SHERLOCK

Security Review For Harbor

Collaborative Audit Prepared For: **Harbor**
Lead Security Expert(s): **givn**
vinica_boy
Date Audited: **July 2 - July 20, 2025**

Introduction

Harbor is a decentralized platform for creating and trading synthetic assets – tokens that mirror the value of real-world or digital assets without requiring direct custody of those assets.

Scope

Repository: baofinance/bao-base

Audited Commit: 35e593ca98fe873dea7253482b18ea536598a7c2

Final Commit: 2016ba72c99dc5a0c2cc46cd3f715160ca8a63d4

Files:

- src/BaoOwnableRoles.sol
- src/BaoOwnableRoles_v2.sol
- src/BaoOwnable.sol
- src/BaoOwnableTransferrableRoles.sol
- src/BaoOwnableTransferrable.sol
- src/BaoOwnable_v2.sol
- src/Deployed.sol
- src/ERC165.sol
- src/internal/BaoCheckOwner.sol
- src/internal/BaoCheckOwner_v2.sol
- src/internal/BaoRoles.sol
- src/internal/BaoRoles_v2.sol
- src/MintableBurnableERC20_v1.sol
- src/PermittableERC20_v1.sol
- src/Stem_v1.sol
- src/TokenHolder.sol
- src/Token.sol

Repository: baofinance/bao-minter

Audited Commit: fd24d3326f43ec115eb8c06e5c64a019aa5a4434

Final Commit: ffd56246dd27fb449d5b15c54f9a0c6328bdd495

Files:

- src/math/DecrementalFloatingPoint.sol
- src/minter/Genesis_v1.sol
- src/minter/library/ConfigIncentiveLib.sol
- src/minter/library/Config_v1.sol
- src/minter/Minter_v1.sol
- src/minter/ReservePool_v1.sol
- src/minter/StabilityPoolManager_v1.sol
- src/minter/StabilityPool_v1.sol
- src/minter/TokenDistributor_v1.sol
- src/price/PriceOracle_v1.sol
- src/price/StakedETHWrappedPriceOracle_v1.sol
- src/reward/accumulator/MultipleRewardCompoundingAccumulator.sol
- src/reward/distributor/LinearMultipleRewardDistributor.sol
- src/reward/distributor/LinearReward.sol
- src/reward/steam/Steam_v1.sol
- src/reward/voting-escrow/VotingEscrow_v1.sol
- src/util/WordCodec.sol

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
4	9	6

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Incorrect accounting for underlying collateral during leveraged token redeem

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/36

Summary

In a leveraged token redemption operation, the leveraged token is burned from the user, and collateral is withdrawn from the protocol. However, instead of subtracting the withdrawn collateral, collateral is added in the underlying calculations.

Vulnerability Detail

The `_redeemLeveragedAdjustments()` function calculates the amount of leveraged tokens to be taken from the user and the corresponding collateral to be returned. This calculation is performed per band, as the operation reduces the collateral ratio, and different bands have varying incentive fees.

In each loop, we calculate the amount of collateral withdrawn, which should be subtracted to accurately reflect the collateral available for the next band. However, it is incorrectly added, allowing the redemption of more leveraged tokens than permitted and disrupting the per-band calculation.

Impact

Incorrect accounting for the available collateral.

Code Snippet

Minter::_redeemLeveragedAdjustments()

Tool Used

Manual Review

Recommendation

`collateralInBand` should be subtracted from `collateralTokenBalance`.

Issue H-2: Gauge's claimReward() functionality can be abused leading to stuck tokens

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/37

Summary

The `LiquidityGaugeV6::claim_rewards()` function allows a third party to call it on behalf of the Stability Pool. Reward tokens are still transferred to the Stability Pool, but they are not accounted for, causing them to remain stuck in the contract.

This can be abused by a malicious actor and thus SP stakers wont get any rewards coming from the gauge.

Impact

Lost rewards for SP stakers.

Code Snippet

LiquidityGaugeV6::claim_rewards()

Tool Used

Manual Review

Recommendation

Consider changing the `claim_rewards()` implementation to disallow third party claiming on behalf of the SP.

Issue H-3: Users dont get any compensation in case of a full liquidation of Stability Pool

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/38

Summary

When the SP is used to rebalance the CR of the protocol, users get collateral/leveraged token (based on the SP type) in return for their staked pegged token. However, in case all the deposited pegged tokens in a SP are used, users wont get any tokens in return.

Vulnerability Detail

The `StabilityPoolManager::rebalance()` function is used to sweep pegged tokens from the Stability Pool and accumulate the corresponding reward tokens. A user's share of the rewards is based on their stake in the Stability Pool. However, when pegged tokens are swept, the users' stakes are essentially zero due to the epoch change, resulting in no pre-liquidation depositors receiving their corresponding rewards.

Apart from the liquidation rewards, pre-liquidation depositors would lose the gauge rewards which are released over time.

Impact

Users lose their pegged tokens and get 0 rewards in return.

Tool Used

Manual Review

Recommendation

As discussed with the team, one option is to accumulate rewards before sweeping pegged tokens from the Stability Pool, which would resolve the liquidation rewards issue but not the gauge rewards problem.

A more comprehensive solution is to redesign the Stability Pool to ensure a minimum amount of pegged tokens remains deposited at all times. This should apply to every operation–deposit, withdrawal, and liquidation. This way, users' balances will not drop to zero from a single ”full” liquidation, and rewards can be accurately accounted for.

Issue H-4: Harvest can be frontran to steal rewards from other depositors

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/51

Summary

Users can deposit pegged tokens in stability pools. One of the incentives for doing so is that they will receive a portion of the `Minter` yield that the underlying collateral in generates. The issue is that these rewards are accounted for instantly and an attacker can deposit, harvest and then withdraw deposit + rewards.

Vulnerability Detail

The `StabilityPoolManager_v1` has the following function, which can be called by anyone:

```
function harvest(
    address bountyReceiver,
    uint256 minBounty
) external nonReentrant returns (uint256 harvestedAmount)
```

The amount of underlying collateral accrued fetched on this line: `uint256 harvestableAmount = IMinter(MINTER).harvestable();`

If the pools hold pegged tokens, they will get underlying collateral distributed:

```
IERC20(WRAPPED_COLLATERAL_TOKEN).safeTransfer(pool, harvestedAmount);
IStabilityPool(pool).accumulateReward(WRAPPED_COLLATERAL_TOKEN, harvestedAmount);
```

Accounting is done with the current contract values in the StabilityPool, allowing proportional distribution to all depositors:

```
function _accumulateReward(address token, uint256 amount) internal virtual override
 ↪  {
        //...
        uint48 epochExponent = currentProd.epochAndExponent();
        uint256 magnitude = currentProd.magnitude();

        uint192 integral = $.tokenToEpochExponentToIntegral[token][epochExponent];
        integral += (uint192((amount * _REWARD_PRECISION) / totalShare) *
         ↪  uint192(magnitude));
        $.tokenToEpochExponentToIntegral[token][epochExponent] = integral;
    }
```

Impact

Harvest rewards can be stolen by frontrunning pegged token deposit, harvesting and then withdrawing the pegged tokens and wstETH rewards.

Code Snippet

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.28 <0.9.0;

import {console2 as console} from "forge-std/console2.sol";
import {Vm} from "forge-std/Vm.sol";

import {Deployed} from "@bao/Deployed.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IBaoOwnable} from "@bao/interfaces/IBaoOwnable.sol";
import {IBaoRoles} from "@bao/interfaces/IBaoRoles.sol";
import {IBurnable2Arg} from "@bao/interfaces/IBurnable2Arg.sol";
import {Minter_v1} from "src/minter/Minter_v1.sol";
import {MintableBurnableERC20_v1} from "@bao/MintableBurnableERC20_v1.sol";
import {StabilityPoolManager_v1} from "src/minter/StabilityPoolManager_v1.sol";
import {ReservePool_v1} from "src/minter/ReservePool_v1.sol";
import {IGenesis} from "src/interfaces/IGenesis.sol";
import {IMinter} from "src/interfaces/IMinter.sol";
import {IStabilityPool} from "src/interfaces/IStabilityPool.sol";
import {IStabilityPoolManager} from "src/interfaces/IStabilityPoolManager.sol";
import {IMultipleRewardAccumulator} from
↪  "src/interfaces/IMultipleRewardAccumulator.sol";

import {Genesis_v1} from "src/minter/Genesis_v1.sol";

import {TestStabilityPoolSetUp} from "test/StabilityPool.t.sol";
import {Token} from "@bao/Token.sol";

import {TestMinterSetUp} from "test/Minter_base.t.sol";
import {UnsafeUpgrades} from "openzeppelin-foundry-upgrades/Upgrades.sol";
import {MockERC20, MockERC20Burn2Arg, MockERC20Burn1Arg, MockERC20BurnFrom} from
↪  "test/mock/MockERC20.sol";
import {MockWrappedPriceOracle} from "test/mock/MockWrappedPriceOracle.sol";

contract BaoPoCFrontrunHarvest is TestStabilityPoolSetUp {
    address genesisImpl;
    address genesis;
    address treasury;

    address alice;
    address bob;
    address george;
```

```
function setUp() override public {
    vm.createSelectFork(vm.rpcUrl("mainnet"), 19210000);
    owner = makeAddr("owner");
    feeReceiver = makeAddr("feeReceiver");
    treasury = makeAddr("treasury");

    alice = makeAddr("alice");
    bob = makeAddr("bob");
    george = makeAddr("george");

    vm.startPrank(owner);
    priceOracle = address(new MockWrappedPriceOracle());
    setUp_leveragedToken();
    peggedToken = address(new MockERC20("BaoUSD", "BAOUSD", 18));
    vm.label(peggedToken, "PeggedToken");
    vm.label(leveragedToken, "LeveragedToken");
    peggedTokenBurnSig = "burnFrom(address,uint256)";
    wrappedCollateralToken = Deployed.wstETH;
    collateralToken = Deployed.stETH;
}

/// @notice Alice & Bob bootstrap liquidity and claim their tokens
function genesisDepositsAndClaim() public {
    // Setup Genesis
    genesisImpl = address(new Genesis_v1(minter));
    genesis = UnsafeUpgrades.deployUUPSProxy(
        genesisImpl, //"Genesis_v1.sol",
        abi.encodeCall(Genesis_v1.initialize, owner)
    );
    IBaoOwnable(genesis).transferOwnership(owner);

    deal(address(Deployed.wstETH), alice, 1 ether);
    deal(address(Deployed.wstETH), bob, 3 ether);

    // setup access
    vm.startPrank(owner);
    IBaoRoles(minter).grantRoles(genesis, zeroFeeRole);

    // Deposit & End Genesis

    vm.startPrank(alice);
    IERC20(Deployed.wstETH).approve(genesis, 1 ether);
    Genesis_v1(genesis).deposit(1 ether, alice);

    vm.startPrank(bob);
    IERC20(Deployed.wstETH).approve(genesis, 3 ether);
    Genesis_v1(genesis).deposit(3 ether, bob);

    vm.startPrank(owner);
```

```
        Genesis_v1(genesis).endGenesis();

        console.log("\n--- Genesis State after endGenesis() ---");
        console.log("Minter wstETH balance:   ",
        ↪  IERC20(Deployed.wstETH).balanceOf(minter));
        console.log("Minter CR:               ", IMinter(minter).collateralRatio());
        console.log("Genesis pegged tokens:   ",
        ↪  IERC20(peggedToken).balanceOf(genesis));
        console.log("Genesis leveraged tokens:",
        ↪  IERC20(leveragedToken).balanceOf(genesis));

        // Users claim
        vm.startPrank(alice);
        Genesis_v1(genesis).claim(alice);

        vm.startPrank(bob);
        Genesis_v1(genesis).claim(bob);

        console.log("\n--- Users Claim tokens for shares ---");
        console.log("Alice pegged:   ", IERC20(peggedToken).balanceOf(alice));
        console.log("Alice leveraged:", IERC20(leveragedToken).balanceOf(alice));
        console.log("Alice shares:   ", Genesis_v1(genesis).balanceOf(alice));

        console.log("Bob pegged:   ", IERC20(peggedToken).balanceOf(bob));
        console.log("Bob leveraged:", IERC20(leveragedToken).balanceOf(bob));
        console.log("Bob shares:   ", Genesis_v1(genesis).balanceOf(bob));
        console.log();
        console.log("Minter CR:    ", IMinter(minter).collateralRatio());
        console.log();
    }

    /// @notice default minter setup
    function setupMinter() public {
        minter = UnsafeUpgrades.deployUUPSProxy(
            address(new Minter_v1(wrappedCollateralToken, peggedToken,
            ↪  leveragedToken, peggedTokenBurnSig)), // "Minter_v1.sol",
            abi.encodeCall(Minter_v1.initialize, (owner))
        );
        zeroFeeRole = IMinter(minter).ZERO_FEE_ROLE();

        IMinter(minter).updatePriceOracle(priceOracle);
        IMinter(minter).updateFeeReceiver(feeReceiver);
        IMinter(minter).updateReservePool(reservePool);
        IMinter(minter).updateConfig(config);

        IBaoRoles(leveragedToken).grantRoles(minter, minterRole);
        IBaoRoles(leveragedToken).grantRoles(minter, burnerRole);
    }

    /// FOUNDRY_PROFILE=novyper forge test --mt testFrontrunHarvestRewards -vv
```

```
    function testFrontrunHarvestRewards() public {
        setUp_config_likely();

        setupMinter();

        MockWrappedPriceOracle(priceOracle).setLatestAnswer(
            2000000000 * 1e10,      // min price 20.00
            2000000000 * 1e10,      // max price 20.00
            1200000000000000000,    // min rate 1.20
            1200000000000000000     // max rate 1.20
        );

        genesisDepositsAndClaim();

        vm.stopPrank();

        // Setup SPs and SPManager

        stabilityPoolCollateral = _setupStabilityPool(wrappedCollateralToken);
        address stabilityPoolLeveraged = _setupStabilityPool(leveragedToken);

        address stabilityPoolManager = UnsafeUpgrades.deployUUPSProxy(
            address(new StabilityPoolManager_v1(minter, treasury,
            ↪  stabilityPoolCollateral, stabilityPoolLeveraged)),
            abi.encodeCall(StabilityPoolManager_v1.initialize, owner)
        );
        IBaoOwnable(stabilityPoolManager).transferOwnership(owner);

        uint256 rebalancerRole =
        ↪  IStabilityPool(stabilityPoolCollateral).REBALANCER_ROLE();
        uint256 harvesterRole = IMinter(minter).HARVESTER_ROLE();

        // Grant roles
        vm.startPrank(owner);
        IBaoRoles(stabilityPoolCollateral).grantRoles(stabilityPoolManager,
        ↪  rebalancerRole);
        IBaoRoles(stabilityPoolLeveraged).grantRoles(stabilityPoolManager,
        ↪  rebalancerRole);
        IBaoRoles(minter).grantRoles(stabilityPoolManager, zeroFeeRole);
        IBaoRoles(minter).grantRoles(stabilityPoolManager, harvesterRole);
        vm.stopPrank();


        console.log("\n--- Alice deposits in SP pegged tokens ---");
        vm.startPrank(alice);
        uint256 peggedTokenDeposit = 10 ether;
        IERC20(peggedToken).approve(stabilityPoolCollateral, peggedTokenDeposit);
        IStabilityPool(stabilityPoolCollateral).deposit(peggedTokenDeposit, alice,
        ↪  0);
```

```
        console.log("Alice's contribution to SP:",
        ↪  IStabilityPool(stabilityPoolCollateral).assetBalanceOf(alice));
        console.log("Alice's claimable rewards: ",
        ↪  IMultipleRewardAccumulator(stabilityPoolCollateral).claimable(alice,
        ↪  wrappedCollateralToken));

        console.log("\n--- wstETH rate increases ---");

        // Increasing rate increases harvestable
        uint256 newRate = 1220000000000000000; // from 1.20 to 1.22
        MockWrappedPriceOracle(priceOracle).setLatestAnswer(
            2000000000 * 1e10,      // min price 20.00
            2000000000 * 1e10,      // max price 20.00
            newRate,
            newRate
        );

        uint256 harvestable = IMinter(minter).harvestable();
        console.log("harvestable:", harvestable);
        console.log("SP collateral wstETH balance:",
        ↪  IERC20(wrappedCollateralToken).balanceOf(stabilityPoolCollateral));
        console.log("SP leveraged wstETH balance: ",
        ↪  IERC20(wrappedCollateralToken).balanceOf(stabilityPoolLeveraged));
        console.log("Alice's wstETH balance:      ",
        ↪  IERC20(wrappedCollateralToken).balanceOf(alice));
        console.log("Bob's wstETH balance  :      ",
        ↪  IERC20(wrappedCollateralToken).balanceOf(bob));

        uint256 beforeHarvest = vm.snapshotState();

        console.log("\n--- A third party - George, calls the harvest ---");
        // Since harvest bounty ratio has not been setup, George will receive 0
        vm.startPrank(george);
        StabilityPoolManager_v1(stabilityPoolManager).harvest(george, 0);

        console.log("SP collateral wstETH balance:",
        ↪  IERC20(wrappedCollateralToken).balanceOf(stabilityPoolCollateral));
        console.log("SP leveraged wstETH balance: ",
        ↪  IERC20(wrappedCollateralToken).balanceOf(stabilityPoolLeveraged));
        console.log("Alice's claimable rewards: ",
        ↪  IMultipleRewardAccumulator(stabilityPoolCollateral).claimable(alice,
        ↪  wrappedCollateralToken));
        console.log("Alice gets 100% of the rewards, since she's the sole depositor
        ↪  in stabilityPoolCollateral");
        vm.startPrank(alice);
        IMultipleRewardAccumulator(stabilityPoolCollateral).claim();
        console.log("Alice's wstETH balance after harvest:",
        ↪  IERC20(wrappedCollateralToken).balanceOf(alice));

        vm.revertToState(beforeHarvest);
```

```
        console.log("\n--- Revert to state before harvest and have Bob front-run
        ↪  ---");
        console.log("Bob's wstETH balance before harvest:",
        ↪  IERC20(wrappedCollateralToken).balanceOf(bob));

        // Bob front-runs harvest by depositing all his pegged tokens
        vm.startPrank(bob);
        peggedTokenDeposit = 36 ether;
        IERC20(peggedToken).approve(stabilityPoolCollateral, peggedTokenDeposit);
        IStabilityPool(stabilityPoolCollateral).deposit(peggedTokenDeposit, bob, 0);

        vm.startPrank(george);
        StabilityPoolManager_v1(stabilityPoolManager).harvest(george, 0);

        console.log("SP collateral wstETH balance:",
        ↪  IERC20(wrappedCollateralToken).balanceOf(stabilityPoolCollateral));
        console.log("SP leveraged wstETH balance: ",
        ↪  IERC20(wrappedCollateralToken).balanceOf(stabilityPoolLeveraged));
        console.log("Alice's claimable rewards: ",
        ↪  IMultipleRewardAccumulator(stabilityPoolCollateral).claimable(alice,
        ↪  wrappedCollateralToken));
        vm.startPrank(alice);
        IMultipleRewardAccumulator(stabilityPoolCollateral).claim();
        console.log("Alice's wstETH balance after harvest:",
        ↪  IERC20(wrappedCollateralToken).balanceOf(alice));

        vm.startPrank(bob);
        IMultipleRewardAccumulator(stabilityPoolCollateral).claim();
        console.log("Bob's wstETH balance after harvest:",
        ↪  IERC20(wrappedCollateralToken).balanceOf(bob));

        // Bob gets back his pegged tokens
        IStabilityPool(stabilityPoolCollateral).withdraw(peggedTokenDeposit, bob,
        ↪  peggedTokenDeposit);
        console.log("Bob's pegged token balance after harvest:",
        ↪  IERC20(peggedToken).balanceOf(bob));
    }
}
```

Tool Used

Manual Review

Recommendation

Use the built-in linear distribution mechanism that will prevent attacker from depositing pegged tokens for a tx and then withdraw them after receiving the full reward.

Issue M-1: Lido slashing event can result in temporal undercollateralization of the protocol

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/39

Summary

If there is a slashing event in Lido this would reduce the `wstETH : stETH` rate meaning that the saved underlying collateral will be worth more wstETH than the actual deposit value.

Vulnerability Detail

Users deposit wstETH which is then accounted for in stETH terms based on the current `wstETH : stETH` rate. Usually, this rate should be rising due to the staking yield, but it can also lowers due to a slashing event which is an unusual scenario.

If the total deposited wstETH is 10 at rate 1.2, the accounted underlying collateral will be worth 12 stETH. If there is even a small reduce the rate due to slashing, there wont be enough wstETH in the contract. For example, at rate 1, the 12 stETH will be worth 12 wstETH while deposited is only 10 wstETH.

This gives users a hedge against slashing events as they would still get the same value for their redeemed pegged tokens. But it is also results in impossibility for all users to withdraw their collateral because there wont be enough `wstETH` in the protocol.

Impact

Inability for all users to withdraw.

Tool Used

Manual Review

Recommendation

As discussed with the team, this risk is accepted as there have to be a huge slashing event in order to eliminate the accrued yield from the staked `ETH` (yield results in increased rate, while slashing results in decreasing that rate).

Issue M-2: Last pegged token redeemer may not be able to redeem

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/40

Summary

When redeeming pegged tokens, users may receive additional collateral from the reserve pool based on the current collateral ratio. However, the last redeemer may be unable to redeem due to flawed accounting logic. This logic first subtracts the collateral returned to the user (underlying collateral plus the discount from the reserve pool) and only then adds the discount to the total underlying collateral. This fails to properly account for the fact that not all collateral given to the user comes from the underlying collateral.

Vulnerability Detail

At a high level, the logic for accounting for collateral when redeeming pegged tokens is as follows:

```
underlyingCollateral_ -= (underlyingCollateralOut$ / 1 ether);
if (underlyingFee$ > 0) {
    underlyingCollateral_ -= (underlyingFee$ / 1 ether);
}
if (underlyingDiscount$ > 0) {
    underlyingCollateral_ += underlyingDiscount$ / 1 ether;
}
```

Here, `underlyingCollateralOut$` includes both the underlying collateral for the pegged tokens and the discount. For the last remaining pegged token holder, their pegged tokens may be worth the entire `underlyingCollateral_`. Consequently, subtracting `underlyingCollateralOut$`, which equals `underlyingCollateral + discount`, from `underlyingCollateral_` would result in an underflow, preventing the redemption.

Impact

Last pegged token holder may not be able to redeem his tokens.

Code Snippet

Underlying collateral accounting in redeemPeggedToken()

Tool Used

Manual Review

Recommendation

Consider, first accounting for the fee and discount and then subtract the `underlyingCollateralOut$`.

Issue M-3: Handling zero deposits when accumulating rewards is insufficient

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/41

Summary

The `MultipleRewardCompoundingAccumulator::_accumulateReward()` function has logic to queue rewards when there are not any stakers in the SP. But checking the `totalShare` value does not accurately validates if there are stakers or not due to the rounding in favor of the protocol which always leave some shares even if all users withdraw.

Vulnerability Detail

```
if (totalShare == 0) {
    // no deposits, queue rewards
    _getRewardData(token).queued += uint96(amount);
    return;
}
```

Even if all stakers withdraw, the `totalShare` will never reach 0 as on each withdrawal, the rounding in the calculations is in favor of the protocol. The only case, where this value is going to be 0 is when there was a full liquidation.

In case we have gauge rewards, during a time with 0 depositors, rewards will be stuck instead of queued.

Impact

Stuck rewards.

Code Snippet

No deposits handling in accumulateReward()

Tool Used

Manual Review

Recommendation

Implementation of a non-zero deposit SP due to the issue with liquidation rewards fixes this problem.

Issue M-4: Overflow risk when updating the integral value for rewards accounting in Stability Pool

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/42

Summary

There is a risk of overflow or moving the value of the SP's integral close to the max supported value when rebalance is executed with CR close to 100%, but over it.

Vulnerability Detail

The `MultipleRewardCompoundingAccumulator::_accumulateReward()` function updates the `integral` value when reward are added in the SP due to liquidation of pegged tokens:

```
integral += (uint192((amount * _REWARD_PRECISION) / totalShare) *
↪  uint192(magnitude));
```

For the leveraged stability pool, the number of leveraged tokens to mint and send to the pool with equal value to the pegged token liquidation is calculated as follows:

```
    function _leveragedTokensForPegged(
        uint256 peggedIn,
        uint256 leveragedTokenBalance_,
        uint256 peggedTokenBalance_,
        uint256 collateralTokenBalance_,
        uint256 collateralPrice
    ) private pure returns (uint256 leveragedTokens) {
        // this is the first derivative of the legeraged balance with respect to
        ↪  the collateral balance
        // in the invariant: collateral value = leveraged value + pegged value.
        if (leveragedTokenBalance_ > 0) {
            uint256 collateralValue = collateralTokenBalance_ * collateralPrice;
            uint256 peggedValue = peggedTokenBalance_ * 1 ether;

            if (peggedValue >= collateralValue) {
                leveragedTokens = 0;
            } else {
@>              leveragedTokens = (peggedIn * 1 ether * leveragedTokenBalance_) /
↪  (collateralValue - peggedValue);
            }
        } else {
            leveragedTokens = peggedIn; // TODO: the third place initial price of 1
            ↪  ether is assumed
        }
```

```
    }
```

If we are in close to 100% CR, the difference between `collateralValue` and `peggedValue` can be a small number (value of leveraged token is close to 0). Thus the number of minted leveraged tokens can be in the magnitude of `1e54`. This is the `amount` value in the integral formula.

Max supported value for `uint192` is `~ 6e57`, thus depending on denominator in the `leveragedTokens` formula the `integral` calculation can either revert or drive the value close to the max supported value which will prevent rebalancing in the future.

Impact

Overflow resulting in inability to rebalance the CR.

Code Snippet

MultipleRewardCompoundingAccumulator::_accumulateReward()

Minter::_leveragedTokensForPegged()

Tool Used

Manual Review

Recommendation

One option is to reduce the size of the timestamp and increase the size of the integral in the `RewardSnapshot` struct:

```
    struct RewardSnapshot {
        // The timestamp when the snapshot is updated.
-       uint64 timestamp;
+       uint40 timestamp;
        // The reward integral until now.
-       uint192 integral;
+       uint216 integral;
    }
```

`uint40` can represent timestamp up to 2106 year, but still the max value of the integral might not be enough as it can hold up to `~1e65` and it can be reached for larger `peggedIn` and `leveragedTokenBalance` values.

Another options is to reduce the precision in the `collateralValue` and `peggedValue` related calculations to `1e18`.

Issue M-5: Rebalancing does not work in a depeg scenario

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/43

Summary

The rebalance operation aims to increase the collateral ratio by redeeming pegged tokens and swapping them for leveraged tokens based on the SP type. Under normal conditions with a collateral ratio above 100%, it effectively increases the ratio. However, in a depeg scenario, the underlying mechanisms do not function correctly.

Vulnerability Detail

The `Minter::_redeemPeggedForCollateralRatio()`, which gives us how much pegged tokens should be redeemed to reach a certain ratio, does not work for CR < 100%, as it would return values higher than the total pegged minted.

Apart from the collateral stability pool mechanism, swapping pegged for leveraged in a depeg case, results in 0 minted leveraged tokens, due to a depeg check in `Minter::_leveragedTokensForPegged()`, which means that stability pool stakers' deposits would be burned in exchange for nothing.

The following test shows the behavior:

```
function test_rebalanceDepeg() public {
        uint256 threshold = 1.3 ether;
        uint256 bountyRatio = 0.2 ether;

        vm.startPrank(owner);
        IStabilityPoolManager(stabilityPoolManager).setRebalanceThreshold(threshold
        ↪ );
        IStabilityPoolManager(stabilityPoolManager).setRebalanceBountyRatio(bountyR
        ↪ atio);
        vm.stopPrank();
        // Check rebalanceable
        assertTrue(IStabilityPoolManager(stabilityPoolManager).rebalanceable(),
        ↪ "Should be rebalanceable");

        (uint256 price, , , ) = IWrappedPriceOracle(priceOracle).latestAnswer();

        // Setup conditions for successful rebalance using the manager's ratio
        setUp_collateral(100 ether, 20 ether, user); // CR = 120 / 100 = 120%

        // Fund the stability pools
        vm.startPrank(user);
```

```
        IERC20(peggedToken).approve(stabilityPoolCollateral, type(uint256).max);
        IERC20(peggedToken).approve(stabilityPoolLeveraged, type(uint256).max);

        uint256 userPegged = IERC20(peggedToken).balanceOf(user);
        assertEq(userPegged, 100 * price, "User should have 100 pegged tokens");

        IStabilityPool(stabilityPoolCollateral).deposit(userPegged / 3, user, 0);
        IStabilityPool(stabilityPoolLeveraged).deposit(userPegged - (userPegged /
        ↪  2), user, 0);
        vm.stopPrank();

        // Execute rebalance as liquidator
        Balances memory before = _readBalances();
        assertEq(IERC20(leveragedToken).balanceOf(stabilityPoolCollateral), 0,
        ↪  "pool1 has no leveraged");

        MockWrappedPriceOracle(priceOracle).setLatestAnswer(1555 ether); // makes
        ↪  the collateral ratio = 0.93
        (price, , , ) = IWrappedPriceOracle(priceOracle).latestAnswer();

        console.log("price: %s", price);
        console.log("CR: %s", IMinter(minter).collateralRatio());
        console.log("Pegged balance before rebalance: %s",
        ↪  IMinter(minter).peggedTokenBalance());

        IStabilityPoolManager(stabilityPoolManager).rebalance(bountyReceiver, 0);

        console.log("Pegged balance AFTER: %s",
        ↪  IMinter(minter).peggedTokenBalance());
        console.log("CR AFTER: %s", IMinter(minter).collateralRatio());

        // we hit the rebalance collateral ratio exactly
        assertEq(IMinter(minter).collateralRatio(), threshold, "collateral ratio is
        ↪  reset after rebalance");
}

[FAIL: collateral ratio is reset after rebalance: 1250799216252449211 !=
↪  1300000000000000000]
Logs:
  price: 1555000000000000000000
  CR: 933000000000000000
  Pegged balance before rebalance: 200000000000000000000000
  Pegged balance AFTER:            994564102564102564102 56
  CR AFTER: 1250799216252449211
```

Impact

Rebalance operation does not work as expected in a depeg scenario + leveraged stability pool stakers' deposit is burned in exchange for nothing.

Tool Used

Manual Review

Recommendation

Based on an internal discussion, fixing include capping the number of pegged tokens that can be redeemed in CR < 1, capping the leverage ratio and swapping pegged for collateral and for leveraged tokens at the same time, instead of one after the other, both in terms of querying amounts and when doing an actual swap. This prevents the collateral ratio (CR) changing between swapping one followed by the other.

Issue M-6: Leveraged token holder may not be able to redeem

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/44

Summary

If the protocol reaches a state where no pegged tokens are minted, leveraged token holders will not be able to redeem their tokens. In this scenario, leveraged tokens absorb all price movements of the tracked oracle price, and users cannot redeem their tokens. Additionally, minting new pegged tokens is not possible, as all mint and redeem actions are frozen.

Vulnerability Detail

Each mint or redeem action includes a check for the pegged token balance, and if this balance is zero, the transaction reverts. This design effectively locks all leveraged token collateral. The only way to exit this state is for an admin to use the freeMintPeggedToken() function to mint new pegged tokens and unfreeze the protocol.

Impact

Temporally locked funds.

Tool Used

Manual Review

Recommendation

Consider allowing the redemption of leveraged tokens in such cases, distributing collateral to the redeemer proportional to their share of the total leveraged tokens.

Issue M-7: No additional incentive for stability pool stakers to keep their deposits during a rebalance

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/45

Summary

Stability pool stakers have no direct incentive to keep their pegged tokens staked during a rebalance operation. Users can sandwich a rebalance call, avoiding a reduction in their staked balance and thereby receiving better future rewards compared to stakers whose balances were reduced due to the liquidation.

Vulnerability Detail

For example, Liquity V2 Stability Pool stakers have an additional incentive to maintain their deposits during a liquidation, as the value of the tokens they receive for their stake is slightly higher than the value taken from them.In the current implementation, the value of the rewards equals the value of the pegged tokens taken from users. More sophisticated users can withdraw just before the rebalance and redeposit afterward. As a result, they maintain the absolute value of their original stake, which now represents a higher percentage of the total deposits, allowing them to claim a larger share of the accumulated rewards.

Providing an additional incentive by increasing the reward value for the liquidated amount can encourage users to maintain their stake during liquidation. It also incentivizes just-in-time liquidity providers, which is beneficial for the protocol as it helps maintain a higher Stability Pool balance.

Impact

Users gaming the rebalance mechanism, essentially stealing rewards from other depositors.

Tool Used

Manual Review

Recommendation

Consider adding some additional instant incentive for the stakers during a rebalance operation.

Issue M-8: Minting pegged tokens will begin to revert with higher TVL

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/46

Summary

Minting any amount of pegged tokens requires calculating how much collateral exists in a fee band. Due to how values are scaled, it is possible that `collateralInBand` overflows.

Vulnerability Detail

The problematic value is this intermediate expression: `((bandLowerBound * peggedTokenBalance_) * 1 ether * 1 ether)` Since the first two multipliers are 1e18 scale and they are then scaled to 1e36, with big enough `peggedTokenBalance_` the expression will cause an overflow.

Impact

Minting pegged tokens will revert

Code Snippet

Example:

```
peggedTokenBalance_ = 10M = 1e25
bandLowerBound = 110 = scaled -> 1.1e18
1 ether * 1 ether = 1e36

numerator = 1e79
uint256 overflows at 1e78
```

Tool Used

Manual Review

Recommendation

Use `Math.mulDiv(a, b, c)` to handle the overflow.

Issue M-9: Underlying collateral is miscalculated when minting pegged tokens

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/48

Summary

In the function `mintPeggedToken` users supply wrapped collateral and get pegged token in return. The minter contract keeps track of the unwrapped collateral in the `underlyingCollateral` variable. The issue is that when fees are subtracted they need to be in the unwrapped form.

Vulnerability Detail

In the following calculation:

```
$.underlyingCollateral = underlyingCollateral_ + underlyingCollateralIn -
↪  wrappedFee;
```

It should be `underlyingFee` that should be subtracted, instead of `wrappedFee`.

Impact

The `underlyingCollateral` will become bigger than it should be, which will lead to overestimating the collateral ratio in the system.

Tool Used

Manual Review

Recommendation

Subtract `underlyingFee` instead.

Issue L-1: Wrapped collateral for collateral ratio formula produces unintended result

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/47

Summary

The formula for calculating how much wrapped collateral is necessary to reach a target collateral ratio (`wrappedCollateralForCollateralRatio`) has dimensions mixed up.

Vulnerability Detail

This formula: `($.peggedTokenBalance * 1 ether * 1 ether) / (targetCollateralRatio * oracle.price) - currentCollateralRatio` Does not produce how much additional collateral has to be added to reach the target CR.

Impact

The wrapped collateral amount returned by the function is wrong and in certain cases it will revert.

Code Snippet

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.28 <0.9.0;

import {console2 as console} from "forge-std/console2.sol";
import {Vm} from "forge-std/Vm.sol";

import {Deployed} from "@bao/Deployed.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IBaoOwnable} from "@bao/interfaces/IBaoOwnable.sol";
import {IBaoRoles} from "@bao/interfaces/IBaoRoles.sol";
import {IBurnable2Arg} from "@bao/interfaces/IBurnable2Arg.sol";
import {Minter_v1} from "src/minter/Minter_v1.sol";
import {MintableBurnableERC20_v1} from "@bao/MintableBurnableERC20_v1.sol";
import {ReservePool_v1} from "src/minter/ReservePool_v1.sol";
import {IGenesis} from "src/interfaces/IGenesis.sol";
import {IMinter} from "src/interfaces/IMinter.sol";

import {Genesis_v1} from "src/minter/Genesis_v1.sol";

import {TestMinterSetUp} from "test/Minter_base.t.sol";
import {Token} from "@bao/Token.sol";
```

```
import {TestMinterSetUp} from "test/Minter_base.t.sol";
import {UnsafeUpgrades} from "openzeppelin-foundry-upgrades/Upgrades.sol";
import {MockERC20, MockERC20Burn2Arg, MockERC20Burn1Arg, MockERC20BurnFrom} from
↪  "test/mock/MockERC20.sol";
import {MockWrappedPriceOracle} from "test/mock/MockWrappedPriceOracle.sol";

contract BaoPoCWrappedCollateralForTCR is TestMinterSetUp {
    address genesisImpl;
    address genesis;

    address alice;

    function setUp() override public {
        vm.createSelectFork(vm.rpcUrl("mainnet"), 19210000);
        owner = makeAddr("owner");
        feeReceiver = makeAddr("feeReceiver");

        alice = makeAddr("alice");

        vm.startPrank(owner);
        priceOracle = address(new MockWrappedPriceOracle());
        setUp_leveragedToken();
        setUp_reservePool();
        peggedToken = address(new MockERC20("BaoUSD", "BAOUSD", 18));
        vm.label(peggedToken, "PeggedToken");
        vm.label(leveragedToken, "LeveragedToken");
        peggedTokenBurnSig = "burnFrom(address,uint256)";
        wrappedCollateralToken = Deployed.wstETH;
        collateralToken = Deployed.stETH;
    }

    /// FOUNDRY_PROFILE=novyper forge test --mt
    ↪  testWrappedCollateralForTargetCollateral -vv
    function testWrappedCollateralForTargetCollateral() public {
        minter = UnsafeUpgrades.deployUUPSProxy(
            address(new Minter_v1(wrappedCollateralToken, peggedToken,
            ↪  leveragedToken, peggedTokenBurnSig)),
            abi.encodeCall(Minter_v1.initialize, (owner))
        );
        zeroFeeRole = IMinter(minter).ZERO_FEE_ROLE();

        setUp_config_likely();

        IMinter(minter).updatePriceOracle(priceOracle);
        IMinter(minter).updateFeeReceiver(feeReceiver);
        IMinter(minter).updateReservePool(reservePool);
        IMinter(minter).updateConfig(config);

        IBaoRoles(leveragedToken).grantRoles(minter, minterRole);
```

```
        IBaoRoles(leveragedToken).grantRoles(minter, burnerRole);

        // Setup Genesis
        genesisImpl = address(new Genesis_v1(minter));
        genesis = UnsafeUpgrades.deployUUPSProxy(
            genesisImpl, //"Genesis_v1.sol",
            abi.encodeCall(Genesis_v1.initialize, owner)
        );
        IBaoOwnable(genesis).transferOwnership(owner);

        deal(address(Deployed.wstETH), alice, 1 ether);
        MockWrappedPriceOracle(priceOracle).setLatestAnswer(
            2000000000 * 1e10,      // min price 20.00
            2000000000 * 1e10,      // max price 20.00
            1200000000000000000,    // min rate 1.20
            1200000000000000000     // max rate 1.20
        );

        // setup access
        vm.startPrank(owner);
        IBaoRoles(minter).grantRoles(genesis, zeroFeeRole);
        IBaoRoles(reservePool).grantRoles(minter,
        ↪  ReservePool_v1(reservePool).REQUESTER_ROLE());

        // Deposit & End Genesis

        vm.startPrank(alice);
        IERC20(Deployed.wstETH).approve(genesis, 1 ether);
        Genesis_v1(genesis).deposit(1 ether, alice);

        vm.startPrank(owner);
        Genesis_v1(genesis).endGenesis();

        console.log("\n--- End Genesis State ---");
        console.log("Minter wstETH balance:   ",
        ↪  IERC20(Deployed.wstETH).balanceOf(minter));
        console.log("Minter CR:               ", IMinter(minter).collateralRatio());
        console.log("Minter underlyingColl:   ",
        ↪  IMinter(minter).collateralTokenBalance());
        console.log("Genesis pegged tokens:   ",
        ↪  IERC20(peggedToken).balanceOf(genesis));
        console.log("Genesis leveraged tokens:",
        ↪  IERC20(leveragedToken).balanceOf(genesis));

        // Users claim
        vm.startPrank(alice);
        Genesis_v1(genesis).claim(alice);

        console.log("\n--- Alice claims tokens for shares ---");
        console.log("Alice pegged:   ", IERC20(peggedToken).balanceOf(alice));
```

```
        console.log("Alice leveraged:", IERC20(leveragedToken).balanceOf(alice));
        console.log("Alice wstETH:   ", IERC20(Deployed.wstETH).balanceOf(alice));
        console.log("Alice shares:   ", Genesis_v1(genesis).balanceOf(alice));

        console.log("\n--- Alice redeems leverage tokens ---");

        // Case 1: When alice redeems <= 8 leveraged, keeping CR >= 1.33,
        ↪  wrappedCollateralForCollateralRatio returns 0 as expected
        //         In this situation wrappedCollateral is always 0, because tCR
        ↪  1.33 < current CR
        //
        // Case 2: When alice redeems > 8 leveraged, making CR < 1.33,
        ↪  wrappedCollateralForCollateralRatio reverts with an underflow
        //         This is the practical use-case current CR < tCR, how much
        ↪  wrapped collateral do we need to deposit? We get a revert.

        uint256 amount = 9 ether;
        IERC20(leveragedToken).approve(minter, amount);
        IMinter(minter).redeemLeveragedToken(amount, alice, 0);
        console.log("Alice pegged:   ", IERC20(peggedToken).balanceOf(alice));
        console.log("Alice leveraged:", IERC20(leveragedToken).balanceOf(alice));
        console.log("Alice wstETH:   ", IERC20(Deployed.wstETH).balanceOf(alice));
        console.log("Minter CR:      ", IMinter(minter).collateralRatio());

        uint256 wrappedCollateral = IMinter(minter).wrappedCollateralForCollateralR
        ↪  atio(1333333333333333333); // tCR = 1.33
        console.log("wrappedCollateral: ", wrappedCollateral);
    }

}
```

Tool Used

Manual Review

Recommendation

The formula should be:

```
uint256 newCollateralBalance = (targetCollateralRatio * peggedTokenBalance_) /
↪  oracle.price;
uint256 additionalCollateral = newCollateralBalance - collateralTokenBalance_;
wrappedCollateral = _wrappedValueOf(additionalCollateral, oracle.rate);
```

Issue L-2: VotingEscrow_v1 contains unnecessary field

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/49

Summary

The `VotingEscrow_v1` contract is a solidity version of `VotingEscrow.vy`, which was initially created by the Curve protocol. The `transfersEnabled` field comes from a compatibility requirement for the Aragon DAO. To our knowledge this token will not be integrated with their UI so it can safely get dropped.

Impact

Contract contains unnecessary legacy integrations.

Tool Used

Manual Review

Recommendation

Remove the `bool transfersEnabled;` field altogether.

Issue L-3: VotingEscrow_v1 contains anti-tokenization mechanism that no longer works after Pectra

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/50

Summary

When initially the Curve protocol developed this contract they made a check to make sure only EOAs would be able to lock tokens and receive voting power. This was done with the intention of preventing tokenizing and commoditizing the voting power.

However, after the Pectra upgrade the implemented check can be worked around.

Vulnerability Detail

The following modifier checks for EOA:

```
modifier onlyAllowedContractOrEOA() {
    // Check if the caller is an EOA or has the required role
    if (
        // solhint-disable-next-line avoid-tx-origin
        msg.sender != tx.origin && // wake-disable-line tx-origin
        !hasAnyRole(msg.sender, SMART_CONTRACT_MANAGER_ROLE)
    ) {
        revert Unauthorized();
    }
    _;
}
```

The issue is that after the Pectra upgrade, EOAs can delegate contracts to themselves and allow for others to call said contracts. Thus, the following is possible:

1. EOA locks tokens for voting power
2. EOA delegates to a contract via EIP-7702
3. Anyone can interact with the contract from step 2 and buy EOA's voting power

Impact

Check preventing contracts to buy voting power no longer works.

Tool Used

Manual Review

Recommendation

The team is made aware of this problem and they are not against the commoditization of voting power.

The `onlyAllowedContractOrEOA` modifier and the related `SMART_CONTRACT_MANAGER_ROLE` should be removed altogether.

Issue L-4: redeemPeggedForCollateralRatio misbehaves when CR < 1

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/52

Summary

When `Minter` collateral ratio is below 1, no amount of pegged tokens redemption will improve the collateral ratio.

Vulnerability Detail

If we call `redeemPeggedForCollateralRatio` when CR is below 1, it will return an amount of pegged tokens, which shouldn't be the case.

Impact

Returning an amount different from 0 could be interpreted that pegged tokens redemption will improve CR.

Code Snippet

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.28 <0.9.0;

import {console2 as console} from "forge-std/console2.sol";
import {Vm} from "forge-std/Vm.sol";

import {Deployed} from "@bao/Deployed.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IBaoOwnable} from "@bao/interfaces/IBaoOwnable.sol";
import {IBaoRoles} from "@bao/interfaces/IBaoRoles.sol";
import {IBurnable2Arg} from "@bao/interfaces/IBurnable2Arg.sol";
import {Minter_v1} from "src/minter/Minter_v1.sol";
import {MintableBurnableERC20_v1} from "@bao/MintableBurnableERC20_v1.sol";
import {ReservePool_v1} from "src/minter/ReservePool_v1.sol";
import {IGenesis} from "src/interfaces/IGenesis.sol";
import {IMinter} from "src/interfaces/IMinter.sol";

import {Genesis_v1} from "src/minter/Genesis_v1.sol";

import {TestMinterSetUp} from "test/Minter_base.t.sol";
import {Token} from "@bao/Token.sol";
```

```
import {TestMinterSetUp} from "test/Minter_base.t.sol";
import {UnsafeUpgrades} from "openzeppelin-foundry-upgrades/Upgrades.sol";
import {MockERC20, MockERC20Burn2Arg, MockERC20Burn1Arg, MockERC20BurnFrom} from
↪  "test/mock/MockERC20.sol";
import {MockWrappedPriceOracle} from "test/mock/MockWrappedPriceOracle.sol";

contract BaoPoCRedeemPeggedLowCR is TestMinterSetUp {
    address genesisImpl;
    address genesis;

    address alice;

    function setUp() override public {
        vm.createSelectFork(vm.rpcUrl("mainnet"), 19210000);
        owner = makeAddr("owner");
        feeReceiver = makeAddr("feeReceiver");

        alice = makeAddr("alice");

        vm.startPrank(owner);
        priceOracle = address(new MockWrappedPriceOracle());
        setUp_leveragedToken();
        setUp_reservePool();
        peggedToken = address(new MockERC20("BaoUSD", "BAOUSD", 18));
        vm.label(peggedToken, "PeggedToken");
        vm.label(leveragedToken, "LeveragedToken");
        peggedTokenBurnSig = "burnFrom(address,uint256)";
        wrappedCollateralToken = Deployed.wstETH;
        collateralToken = Deployed.stETH;
    }


    /// FOUNDRY_PROFILE=novyper forge test --mt testBasicPeggedRedeemCRBelow1 -vv
    function testBasicPeggedRedeemCRBelow1() public {
        minter = UnsafeUpgrades.deployUUPSProxy(
            address(new Minter_v1(wrappedCollateralToken, peggedToken,
            ↪  leveragedToken, peggedTokenBurnSig)),
            abi.encodeCall(Minter_v1.initialize, (owner))
        );
        zeroFeeRole = IMinter(minter).ZERO_FEE_ROLE();

        setUp_config_likely();

        IMinter(minter).updatePriceOracle(priceOracle);
        IMinter(minter).updateFeeReceiver(feeReceiver);
        IMinter(minter).updateReservePool(reservePool);
        IMinter(minter).updateConfig(config);

        IBaoRoles(leveragedToken).grantRoles(minter, minterRole);
        IBaoRoles(leveragedToken).grantRoles(minter, burnerRole);
```

```
// Setup Genesis
genesisImpl = address(new Genesis_v1(minter));
genesis = UnsafeUpgrades.deployUUPSProxy(
    genesisImpl, //"Genesis_v1.sol",
    abi.encodeCall(Genesis_v1.initialize, owner)
);
IBaoOwnable(genesis).transferOwnership(owner);

deal(address(Deployed.wstETH), alice, 1 ether);
MockWrappedPriceOracle(priceOracle).setLatestAnswer(
    2000000000 * 1e10,      // min price 20.00
    2000000000 * 1e10,      // max price 20.00
    1200000000000000000,    // min rate 1.20
    1200000000000000000     // max rate 1.20
);

// setup access
vm.startPrank(owner);
IBaoRoles(minter).grantRoles(genesis, zeroFeeRole);
IBaoRoles(reservePool).grantRoles(minter,
↪  ReservePool_v1(reservePool).REQUESTER_ROLE());

// Fund reserve pool
// deal(address(Deployed.wstETH), reservePool, 0.005 ether);


// Deposit & End Genesis

vm.startPrank(alice);
IERC20(Deployed.wstETH).approve(genesis, 1 ether);
Genesis_v1(genesis).deposit(1 ether, alice);

vm.startPrank(owner);
Genesis_v1(genesis).endGenesis();

console.log("\n--- End Genesis State ---");
console.log("Minter wstETH balance:   ",
↪  IERC20(Deployed.wstETH).balanceOf(minter));
console.log("Minter CR:               ", IMinter(minter).collateralRatio());
console.log("Minter underlyingColl:   ",
↪  IMinter(minter).collateralTokenBalance());
console.log("Genesis pegged tokens:   ",
↪  IERC20(peggedToken).balanceOf(genesis));
console.log("Genesis leveraged tokens:",
↪  IERC20(leveragedToken).balanceOf(genesis));

// Users claim
vm.startPrank(alice);
Genesis_v1(genesis).claim(alice);
```

```
        console.log("\n--- Alice claims tokens for shares ---");
        console.log("Alice pegged:   ", IERC20(peggedToken).balanceOf(alice));
        console.log("Alice leveraged:", IERC20(leveragedToken).balanceOf(alice));
        console.log("Alice wstETH:   ", IERC20(Deployed.wstETH).balanceOf(alice));
        console.log("Alice shares:   ", Genesis_v1(genesis).balanceOf(alice));

        console.log("\n--- Alice redeems leverage tokens ---");
        uint256 amount = 11 ether;
        IERC20(leveragedToken).approve(minter, amount);
        IMinter(minter).redeemLeveragedToken(amount, alice, 0);
        console.log("Alice pegged:   ", IERC20(peggedToken).balanceOf(alice));
        console.log("Alice leveraged:", IERC20(leveragedToken).balanceOf(alice));
        console.log("Alice wstETH:   ", IERC20(Deployed.wstETH).balanceOf(alice));
        console.log("Minter CR:      ", IMinter(minter).collateralRatio());


        console.log("\n--- Price falls ---");
        MockWrappedPriceOracle(priceOracle).setLatestAnswer(
            1000000000 * 1e10,      // min price 10.00
            1000000000 * 1e10,      // max price 10.00
            1200000000000000000,    // min rate 1.20
            1200000000000000000     // max rate 1.20
        );
        console.log("Minter CR:      ", IMinter(minter).collateralRatio());

        console.log("\n--- Alice dry run redeem pegged tokens ---");
        vm.startPrank(alice);
        IERC20(peggedToken).approve(minter, amount);

        console.log("uncerlyingCollateral:",
        ↪  IMinter(minter).collateralTokenBalance());
        (
            int256 incentiveRatio,
            uint256 fee,
            uint256 discount,
            uint256 peggedRedeemed,
            uint256 wrappedCollateralReturned,
            uint256 price,
            uint256 rate
        ) = IMinter(minter).redeemPeggedTokenDryRun(amount);

        console.log("incentiveRatio:", incentiveRatio);
        console.log("discount:", discount);
        console.log("peggedRedeemed:", peggedRedeemed);
        console.log("wrappedCollateralReturned:", wrappedCollateralReturned);
        console.log("price:", price);

        console.log("\n--- Alice redeems pegged tokens ---");
```

```
        uint256 collateralOut = IMinter(minter).redeemPeggedToken(amount, alice,
        ↪  0.1 ether);

        console.log("Alice aquired wstETH: ", collateralOut);
        console.log("Alice wstETH:         ",
        ↪  IERC20(Deployed.wstETH).balanceOf(alice));
        console.log("Alice holds pegged:   ", IERC20(peggedToken).balanceOf(alice));
        console.log("Minter CR:            ", IMinter(minter).collateralRatio());

        console.log("\n--- redeemPeggedForCollateralRatio ---");
        uint256 peggedTokensToRedeem =
        ↪  IMinter(minter).redeemPeggedForCollateralRatio(1333333333333333333); //
        ↪  tCR = 1.33
        console.log("peggedTokensToRedeem: ", peggedTokensToRedeem);
        console.log("pegged tokens supply: ", IERC20(peggedToken).totalSupply());

    }

}
```

Tool Used

Manual Review

Recommendation

When CR < 1, either return 0 or revert in `redeemPeggedForCollateralRatio`.

Issue L-5: mintPeggedTokenDryRun is inconsistent with how it handles wrappedCollateralIn

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/53

Summary

In `mintPeggedToken` the `wrappedCollateralIn` is adjusted like so:

```
wrappedCollateralIn = Token.allOf(_msgSender(), WRAPPED_COLLATERAL_TOKEN,
↪  wrappedCollateralIn);
```

However, in `mintPeggedTokenDryRun` this adjustment never happens.

Impact

- Mint pegged dry run callers can't use special value (`type(uint256).max`) to deposit their whole wstETH balance
- Inconsistency with how input value is transformed across dry run & real functions

Tool Used

Manual Review

Recommendation

Use `wrappedCollateralIn = Token.allOf(_msgSender(), WRAPPED_COLLATERAL_TOKEN, wrappedCollateralIn);` in dry run as well.

Issue L-6: TokenDistributor_v1 remove loop can return after pop

Source: https://github.com/sherlock-audit/2025-07-zhenglong/issues/54

Summary

When removing a split from the `distribution`, the loop can return after `.pop()` being called.

Impact

- Unnecessary loop iterations can be performed

Code Snippet

```
function removeRecipient(address recipient) public onlyOwner {
    TokenDistributorStorage storage $ = _getTokenDistributorStorage();
    // solhint-disable-next-line explicit-types
    for (uint i = 0; i < $.distribution.length; i++) {
        // solhint-disable-line explicit-types
        if (recipient == $.distribution[i].recipient) {
            // existing recipient removed
            $.totalShares = $.totalShares - $.distribution[i].share;
            if (i < $.distribution.length - 1) {
                // if it's in the middle,
                // copy the last one over it
                $.distribution[i] = $.distribution[$.distribution.length - 1];
            }
            // remove the last one
            $.distribution.pop();
            // @audit-issue -> we can return here
        }
    }
}
```

Tool Used

Manual Review

Recommendation

Add a return statement after `.pop()`.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.