CONCENTRATED LIQUIDITY TOOL AUDIT REPORT

TABLE OF CONTENTS

PROJECT DETAILS

Project Name: Concentrated Liquidity Tool
Blockchain: Ethereum and EVM based chains
Language: Solidity
Report Type: Final

Audit Findings Summary

#	Finding	Risk	Status
1	Insufficient input validation can lock user funds into strategy	Critical	Fixed
2	TWAP Manipulation and Liquidity Risk	Critical	Fixed
3	Incomplete Implementation in shiftLeft and shiftRight:	Medium	Fixed
4	Insufficient Error Messaging in require Statements for Authorization Checks	Low	Fixed
5	Implementing Dual-Signature Fail-Safe Mechanism for Emergency Withdrawals	Info	Acknowledged

FINDINGS

1. Insufficient input validation can lock user funds into strategy

STATUS: FIXED

DESCRIPTION

In the CLTBase contract, there is a potential vulnerability where the strategy owner can renounce ownership by setting the owner address to zero. This action could lead to a critical issue during fee transfer in the withdraw function. If the withdraw function attempts to transfer fees to the renounced strategy owner, the ERC20 token transfer will fail because tokens cannot be transferred to the zero address according to the ERC20 standard. As a result, if the fee transfer fails, user funds may become trapped in the strategy contract indefinitely without an emergency withdrawal mechanism in place. It's essential to consider this behavior when designing the system and ensure that appropriate checks and validations are in place to handle ownership changes.

IMPACT

- User funds may become stuck in the strategy contract if the withdrawal function fails to transfer fees to the renounced strategy owner.
- Lack of an emergency withdrawal mechanism means users may be unable to retrieve their funds in case of contract failure or unexpected circumstances.

POC

```
function test_renounceStrategy() public {
        //deposit
        loopedDepositsToStrategy();
        loopedDepositsToUniswap();
```

```
        //swap
        uint256 swapAmount = 1e30;
        for (uint256 i = 0; i < 10; i++) {
            swapsByContract(swapAmount);
        }
        console.log("10 users swapped in univ3 pool");

        //renounce
        bytes32 strategyID = getStrategyID(address(this), 1);
        //setting some fee
        ICLTBase.PositionActions memory actions = createStrategyActions(1, 3, 0,
3, 0, 0);
        //sending zero address as strategy owner
        base.updateStrategyBase(strategyID, address(0), 1, 1, actions);

        // update strategy fee
        base.getStrategyReserves(getStrategyID(address(this), 1));
        base.getStrategyReserves(getStrategyID(address(this), 2));
        base.getStrategyReserves(getStrategyID(address(this), 3));

        //withdraw
        try this.loopedWithdrawFromStrategy() {
            emit CallSucceeded();
        } catch Error(string memory reason) {
            emit CallFailed(reason);
        }
    }
```

REMEDY

Place a zero address check while changing the new strategy. Additionally, it is suggested to introduce a well-thought emergency withdrawal mechanism (as mentioned in 4 below) in case funds are stuck in any strategy.

2. **TWAP Manipulation and Liquidity Risk**

   STATUS: FIXED

   Description:
   A significant vulnerability in the ALP tool's handling of liquidity strategies, specifically within the ShiftLiquidity function of the CLTBase contract. The core of the issue lies in the absence of a checkDeviation safeguard, which opens up the potential for frontrunning and price manipulation attacks in volatile pools.

   An attacker, by manipulating the current tick through strategic trades right before executing key functions such as deposit, withdraw, or shiftLiquidity, can adversely affect the protocol's operations. For instance, in a volatile pool, a malicious actor can execute a large trade that drastically moves the current tick, making the protocol compute an incorrect amount for token conversion, leading to the deposit or withdrawal of funds at a manipulated and unfavorable rate. This kind of attack not only disrupts the intended liquidity provision strategy but also exposes users to potential financial losses due to Miner Extractable Value (MEV) attacks.

   Impact:
   The absence of the checkDeviation modifier in critical user-facing functions (deposit, withdraw) and admin/operator functions (shiftLiquidity) significantly heightens the risk of adverse user experiences and financial losses. Users could find their transactions executing at rates far removed from fair market values, resulting in direct financial losses or inefficient capital allocation. Furthermore, these vulnerabilities can be exploited by attackers to perform MEV attacks, extracting value from users' transactions through frontrunning or sandwich attacks, ultimately eroding trust in the platform and potentially leading to substantial liquidity withdrawal.

POC:
We tried to depict a front-running scenario in our local environment which is far different from a real world scenario where the functionality works but with unfavorable results for users' funds.

```
function testFrontrunning() public {
    vm.prank(specialUsers[1]);
    base.deposit(
      ICLTBase.DepositParams({
        strategyId: getStrategyID(address(this), 1),
        amount0Desired: 100 ether,
        amount1Desired: 100 ether,
        amount0Min: 0,
        amount1Min: 0,
        recipient: specialUsers[1]
      })
    );
    emit log("Deposit 1st user thru A51 strategy");
    emit log_named_uint("1st user balanceOf t0",
token0.balanceOf(specialUsers[1]));
    emit log_named_uint("1st user balanceOf t1",
token1.balanceOf(specialUsers[1]));

    (, int24 currTick,,,,,) = pool.slot0();
    emit log("\nFor the record: ");
    emit log_named_int("\tCurrent Tick from slot0 BEFORE swaps: \t",
currTick);

    emit log_named_uint("\t Pool T0 balance: \t",
token0.balanceOf(address(pool)));
    emit log_named_uint("\t Pool T1 balance: \t",
token1.balanceOf(address(pool)));

    emit log_named_uint("Swapper Balance T0", token0.balanceOf(user3[1]));
    emit log_named_uint("Swapper Balance T1", token1.balanceOf(user3[1]));

    emit log("\n====================First
Deposit====================\n");
    for (uint256 i = 0; i < 2; i++) {
```

```
        vm.warp(block.timestamp + 1 minutes);
        //swap t0 for t1
        vm.prank(user3[1]);

        router.exactInputSingle(
            ISwapRouter.ExactInputSingleParams({
                tokenIn: address(token0),
                tokenOut: address(token1),
                fee: 500,
                recipient: user3[1],
                deadline: block.timestamp + 1 hours,
                amountIn: 49.999 ether,
                amountOutMinimum: 0,
                sqrtPriceLimitX96: 0
            })
        );
    }
    console.log("\n Record: 2 x 1 ether swaps for t0->t1 \n");

    (, currTick,,,,,) = pool.slot0();
    emit log("For the record: ");
    emit log_named_int("\tCurrent Tick from slot0 AFTER swaps: \t",
currTick);

    emit log_named_uint("\t Pool T0 balance: \t",
token0.balanceOf(address(pool)));
    emit log_named_uint("\t Pool T1 balance: \t",
token1.balanceOf(address(pool)));

    emit log_named_uint("Swapper Balance T0", token0.balanceOf(user3[1]));
    emit log_named_uint("Swapper Balance T1", token1.balanceOf(user3[1]));

    emit log("\n====================First Swap====================\n");

    vm.prank(specialUsers[2]);
    (uint256 tokenIdOf2ndDepositor,, uint256 amount0, uint256 amount1) =
base.deposit(
        ICLTBase.DepositParams({
            strategyId: getStrategyID(address(this), 1),
            amount0Desired: 100 ether,
```

```
            amount1Desired: 100 ether,
            amount0Min: 0,
            amount1Min: 0,
            recipient: specialUsers[2]
        })
    );

    vm.warp(block.timestamp + 1 minutes);

    emit log("\n Deposit 2nd user thru A51 strategy");
    emit log_named_uint("2nd user balanceOf t0",
token0.balanceOf(specialUsers[2]));
    emit log_named_uint("2nd user balanceOf t1",
token1.balanceOf(specialUsers[2]));

    emit log_named_uint("\t Pool T0 balance: \t",
token0.balanceOf(address(pool)));
    emit log_named_uint("\t Pool T1 balance: \t",
token1.balanceOf(address(pool)));

    emit log("\n====================Second
Deposit====================\n");

    for (uint256 i = 0; i < 2; i++) {
        vm.warp(block.timestamp + 1 minutes);
        //swap t0 for t1
        vm.prank(user3[1]);
        router.exactInputSingle(
            ISwapRouter.ExactInputSingleParams({
                tokenIn: address(token1),
                tokenOut: address(token0),
                fee: 500,
                recipient: user3[1],
                deadline: block.timestamp + 1 hours,
                amountIn: 75 ether,
                amountOutMinimum: 0,
                sqrtPriceLimitX96: 0
            })
        );
    }
```

```
        console.log("\n Record: 2 x 1 ether swaps for t0->t1 \n");

        (, currTick,,,,,) = pool.slot0();
        emit log("For the record: ");
        emit log_named_int("\tCurrent Tick from slot0 AFTER swaps: \t",
currTick);

        emit log_named_uint("\t Pool T0 balance: \t",
token0.balanceOf(address(pool)));
        emit log_named_uint("\t Pool T1 balance: \t",
token1.balanceOf(address(pool)));

        emit log_named_uint("Swapper Balance T0", token0.balanceOf(user3[1]));
        emit log_named_uint("Swapper Balance T1", token1.balanceOf(user3[1]));
        emit log("\n====================Second
Swap====================\n");

        (, int24 tick,,,,,) = pool.slot0();
        int24 tickSpacing = key.pool.tickSpacing();

        tick = utils.floorTicks(tick, tickSpacing);

        ICLTBase.StrategyKey memory newKey = ICLTBase.StrategyKey({ pool:
pool, tickLower: -200, tickUpper: 0 });

        (,,,,,,,, ICLTBase.Account memory accountStrategy1) =
base.strategies(getStrategyID(address(this), 1));

        // invalid ticks stored here because we are trying to add in range liquidity
with only 1 asset
        vm.prank(base.owner());
        base.toggleOperator(base.owner());
        vm.prank(base.owner());
        base.shiftLiquidity(
            ICLTBase.ShiftLiquidityParams({
                key: newKey,
                strategyId: getStrategyID(address(this), 1),
                shouldMint: true,
                zeroForOne: false,
                swapAmount: int256(accountStrategy1.balance1 / 2),
```

```
            moduleStatus: "",
            sqrtPriceLimitX96: 0
        })
    );

    (, currTick,,,,,) = pool.slot0();
    emit log("For the record: ");
    emit log_named_int("\tCurrent Tick from slot0 AFTER swaps: \t",
currTick);

    emit log_named_uint("\t Pool T0 balance: \t",
token0.balanceOf(address(pool)));
    emit log_named_uint("\t Pool T1 balance: \t",
token1.balanceOf(address(pool)));

    emit log("\n====================First
ShiftBase====================\n");
  }
```

REMEDY:
It is strongly recommended to incorporate a checkDeviation modifier in all critical functions (deposit, withdraw, and shiftLiquidity) to mitigate the risks associated with price manipulation and frontrunning. This safeguard should compare the current tick against a TWAP to ensure that the operation proceeds only if the price deviation is within acceptable limits (aka THRESHOLD).

3. Incomplete Implementation in shiftLeft and shiftRight

STATUS: FIXED

DESCRIPTION

Both shiftLeft and shiftRight functions appear to be missing return statements for certain execution paths. Specifically, if the current tick is not below the lower tick (for shiftLeft) or above the upper tick (for shiftRight), the function does not return any values, leading to potential undefined behavior or compilation errors.

In the scenario where currentTick is equal to any boundary tick, the shift function in our liquidity management contract fails to execute its intended logic due to its conditional check. As a result, when currentTick matches the boundary of our defined liquidity range, the function bypasses its logic designed to adjust tick positions, inadvertently returning zero (0) for both tickLower and tickUpper due to uninitialized return values. This behavior leads to failed transactions because the adjustment logic is not triggered, and users are not informed of the reason for failure through a clear error message.

CODE AFFECTED

```
if (currentTick < key.tickLower) {
    // Logic to adjust tickLower and tickUpper
}
```

Problematic Outcome:

- Transactions Fail Silently: The logic intended to shift liquidity in response to market conditions is skipped, and the function returns zeros without executing any adjustment.
- Lack of Error Messaging: Users receive no feedback or error message indicating why the liquidity adjustment did not occur, leading to confusion.

REMEDY

Error Handling for Unchanged Ticks: Implement a mechanism to check for and handle scenarios where tick adjustments result in no change (e.g., returning initial values or preventing transaction execution), including providing clear error messages to the user.

4. **Insufficient Error Messaging in require Statements for Authorization Checks**

   STATUS: FIXED

   DESCRIPTION
   The _authorization and _authorizationOfStrategy functions use require statements for authorization checks without providing error messages. This absence can confuse users during transaction failures, as no specific reason is disclosed. Additionally, throughout the codebase there exist multiple scenarios where the require statements do not print suitable error messages and thus making the debugging process difficult to track.

   Severity:
   Low - The functionality is intact, but user experience suffers due to unclear feedback on authorization failures.

   CODE AFFECTED

```
function _authorization(uint256 tokenID) private view {
    require(ownerOf(tokenID) == _msgSender());
}

function _authorizationOfStrategy(bytes32 strategyId) private view {
    if (strategies[strategyId].isPrivate) {
        require(strategies[strategyId].owner == _msgSender());
    }
}
```

   REMEDY
   Add descriptive error messages to require statements to clarify the cause of transaction rejections.

5. **Implementing Dual-Signature Fail-Safe Mechanism for Emergency Withdrawals**

   STATUS: ACKNOWLEDGED

   DESCRIPTION:
   To enhance the security and trust in our system, especially concerning emergency withdrawal functionalities that are typically reserved for administrators, we propose implementing a dual-signature failsafe mechanism. This mechanism requires both the user's and the system administrator's signatures for withdrawals, ensuring mutual dependence and enhanced trust.

   RATIONALE:
   Current concerns revolve around the risk associated with granting exclusive emergency withdrawal rights to administrators, which could potentially lead to theft or misuse. By introducing a mechanism where both the user initiates a withdrawal request via a signature and the administrator co-signs this request, we create a balanced trust model. This approach ensures that:

   - Users retain control and oversight over their assets.
   - Administrators can facilitate emergency withdrawals without unilateral control, mitigating the risk of abuse.

   MECHANISM:
   The suggested FailSafe contract utilizes EIP-712 typed data signing to securely process withdrawal requests. The process involves:

   - **User Request:** Users generate a signed withdrawal request containing their tokenID and sender address.
   - **Administrator Approval:** The system verifies the user's signature and requires a subsequent signature from the system administrator, confirming the request.

- **Emergency Withdrawal Execution:** Upon successful verification of both signatures, the emergency withdrawal logic is executed.

This method ensures that withdrawals in emergency scenarios are executed with consent from both parties, thereby reinforcing security and mutual trust.

SECURITY AND TRUST BENEFITS:

- **Mutual Dependency:** By requiring consent from both users and administrators, the system mitigates risks associated with unilateral access to emergency withdrawal functionalities.
- **Enhanced Transparency:** The dual-signature requirement ensures transparent processing of emergency withdrawals, with traceable approvals from both parties.
- **Reduced Risk of Misuse:** The added layer of verification by both user and administrator acts as a deterrent against potential misuse or theft, protecting user assets.