetting functionality will enable interestRate, despite it being 0 before that.

```
function startSunset() external onlyOwner {
    sunsetting = true;
    _accrueActiveInterests();
    interestRate = SUNSETTING_INTEREST_RATE;
    // accrual function doesn't update the timestamp if interest is 0
    lastActiveIndexUpdate = block.timestamp;
    redemptionFeeFloor = 0;
    maxSystemDebt = 0;
}
```

As we see, it will be set to 50%, even though the old interestRate was 0%.

This will impact all the pending debts by increasing their debts, eventually leading to liquidation after the _applyPendingRewards function increases the debt of the troves:

```
if (troveInterestIndex < currentInterestIndex) {
      debt = (debt * currentInterestIndex) / troveInterestIndex;
      t.activeInterestIndex = currentInterestIndex;
}
```

Since InterestRate is 0 and troveInterestIndex will be SUNSETTING_INTEREST_RATE, after sunset, currentInterestIndex will become greater than SUNSETTING_INTEREST_RATE, which will enter the if branch, leading to 50% interest applied to all the debts.

Recommendation: Consider removing all the functionality regarding the interest rates.

Resolution: Acknowledged. Roots team acknowledged the issue and explained that interest must be activated when sunsetting to enforce troves to be closed. An important note is to warn the users prior to starting the sunset, so they can be ready to close troves or increase collateral/decrease debt

5.2.5. `BexPriceFeed` is easily manipulatable

Severity: Medium Risk

Description: Multiple inconsistencies in the feed can make it prone to manipulation by interested parties.

1. Centralized price update mechanism - unlike UniV3's TWAP, where each interaction with the pool adds a new observation, here we have designated actors who are responsible for adding price readings. This can be exploited by continuously sandwiching the price updates, by manipulating the `BEX token balances` and `totalSupply`. By doing that the attacker can fill the price reading mapping with inflated or deflated prices and at a certain point in time move the TWAP in his favor. This can be used to lower the collateral ratios of the borrowers, so they can be liquidated or break the peg mechanism of the `$MEAD` token.

2. High gas consumption if longer `twapTimestampInSeconds` is given - also unlike UniV3 where ticks are used to fetch average price, here `for-loop` is used, which is tracking all the price updates back in time, up to the `twapTimestampInSeconds`. If we want to take the TWAP price for 30 minutes back and we assume price was updated in each block (~3 second block time for Berachain), we'll end up with 600 iterations.If that's executed in conjunction with `LiquidationManager::liquidateTroves`, in times of high volatility and a lot of `Troves` for liquidation this can exceed the block gas limit, terminating the protocol and giving the time for users to close their unhealthy troves (assuming that gas limit is enough for close but insufficient for mass liquidation) this can harm the peg of the `$MEAD` token.

3. No liquidity guarantee - if the given BEX pair experiences sharp liquidity spikes, TWAP will be lagging and will create discrepancies between the actual LP token price and the one reported from the `BexPriceFeed`, which can naturally lead to the concerns mentioned in pt 1.

Recommendation: Since there's no way currently to price the different BEX LP tokens, that will be used as collateral, there's no other simple oracle approach. Furthermore, this is what Berachain recommends.

If we want to have a more stable price feed the following actions might be needed:

1. Price updated from private RPCs, this will make them harder to be sandwiched but won't completely remove the danger.

2. Distribute the price registration tasks across multiple trusted parties.

3. Perform extensive testing for price reliability and adjust the parameters of the `BexPriceFeed`, so it can guarantee maximal price confidence.

4. Do not use pools with low liquidity, this will make it more prone to `BEX token balances` and `totalSupply` manipulations, where the attacker changes the ratios of the tokens.

5. Perform an economic review on the impact of the price manipulation.

6. Implement short-circuit mechanisms that can dynamically adjust the feed's params, by evaluating the BEX Pool.

Here is an article on the different Oracle considerations for these types of projects - https://www.liquity.org/blog/the-oracle-conundrum.

Resolution: Acknowledged

5.2.6. PriceFeed blocked if Pyth haven't updated for > 1 minute

Severity: Medium Risk

Description: The call to `PYTH.getPriceNoOlderThan` is hardcoded to always be with a 1 minute timeframe, this could be a problem if the price hasn't been updated in the last 1 minute as the function will revert.

This will happen when more stable asset is used, whose price hasn't been moved more than the defined deviation threshold for the sponsored feeds. If that happens, all the functions in Roots will be blocked, as they're relying on the price.

Recommendation: Similar to `twapTimestampInSeconds`, consider making this timeframe also variable and able to be changed with setter.

Resolution: Fixed

5.2.7. BexPriceFeed not compatible with some pool types

Severity: Medium Risk

Description:

1. When using stable pools, getActualSupply() should be used instead of totalSupply() as the pools have pre-minted LP tokens.

https://docs.bex.berachain.com/developers/contracts/lp_tokens/valuing#stable-pools

That's the case for:

- Dinero/WBERA - https://berascan.com/address/0x2461e93d5963c2bb69de499676763e67a63c7ba5#readContract (this one is weighted, but still has differences b/n totalSupply and actualSupply)
- USDC.e/HONEY - https://berascan.com/address/0xf961a8f6d8c69e7321e78d254ecafbcc3a637621#readContract

Since totalSupply will give a higher number, this will make the LP token price in the BexPriceFeed lower than the actual, which will open arbitrage opportunities.

2. Other non-compatible pools are the ones that can dynamically change their token order and add/remove tokens. See note in getPoolTokens:

For majority of the pools and the more stable ones - registerTokens is called only at the initialization, but if a pool with this functionality is used, this will mess the price in Roots, since the token order has been cached.

Recommendation: Add a check based on a bool value when using getActualSupply() and when totalSupply(). For the 2nd issue, if you insist on supporting such pairs, make sure to dynamically get the token orders from the BEXVault contract, instead of caching them.

Resolution: Fixed

5.3. Low severity

5.3.1. DebtToken's flashFee function is not ERC3156 compatible

Severity: Low Risk

Description: According to the EIP3165, `flashFee` must revert instead of returning 0 if the token is not supported. However, currently `DebtToken::flashFee` returns 0.

The `flashFee` function MUST return the fee charged for a loan of `amount token`. If the token is not supported `flashFee` MUST revert.

Recommendation: Modify the function to revert if non-supported token is given.

Resolution: Acknowledged

5.3.2. pyth expo is not guaranteed to return negative number all the time

Severity: Low Risk

Description: expo is used for both the price and confidence interval. By taking a look at the `Pyth` onchain code, we'll see that it's not hardcoded in the feeds, but instead of that passed with each price update.

However, it's not guaranteed to 100% be only a negative number, as there's no validation in Pyth itself. If there's a price update with a positive exponent, `BexPriceFeed` will compute the wrong prices, since it relies on the number always being negative.

Despite not having a concrete example of a +ve expo, in theory, it's possible. Below is the result of 1308 price feeds and their exponents:

Recommendation: Add check to restrict processing -ve exponents, as shown in the Pyth CrossSwap example.

Resolution: Acknowledged

5.3.3. stabilityPool can be blocked if a lot of collaterals are added

Severity: Low Risk

Description: Since `StabilityPool` is originally written for only 1 collateral token, the additionally added logic introduces multiple `for-loops` across the entire contract. Extensive testing and care should be taken when adding new collaterals, in order not to increase gas consumption, potentially leading to out-of-gas issues. Even though there has to be a large count of tokens, it's still not impossible.

Recommendation: No, code changes are needed, simply make sure to perform testing and simulations before adding each new collateral.

Resolution: Acknowledged

5.3.4. low-level transfer can restrict gnosis wallets from interacting with the `BeraAdapter`

Severity: Low Risk

Description: `BeraAdapter.withdrawColl()`, `adjustTrove()` and `recoverBera()` are limited to working with EOA users only as it uses a `transfer` for the native Berachain token.

Recommendation: Consider using the `call` to allow smart wallets and contracts to interact with the contract.

Resolution: Acknowledged

5.3.5. Missing `address(0)` checks

Severity: Low Risk

Description: Sanity checks are missing across the projects, that make the functions prone to input errors.

- `BorrowerOperations::initialize – address(0)` checks

- `Factory::initialize – address(0)` checks

- `Factory::setImplementations – address(0)` checks

- `StabilityPool::initialize – address(0)` checks

- `Staker::initialize – address(0)` checks

- `RootsCore::setFeeReceiver – address(0)` checks

- `RootsCore::setGuardian – address(0)` checks

- `RootsCore::setPriceFeed – address(0)` checks

- `TroveManager::setAddresses – address(0)` checks

Recommendation: Add zero checks for all addresses.

Resolution: Acknowledged

5.3.6. Unsafe ERC20 operations used

Severity: Low Risk

Description: Non-safe ERC20 operations are used in the following files/functions:

- `EmissionScheduler.sol`

- `Staker::onWithdrawal()`

- `Staker::initialize()`

- `Staker::setGauge()`

- `TroveManager::updateTroveFromAdjustment()`

- `TroveManager::openTrove()`

Recommendation: Replace all normal ERC20 operations with their safe substitutes from SafeERC20.

Resolution: Acknowledged

5.3.7. Precision loss in the reward rate calculation

Severity: Low Risk

Description: Dust amount between 1-604799 will be left in the `EmissionScheduler`, due to the `rewardRate` division done in `StabilityPool::_triggerRewardIssuance`:

Recommendation: Add recover function in the `EmissionScheduler`.

Resolution: Acknowledged

5.3.8. `collateralGainsByDepositor` can round if prolonged time passes without being claimed

Severity: Low Risk

Description: If the collateral is low at price ($0.0001 for example) and the gains are not claimed regularly, `collateralGainsByDepositor` can overflow since it uses `uint80` and this is around 1.2 million tokens.

Recommendation: Change the value type to bigger uint.

Resolution: Acknowledged

5.3.9. `collateralGainsByDepositor` can round if prolonged time passes without being claimed

Severity: Low Risk

Description: Division, performed in the `_findTwapPrice` is prone to precision loss due to the nature of the calculations.

Example:

- Setup:
    - Price readings:
        - price[0] = 1.2345e18 (1.2345 tokens) at timestamp = 1000.
        - price[1] = 2.5e18 (2.5 tokens) at timestamp = 1010.
    - block.timestamp = 1020, _timespan = 20 (look back to 1000).
- Calculation:
    - sumPriceWeight = (1.2345e18 * 10) + (2.5e18 * 10) = 12.345e18 + 25e18 = 37.345e18.
    - sumTimeWeight = 10 + 10 = 20.
    - True TWAP = 37.345e18 / 20 = 1.86725e18 (1.86725 tokens).
    - Solidity: 37.345e18 / 20 = 1.8672e18 (1.8672 tokens, truncated).
- Precision Loss:
    - Loss = 1.86725e18 - 1.8672e18 = 0.00005e18 (0.00005 tokens, or 5e13 wei).

Recommendation: Use the `divWadDown` from the `FixedPointMathLib`, but beware that the intermediate multiplication can revert if `sumPriceWeight * 1e18` exceeds `uint256.max`.

Resolution: Acknowledged

5.3.10. Unused structs not removed from the code

Severity: Low Risk

Description: Remove the unused structs:

- StabilityPool.queue
- TroveManager.emissionId

Resolution: Acknowledged