Qubit

Multiplexer

Smart Contracts Security Audit

rev 1.1

Prepared for
Mound

Prepared by
Theori

December 13, 2021

Theori

Table of Contents

Executive Summary

Theori reviewed the xLP (Leveraged Liquidity Provider) feature in Qubit. This new feature allows users to enter a leveraged position in a liquidity pool by borrowing from the Qubit markets. Three critical issues were identified that could result in a loss of Qubit's assets. Two high severity issues were identified that could result in an unkillable position, i.e., it could not be liquidated. Fixes for these issues along with the recommendations were also reviewed.

Scope

The scope of this review was limited to the xLP feature in Qubit. It does not review prior code nor code that is part of other products.

Source code:

- mound/audit/qubit
    - 2f240da8be169586d3bddfbe2b6d725de30279b2

Fixes:

- mound/audit/qubit
    - 98c8ce60fb17b52504ca8fa13a9717df1cfab75e
    - 89b52592add38b0697dc39a6cffa8a168bbc0d1c
    - 7bd309a29b22d87d750e5f63bf309c1a4d030758

Findings

These are the potential issues that may have correctness and/or security impacts.

Summary

#	ID	Title	Severity
1	THE-XLP-001	`setPid` is missing `onlyOwner` modifier	Critical
2	THE-XLP-002	`setDebtRatioLimit` is missing `onlyOwner` modifier	Critical
3	THE-XLP-003	`removePosition` requires O(n) store operations	High
4	THE-XLP-004	`kill` may revert during `_transferOut` to position owner	High
5	THE-XLP-005	`_withdrawAndRepay` allows position owner to remove principal without repaying debt	Critical
6	THE-XLP-006	`_harvest` will incorrectly collect all QBT rewards for a token and add them to the current pool	Medium
7	THE-XLP-007	`_borrowAndSwap` ignores return value of `_transferIn`	Low

Issue #1: setPid is missing onlyOwner modifier

ID	Summary	Severity
THE-XLP-001	*setPid* is missing the *onlyOwner* modifier which allows an attacker to steal funds	Critical

In QPositionManager.sol, *setPid* is used to set the pool ID for a liquidity pool token address. The pool ID is required to deposit and withdraw the liquidity pool tokens with PancakeSwap's MasterChef contract. It is also the only way that a liquidity pool token address is authenticated (see Recommendation #1). If an attacker can change the pool ID, they could use a fake liquidity pool token contract. This would allow them to steal funds from Qubit by minting many fake liquidity pool tokens that are used as collateral for a loan of real assets.

Fix

Mound added the *onlyOwner* modifier to *setPid*.

```
function setPid(address pool, uint pid) external onlyOwner {
    PoolInfo storage poolInfo = pools[pool];
    poolInfo.pid = pid;
}
```

Issue #2: setDebtRatioLimit is missing onlyOwner modifier

ID	Summary	Severity
THE-XLP-002	*setDebtRatioLimit* is missing the *onlyOwner* modifier	Critical

In QPositionManager.sol, *setDebtRatioLimit* is used to set the debt ratio limit for a liquidity pool. The new debt limit must be in the range [0, 10000) so it is not possible to use this issue to directly steal funds; an attacker can only use it to borrow up to 99.99% of the principal. For example, with a debt ratio limit of 99.99%, an attacker can deposit 1 USD, borrow 9999 USD, and create a 10000 USD position. Also, an attacker could lower the debt limit of a pool which may trigger the liquidation of other users' positions. This could result in losses for users.

Fix

Mound added the *onlyOwner* modifier to *setDebtRatioLimit*.

```
function setDebtRatioLimit(address pool, uint limit) external onlyOwner {
    require(limit < 10000, "QPositionManager:: invalid limit value");
    PoolInfo storage poolInfo = pools[pool];
    poolInfo.debtRatioLimit = limit;
}
```

Issue #3: removePosition requires O(n) store operations

ID	Summary	Severity
THE-XLP-003	*removePosition* requires O(n) store operations which can be manipulated by an attacker to prevent liquidation of a position	High

In QPositionManager.sol, *removePosition* clears a position and removes it from the user's list of positions. Removing an arbitrary element from the user's list of positions requires O(n) store operations, e.g., removing the first position from the list requires doing as many stores as positions in the list. Since the number of positions is controlled by the user, and attacker could create many positions so that any attempt to execute *removePosition* will revert due to not enough gas.

This issue become problematic because *removePosition* is called from the *kill* function which is used to liquidate a position. If *removePosition* reverts, then the position cannot be liquidated. This may result in a loss of funds from the protocol.

Fix

Mound added a limit on the length of the user's position list to a small number (10).

```
function pushPosition(address pool, address account, uint id) external override
onlyMultiplexer {
    UserInfo storage userInfo = users[account];
    userInfo.positionList[pool].push(id);
    pools[pool].lastPositionID += 1;
    require(userInfo.positionList[pool].length <= MAX_POS_NUM, "QPositionManager::
MAX_POS_NUM");
}
```

Issue #4: kill may revert during _transferOut to position owner

ID	Summary	Severity
THE-XLP-004	*kill* may transfer BNB to the position owner which can be abused by an attacker to prevent liquidation of a position	High

In QMultiplexer.sol, *kill* liquidates a position whose debt ratio is higher than the debt ratio limit for the liquidity pool. Liquidation does not seize all of the position's collateral, only the collateral necessary to repay the debt and the liquidation incentive. The remaining collateral must be refunded to the position's owner.

```
function kill(address pool, uint id) external onlyWhitelisted nonReentrant {
    require(positionManager.debtRatioOf(pool, id) >
positionManager.debtRatioLimit(pool), "QMultiplexer: Not yet to liquidate");

    (uint token0Refund, uint token1Refund) = _withdrawAndRepay(pool, id,
positionManager.balanceOf(pool, id), positionManager.debtValOfPosition(pool, id));

    (address _token0, address _token1) = positionManager.getBaseTokens(pool);
    address positionOwner = positionManager.getPositionOwner(pool, id);
    uint token0Incentive = token0Refund.mul(incentiveBp).div(10000);
    uint token1Incentive = token1Refund.mul(incentiveBp).div(10000);

    _transferOut(_token0, msg.sender, token0Incentive);
    _transferOut(_token1, msg.sender, token1Incentive);
    _transferOut(_token0, positionOwner, token0Refund.sub(token0Incentive));
    _transferOut(_token1, positionOwner, token1Refund.sub(token1Incentive));

    positionManager.notifyLiquidated(pool, id, token0Refund.sub(token0Incentive),
token1Refund.sub(token1Incentive));
    positionManager.removePosition(pool, positionOwner, id);
    emit KillPosition(pool, id, token0Incentive, token1Incentive);
}
```

If the position's collateral includes BNB, then it is transferred to the position owner with the *safeTransferETH* function, which calls the owner address with the appropriate value. If the call to the owner address reverts, then *safeTransferETH* will also revert causing the entire liquidation transaction to revert. An attacker can use this to prevent liquidations of their positions and cause the protocol to lose funds.

Fix

Mound changed the *kill* function to no longer send collateral back to the position owner. Instead, the position owner must call the *liqudationRefund* function to retrieve their collateral.

```
function kill(address pool, uint id) external onlyWhitelisted nonReentrant {
    ...

    _transferOut(_token0, msg.sender, token0Incentive);
    _transferOut(_token1, msg.sender, token1Incentive);

    positionManager.notifyLiquidated(pool, id, token0Refund.sub(token0Incentive),
token1Refund.sub(token1Incentive));
    emit KillPosition(pool, id, token0Incentive, token1Incentive);
}
```

```
function liquidationRefund(address pool, uint id) public onlyPositionOwner(pool, id)
nonReentrant returns (uint, uint) {
    (bool isLiquidated, uint[] memory refundAmount) =
positionManager.getLiquidationInfo(pool, id);
    require(isLiquidated, "QMultiplexer: not liquidated yet");

    (address _token0, address _token1) = positionManager.getBaseTokens(pool);
    address positionOwner = positionManager.getPositionOwner(pool, id);
    _transferOut(_token0, positionOwner, refundAmount[0]);
    _transferOut(_token1, positionOwner, refundAmount[1]);

    positionManager.removePosition(pool, msg.sender, id);
    emit Withdrawn(msg.sender, pool, id, 0, refundAmount[0], refundAmount[1]);
    return (refundAmount[0], refundAmount[1]);
}
```

Issue #5: Position owner can remove all principal without repaying debt

ID	Summary	Severity
THE-XLP-005	Position owner can remove all principal without repaying debt, which will cause *debtRatioOf* to return 0 and bypass the *checkDebtRatio* validation	Critical

In QMultiplexer.sol, *reducePosition* can be used by users to reduce their debt and withdraw collateral. Afterwards, the *checkDebtRatio* modifier will validate that the debt ratio is less than the debt ratio limit. However, the *debtRatioOf* function will always return 0 if there is no collateral, so an attacker can withdraw all of their collateral without repaying any debt. This could result in a significant loss of funds from Qubit.

```
modifier checkDebtRatio(address pool, uint id) {
    _;
    require(positionManager.debtRatioOf(pool, id) <=
positionManager.debtRatioLimit(pool), "QMultiplexer: Exceed debtRatioLimit");
}
```

```
function debtRatioOf(address pool, uint id) public override view returns (uint) {
    uint positionBalance = balanceOf(pool, id);

    if (positionBalance == 0) return 0;

    (address _token0, address _token1) = getBaseTokens(pool);
    uint[] memory positionDebt = debtValOfPosition(pool, id);
    (, uint positionValInUSD) = priceCalculator.valueOfAsset(pool, positionBalance);
    (, uint token0DebtValInUSD) = priceCalculator.valueOfAsset(_token0,
positionDebt[0]);
    (, uint token1DebtValInUSD) = priceCalculator.valueOfAsset(_token1,
positionDebt[1]);

    return token0DebtValInUSD.add(token1DebtValInUSD).mul(1e18).div(positionValInUSD);
}
```

Fix

Mound added a new check to the *debtRatioOf* function. If the position has no collateral but has debt, then the function returns infinite, i.e., *uint(-1)*.

```
function debtRatioOf(address pool, uint id) public override view returns (uint) {
    uint positionBalance = balanceOf(pool, id);

    (address _token0, address _token1) = getBaseTokens(pool);
    uint[] memory positionDebt = debtValOfPosition(pool, id);

    // early exit to avoid zero-division
    if (positionBalance == 0) return positionDebt[0].add(positionDebt[1]) == 0 ? 0 :
uint (-1);

    (, uint positionValInUSD) = priceCalculator.valueOfAsset(pool, positionBalance);
    (, uint token0DebtValInUSD) = priceCalculator.valueOfAsset(_token0,
positionDebt[0]);
    (, uint token1DebtValInUSD) = priceCalculator.valueOfAsset(_token1,
positionDebt[1]);

    return token0DebtValInUSD.add(token1DebtValInUSD).mul(1e18).div(positionValInUSD);
}
```

Issue #6: _harvest collects all QBT rewards for a token and gives them to the current pool

ID	Summary	Severity
THE-XLP-006	*harvest* collects QBT rewards for a token but does not fairly distribute these rewards to the appropriate pool	Medium

In QMultiplexer.sol, *harvest* can be called by the contract owner to claim rewards for a pool. The rewards are harvested from Qubit and the CAKE that was collected during calls to the MasterChef contract. In commit 98c8ce60fb17b52504ca8fa13a9717df1cfab75e, all QBT claimed from a Qubit market is assigned to the currently harvested pool. If multiple pools have a token in common, then one pool will receive all of the QBT from that market and the other pools will not get any QBT from that market.

```
function _harvest(address pool) private {
    _withdraw(pool, 0);
    (address _token0, address _token1) = positionManager.getBaseTokens(pool);

    uint claimedQBT = _claimQubit(_token0);
    claimedQBT = claimedQBT.add(_claimQubit(_token1));
    if (claimedQBT > 0) {
        _compound(pool, QBT, claimedQBT);
    }

    if (pendingCake[pool] > 0) {
        uint performanceFee = pendingCake[pool].mul(performanceBp).div(10000);
        _compound(pool, address(CAKE), pendingCake[pool].sub(performanceFee));

        performanceFee = qZap.swapExactTokenForToken(address(CAKE), QBT,
performanceFee);
        pendingCake[pool] = 0;

        _transferOut(QBT, DEV_TREASURY, performanceFee);
        emit PerformanceFee(pool, performanceFee);
    }
}
```

Fix

Mound restored the QRewardBox contract which fairly distributes rewards based on the debt of each pool.

```
function _harvest(address pool) private {
    _withdraw(pool, 0);

    uint claimedQBT = qRewardBox.getReward(pool);
    if (claimedQBT > 0) {
        _compound(pool, QBT, claimedQBT);
    }

    if (pendingCake[pool] > 0) {
        uint performanceFee = pendingCake[pool].mul(performanceBp).div(10000);
        _compound(pool, address(CAKE), pendingCake[pool].sub(performanceFee));

        performanceFee = qZap.swapExactTokenForToken(address(CAKE), QBT,
performanceFee);
        pendingCake[pool] = 0;

        _transferOut(QBT, DEV_TREASURY, performanceFee);
        emit PerformanceFee(pool, performanceFee);
    }
}

function _claimQubit(address token) private {
    address market = qore.getQToken(token);
    if (market != address(0)) {
        uint _before = IBEP20(QBT).balanceOf(address(this));
        qore.claimQubit(market);
        uint claimedQBT = IBEP20(QBT).balanceOf(address(this)).sub(_before);
        qRewardBox.notifyRewardAmount(token, claimedQBT);
    }
}
```

Issue #7: _borrowAndSwap ignores return value of _transferIn

ID	Summary	Severity
THE-XLP-007	*_borrowAndSwap* ignores return value of *_transferIn*	Low

In QMultiplexerHelper.sol, *_borrowAndSwap* transfers in collateral from the user, borrows from Qubit, and then mints liquidity pool tokens which are deposited into the MasterChef contract. However, *_transferIn* does not guarantee that the entire amount is transferred, instead it returns the actual amount transferred. Ignoring the return value of *_transferIn* is almost always going to be an issue.

```
function _borrowAndSwap(
    uint id,
    address pool,
    uint[] memory depositAmount,
    uint[] memory borrowAmount
) internal returns (uint){
    (address _token0, address _token1) = positionManager.getBaseTokens(pool);
    uint amount;

    if (depositAmount[0] > 0) {
        _transferIn(_token0, msg.sender, depositAmount[0]);
    }
    if (depositAmount[1] > 0) {
        _transferIn(_token1, msg.sender, depositAmount[1]);
    }
    if (depositAmount[2] > 0) {
        amount = amount.add(_transferIn(pool, msg.sender, depositAmount[2]));
    }
    ...
}
```

The impact of ignoring the return value of *_transferIn* is minimal here because this contract should never have a balance of BNB or tokens. But is easy to fix and may prevent future issues.

Fix

Mound added a check of the return value to ensure that it matches the expected amount.

```
function _borrowAndSwap(
```

```
    uint id,
    address pool,
    uint[] memory depositAmount,
    uint[] memory borrowAmount
) internal returns (uint){
    (address _token0, address _token1) = positionManager.getBaseTokens(pool);
    uint amount;

    if (depositAmount[0] > 0) {
        require(depositAmount[0] == _transferIn(_token0, msg.sender,
depositAmount[0]), "QMultiplexer: insufficient token0 transferred");
    }
    if (depositAmount[1] > 0) {
        require(depositAmount[1] == _transferIn(_token1, msg.sender,
depositAmount[1]), "QMultiplexer: insufficient token1 transferred");
    }
    if (depositAmount[2] > 0) {
        require(depositAmount[2] == _transferIn(pool, msg.sender, depositAmount[2]),
"QMultiplexer: insufficient LP token transferred");
        amount = amount.add(depositAmount[2]);
    }
    ...
}
```

Recommendations

These are the recommendations to improve the code quality for better readability, optimization, and security. They do not impose any immediate security impacts.

Summary

#	Title	Type	Importance
1	*createPosition* should require that pool is a valid liquidity pool address	Code Quality	Minor
2	*removePosition* should delete the PositionInfo struct	Code Quality	Minor

Recommendation #1: createPosition should require that pool is a valid liquidity pool address

If an attacker can create a position with an attacker constructed fake liquidity pool token, they may be able to steal funds from Qubit by minting many fake liquidity pool tokens that are used as collateral for a loan of real assets. This is currently prevented implicitly by the PancakeSwap MasterChef contract, because *pidOf* will return 0 for non-supported liquidity pool tokens and the MasterChef contract's deposit and withdraw functions will revert if *pid* is 0.

We recommend that *createPosition* explicitly checks that *pidOf* is not 0 as a preemptive measure.

Fix

Mound validates that *pidOf* is not 0 in the _deposit and _withdraw functions.

```
function _deposit(address pool, uint amount) internal updateCakeHarvested(pool) {
    uint pid = positionManager.pidOf(pool);
    require(pid != 0, "QMultiplexer: invalid staking token");

    _approveIfNeeded(pool, address(CAKE_MASTER_CHEF));
    CAKE_MASTER_CHEF.deposit(pid, amount);
}

function _withdraw(address pool, uint amount) internal updateCakeHarvested(pool)
returns (uint amountOut) {
    uint pid = positionManager.pidOf(pool);
    require(pid != 0, "QMultiplexer: invalid staking token");

    uint _before = IBEP20(pool).balanceOf(address(this));
    CAKE_MASTER_CHEF.withdraw(pid, amount);
    amountOut = IBEP20(pool).balanceOf(address(this)).sub(_before);
}
```

Recommendation #2: removePosition should delete the PositionInfo struct

After the fix for issue #4, some of the fields in the PositionInfo struct are not zeroed in *removePosition*. This can be partially mitigated by using the delete keyword to reset the struct's non-mapping fields to their default values, e.g., zero. The mapping fields must still be zeroed explicitly, however.

We recommend using the delete keyword, when possible, instead of explicitly zeroing every field in the struct.

Fix

Mound changed *removePosition* to use the delete keyword instead of zero each individual field. Also, added zeroing of the *liquidateAmount* mapping.

```
function removePosition(address pool, address account, uint id) external override
onlyMultiplexer {
    uint[] storage positionList = users[account].positionList[pool];

    require(positionList.length > 0, "QPositionManager:: empty position list");

    if (id != positionList[positionList.length - 1]) {
        uint index = _findValue(positionList, id);
        for (uint i = index; i < positionList.length - 1; i++) {
            positionList[i] = positionList[i + 1];
        }
    }
    positionList.pop();

    delete pools[pool].positions[id];
    delete pools[pool].positions[id].debtShare[IPancakePair(pool).token0()];
    delete pools[pool].positions[id].debtShare[IPancakePair(pool).token1()];
    delete pools[pool].positions[id].liquidateAmount[IPancakePair(pool).token0()];
    delete pools[pool].positions[id].liquidateAmount[IPancakePair(pool).token1()];
}
```