Qubit

Smart Contracts Security Audit

revision 1.1

Prepared for
Mound

Prepared by
Theori

September 24th, 2021

Theori

Table of Contents

Executive Summary

Starting on August 24, 2021, Theori assessed Qubit, a lending protocol for the Binance Smart Chain. Qubit enables users to earn interest on their cryptocurrency deposits by lending the assets to borrowers. The Qubit governance token is rewarded to both borrowers and suppliers based on amount and asset transacted and staked tokens. We focused on identifying issues that result in a loss of deposited assets or ways for users to cheat and earn excess interest or rewards. We found no immediate issues and the code is well-written.

Scope

Qubit, a lending platform for Binance Smart Chain, is audited. The associated PancakeBunny vaults are also audited.

Source code:

- PancakeBunny-finance/qubit-finance (41aee4f821194342beb548f2d61f6ecd8e940c03)
- PancakeBunny-finance/Bunny (0972f4de6ccdde5c20033dff4596bd1a4e68e79a)
    - contracts/vaults/qubit

Fixes:

- PancakeBunny-finance/qubit-finance
    - 83f2247fca1a263cf5bcd74588acec7c7ba7e4ab
    - d7a209b1c282efbcb4dbe262b380cb98cd31ecd5
    - da02b8a82474a0c6b8fc958cf0a2bcbaede34a49

Overview

The assessment of the smart contracts focused on threats to collateral, methods to bypass interest or liquidation, and improper rewards. We did not identify any current issues in the smart contract code.

We systematically reviewed the Qore and QToken smart contracts. We verified that:

- Every function in Qore checks that passed in tokens are listed in the market
- `_enterMarket` prevents user from joining the same market twice
- `exitMarket` checks that `borrowBalance` is zero and borrower has enough liquidity to remove the collateral
- `borrow` checks that the user is in the market for the borrowed token
- `liquidateBorrow` only allows borrower's collateral tokens
- `supply`, `redeemToken`, and `redeemUnderlying` call `notifySupplyUpdated`
- `borrow`, `repayBorrow`, and `repayBorrowBehalf` call `notifyBorrowUpdated`
- `seize` and `_transferTokens` call `notifyTransferred`

Additionally, we reviewed the function modifiers and authentication checks:

- Mutative functions in Qore have `nonReentrant` modifier
- Mutative functions in Qore only modify `msg.sender` account (except for liquidation)
- Functions in QoreAdmin.sol have `onlyKeeper` modifier
- Initializer functions have `initializer` modifier

We also reviewed QDistributor which is responsible for rewards:

- User rewards are updated before new borrow and supply values are used
- Notification functions cannot be manipulated by an attacker
    - They can be called multiple times without adverse effects
    - Notifications must be called after any modification of state
- `claimQubit` always sends rewards to correct `account`

Risks

The risks to collateral are similar to other lending platforms: compromise of admin keys, unknown smart contract vulnerabilities, and monetary risks. There are two admin keys in Qubit: the owner key and the keeper key. Both are capable of changing parameters, such as listed tokens in the market, which could be used to steal collateral. These keys must be secured and should be stored in a cold wallet, preferably with multisig to require multiple signatories.

Smart contract vulnerabilities are a threat to every DeFi platform. Mound has proactively open sourced the Qubit source code and hired companies to review it for vulnerabilities. They also provide a bug bounty on Immunefi to incentivize white hats to report vulnerabilities in Qubit. While the risk of vulnerabilities will never go away, these are good steps to mitigate the risk.

Lastly, a DeFi lending platform relies on the ability of liquidators to quickly liquidate unsafe positions. If markets conditions change rapidly, it may be possible for the platform to have underwater positions that need to be liquidated by admins. This will reduce the available reserves and possibly other users' collateral. Mound should carefully review the historical volatility of an asset before it is listed on the market.

Findings

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

Summary

#	ID	Title	Severity
1	THE-QUBIT-001	Non-reentrancy of QToken transfer functions	Medium
2	THE-QUBIT-002	Liquidation of ERC777 tokens and closeFactor	Minor

Issue #1: Non-reentrancy of QToken transfer functions

ID	Summary	Severity
THE-QUBIT-001	QToken and Qore functions are independently non-reentrant. There are no known vulnerabilities but transfer functions should not be callable during a Qore operation.	Medium

In QToken.sol, *transfer* and *transferFrom* are modified with *nonReentrant* but this is independent of the *nonReentrant* modifier in Qore. This allows someone to call the transfer functions even if Qore is in the middle of an operation. While not currently exploitable, it is risky to allow this interaction.

Recommendations

QToken external functions should call into a non-reentrant function in Qore to prevent reentrancy. For instance, the external mutative functions *transfer* and *transferFrom* could call "transferTokens" function in Qore which then calls "transferTokensInternal" in QToken. As an added benefit, the calls to QDistributor.notifyTransferred could be moved from QToken to Qore.

Fix

Mound applied the recommendations to Qubit.

They created a new Qore function, *transferTokens*, which can only be called by a market token and calls back into the market to transfer the tokens. Since this new function has the *nonReentrant* modifier, it cannot be called during another Qore operation. Also, the calls to *notifyTransferred* were moved into Qore.

```
function transferTokens(address spender, address src, address dst, uint amount)
external override nonReentrant onlyMarket {
    IQToken(msg.sender).transferTokensInternal(spender, src, dst, amount);
    qDistributor.notifyTransferred(msg.sender, src, dst);
}
```

The transfer functions in QToken now call into Qore and there is a new external function, *transferTokensInternal*. This function can only be called by Qore and is the same as the old *_transferTokens* function.

```
function transfer(address dst, uint amount) external override accrue nonReentrant
returns (bool) {
    qore.transferTokens(msg.sender, msg.sender, dst, amount);
    return true;
}

function transferFrom(
    address src,
    address dst,
    uint amount
) external override accrue nonReentrant returns (bool) {
    qore.transferTokens(msg.sender, src, dst, amount);
    return true;
}

function transferTokensInternal(
        address spender,
        address src,
        address dst,
        uint amount
) external override onlyQore {
    ...
}
```

Lastly, as the calls to *notifyTransferred* are now in Qore, QDistributor can use the *onlyQore* modifier:

```
function notifyTransferred(
    address qToken,
    address sender,
    address receiver)
external override
nonReentrant onlyQore
updateDistributionOf(qToken) {
    ...
}
```

Issue #2: Liquidation of ERC777 tokens and closeFactor

ID	Summary	Severity
THE-QUBIT-002	Transfer of ERC777 tokens may invoke external contracts that could transfer additional tokens. A liquidator could use this to bypass closeFactor limits.	Minor

In QToken.sol, *_doTransferIn* transfers in *amount* tokens from *underlying* and returns the actual amount received:

```
uint balanceBefore = IBEP20(underlying).balanceOf(address(this));
underlying.safeTransferFrom(from, address(this), amount);
return IBEP20(underlying).balanceOf(address(this)).sub(balanceBefore);
```

This is generally safe and recommended code. However, if the underlying token implements ERC777, then it may allow an external contract to execute and send more than *amount* tokens. This will result in *_repay* repaying more of the borrowed balance. The liquidation repayment amount is supposed to be limited by *closeFactor*, but with this the liquidator could repay more and earn more profit with the liquidation bonus.

Recommendations

This issue does not need to be addressed until tokens that implement ERC777 are supported. Most BEP20 tokens do not implement ERC777. One mitigation would be to require *balanceAfter − balanceBefore <= amount* in *_doTransferIn*.

Fix

Mound applied the recommendations to Qubit.

In QToken, *_doTransferIn* requires that the amount transferred is not more than *amount*.

```
uint balanceBefore = IBEP20(underlying).balanceOf(address(this));
underlying.safeTransferFrom(from, address(this), amount);
uint balanceAfter = IBEP20(underlying).balanceOf(address(this));
require(balanceAfter.sub(balanceBefore) <= amount);
return balanceAfter.sub(balanceBefore);
```

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	updateSupplyInfo function is unused	Code Quality	Minor
2	_removeUserMarket should update usersOfMarket	Code Quality	Minor

Recommendation #1: updateSupplyInfo function is unused

In QMarket.sol and QToken.sol, the *updateSupplyInfo* function is unused. Instead, updates to *totalSupply* and *accountBalances* are inlined in *supply* and *_redeem*. Consider using *updateSupplyInfo* in these functions to mirror the usage of *updateBorrowInfo*.

Fix

In QToken.sol, *supply* and *_redeem* now use *updateSupplyInfo*.

```
 function supply(address account, uint uAmount) external payable override accrue
onlyQore returns (uint) {
     ...
-    totalSupply = totalSupply.add(qAmount);
-    accountBalances[account] = accountBalances[account].add(qAmount);
+    updateSupplyInfo(account, qAmount, 0);
     ...
 }

 function _redeem(address account, uint qAmountIn, uint uAmountIn) private returns
(uint) {

     ...
-    totalSupply = totalSupply.sub(qAmountToRedeem);
-    accountBalances[account] = accountBalances[account].sub(qAmountToRedeem);
+    updateSupplyInfo(account, 0, qAmountToRedeem);
     ...
 }
```

Recommendation #2: _removeUserMarket should update usersOfMarket

In Qore.sol, the *_removeUserMarket* function removes a token from a user's list in *marketListOfUsers*. However, it does not update *usersOfMarket* to reflect that the user was removed from the market. Instead, the caller, *exitMarket*, updates *usersOfMarket*. This is different from the behavior of *_enterMarket* which updates both at the same time.

Consider moving the code to update *usersOfMarket* to *_removeUserMarket* so it mirrors *_enterMarket*. Alternatively, move the code that updates *marketListOfUsers* to *exitMarket*.

Fix

In Qore.sol, the update of *usersOfMarket* was moved to *_removeUserMarket*:

```
function _removeUserMarket(address qTokenToExit, address _account) private {
    require(marketListOfUsers[_account].length > 0, "Qore: cannot pop user market");
    delete usersOfMarket[qTokenToExit][_account];

    uint length = marketListOfUsers[_account].length;
    for (uint i = 0; i < length; i++) {
        if (marketListOfUsers[_account][i] == qTokenToExit) {
            marketListOfUsers[_account][i] = marketListOfUsers[_account][length - 1];
            marketListOfUsers[_account].pop();
            break;
        }
    }
}
```