Benqi Avalanche Ecosystem Markets Review



Table of Contents

Need logo **1**
Table of Contents **2**
Scope **2**
Summary of Findings **2**
Issues **4**
BQ-001: Supplied tokens that are not entered into markets can be seized through liquidation 4
BQ-002: Loss of precision in liquidateCalculateSeizeTokens due to division before multiplication 5
BQ-003: Entering too many markets can result in out of gas in getHypotheticalAccountLiquidityInternal 6
BQ-004: sweepToken can allow admin to sweep tokens with more than one entry point 7
BQ-005: mintWithPermit is vulnerable to ERC-2612 frontrunning 7
BQ-006: Exchange rate manipulation of QiTokens when the total supply is 0 8
BQ-007: protocolSeizeShareMantissa is a default of 0, and has no upper cap 10
BQ-008: Underlying tokens with more than 18 decimals will revert in the ChainlinkOracle when getting underlying price 10
BQ-009: Upgradable versions of OpenZeppelin contracts should be used for MultiRewardDistributor 11
BQ-010: Loss of precision when calculating borrowerDelta 11
BQ-011: Deprecated markets cannot be liquidated 12

Scope

The scope of the audit is limited to https://github.com/Benqi-fi/isolated-markets. The reviewed commit was bd47157401a5918adb1c0e3d93c23f69ed6d1261.

This report will also document the changes and fixes made before the review commit.

Summary of Findings

In performing a security audit of the Benqi Ecosystem Market, several issues of concern were found. For each finding, a summary of the issue is documented, along with any other finer details regarding the issue. Security recommendations are also provided where applicable.

The table below shows a breakdown of security findings found categorized by severity or risk and impact. A finding that has been reported is listed as pending, and if that finding is satisfactorily mitigated, it will be categorized as resolved.

Severity	**Resolved**	**Unresolved**	**Total**
Critical	0	0	0
High	0	0	0
Medium	4	0	4
Low	1	4	5
Info	0	2	2

Issues

BQ-001: Supplied tokens that are not entered into markets can be seized through liquidation

Severity: Medium
Status: Resolved
There is an issue where qiTokens which are not entered in the market can be seized.

Example:
User has wBTC and USDC deposited into the protocol, but only entered the market for wBTC. He borrows DAI. If wBTC price goes down and his position becomes illiquid, a liquidator can seize either wBTC (expected) or USDC (not expected).

This is because there is a lack of checking if the collateral to be seized is entered in the market for an account.

Ref: https://github.com/compound-finance/compound-protocol/issues/38

Recommendations
Add checks in liquidateBorrowAllowed to ensure that the borrower has entered the market for the collateral to be seized.

```
if (!markets[qiTokenCollateral].accountMembership[borrower]) {
    return uint(Error.MARKET_NOT_ENTERED);
}
```

Resolution
The recommended check was added in

```
if (!markets[qiTokenCollateral].accountMembership[borrower]) {
    return uint(Error.MARKET_NOT_ENTERED);
}
```

Note
This issue was fixed before the review commit

BQ-002: Loss of precision in liquidateCalculateSeizeTokens due to division before multiplication

Severity: Low
Status: Acknowledged
In liquidateCalculateSeizeTokens, seizeTokens is calculated by doing division before multiplication.

```
    numerator = mul_(Exp({mantissa: liquidationIncentiveMantissa}), Exp({mantissa:
priceBorrowedMantissa}));
    denominator = mul_(Exp({mantissa: priceCollateralMantissa}), Exp({mantissa:
exchangeRateMantissa}));
    ratio = div_(numerator, denominator);

    seizeTokens = mul_ScalarTruncate(ratio, actualRepayAmount);
```

In either of the 3 cases, which can cause ratio to be zero if numerator divided by the denominator is less than 1, it can cause liquidators to not receive any of the collateral during liquidation as seizeTokens would be zero:

priceBorrowedMantissa is very low
priceCollateralMantissa is very high
exchangeRateMantissa is very high

This usually happens if the borrowed underlying asset has very few decimals and the collateral asset has many many decimals, causing the delta between them to be too large.

Recommendations
Consider doing multiplication first before division.

```
    numerator = mul_(
      mul_(
        Exp({mantissa: liquidationIncentiveMantissa}),
        Exp({mantissa: priceBorrowedMantissa})
      ),
      actualRepayAmount
    );
    denominator = mul_(
      Exp({mantissa: priceCollateralMantissa}),
      Exp({mantissa: exchangeRateMantissa})
    );
    ratio = div_(numerator, denominator);

    seizeTokens = truncate(ratio);
```

Resolution
The team acknowledged the issue. It is unlikely that the loss of precision will affect the seizeToken amount based on the tokens supported in the market.

BQ-003: Entering too many markets can result in out of gas in getHypotheticalAccountLiquidityInternal

Severity: Low
Status: Acknowledged
In getHypotheticalAccountLiquidityInternal, an account's liquidity in the protocol is calculated by looping over every asset in which the account has.

If there are a large number of markets available, this could cause out of gas issues due to excessive number of loops done when getHypotheticalAccountLiquidityInternal, and could possibly result in denial of service for certain functionality, such as denying liquidations even when an account is illiquid.

Recommendations
Consider adding a cap to the number of assets an account can enter. This cap should be checked against when an account is entering a market.

Resolution
The team acknowledged the issue. The team will keep track of the number of markets added to prevent this from being an issue.

BQ-004: sweepToken can allow admin to sweep tokens with more than one entry point

Severity: Info
Status: Acknowledged
sweepToken can cause pricing issues of the CToken if the underlying token has more than 1 address (e.g. TUSD described in the blog below).

https://medium.com/chainsecurity/trueusd-compound-vulnerability-bc5b696d29e2

By sweeping a token with a secondary address that has an entry point to the underlying token's contract functionality, it would be possible to circumvent the check that disallows the underlying tokens to be transferred out using the sweep function to the admin.

Recommendations
A recommended change would be to check the balance of the underlying before and after the transfer to ensure that the balance has not changed.

Additionally, underlying tokens should be vetted for such unexpected behavior before being added to the market.

Resolution
The team acknowledged the issue. The team will not add tokens with double entry points.

BQ-005: mintWithPermit is vulnerable to ERC-2612 frontrunning

Severity: Medium
Status: Resolved

mintWithPermit uses the permit function of ERC20 tokens that implement ERC-2612m, which is vulnerable to denial of service. This can be done by frontrunning the user's transaction and calling the ERC20 token's permission directly with the signature that the user is intending to use. It will succeed as the signature is valid, and increase the nonce for the user. Thus, when the user's transaction is executed, it will revert since the nonce would be one more than the payload which the user has signed for.

```
function mintWithPermit(
    uint mintAmount,
    uint deadline,
    uint8 v, bytes32 r, bytes32 s
) override external returns (uint) {
    IERC20Permit token = IERC20Permit(underlying);

    // Go submit our pre-approval signature data to the underlying token, but
    // explicitly fail if there is an issue.
    SafeERC20.safePermit(
        token,
        msg.sender, address(this),
        mintAmount, deadline,
        v, r, s
    );

    (uint err,) = mintInternal(mintAmount);
    return err;
```

```
}
```

Reference: https://www.trust-security.xyz/post/permission-denied

Recommendations
Remove mintWithPermit if not necessary. Otherwise, add a check if the user has already granted token allowance to the contract and skip the permit call if there is sufficient allowance.

Resolution
mintWithPermit was removed as it was unused.

Note
This issue was fixed before the review commit.

BQ-006: Exchange rate manipulation of QiTokens when the total supply is 0

Severity: Medium
Status: Resolved

After a QiToken has been deployed and added to the Comptroller as a market, a malicious actor would be able manipulate the exchange rate of the QiToken with the following steps:

- Mint QiToken using a small amount of underlying to set the initial exchange rate
- Redeem all but a small amount (e.g. 1 wei) of QiTokens
- Send a large amount of underlying tokens directly to the QiToken contact.
- As there is only a total supply of 1 wei of QiToken, but a large underlying, the exchange rate would be skewed towards an extremely high rate. This is possible as the exchange rate used is based on the underlying token balance in the QiToken contract
- Any further mints would result in loss of funds for the depositor due to loss of precision when division is done by an extremely huge divisor

Additionally, even if this attack is not done on the first deposit, it can be done if the total supply of the QiToken ever goes back to zero.

Ref: https://www.comp.xyz/t/hundred-finance-exploit-and-compound-v2/4266

Recommendations
Consider minting a minimum liquidity amount (e.g 1e4 wei) to the burn address to make such exchange rate manipulation attacks more difficult. Also, the mintTokens amount can be checked to ensure that it is not zero.

```
uint public constant MINIMUM_LIQUIDITY = 10 ** 3;

if (totalSupply == 0) {
        totalSupply = MINIMUM_LIQUIDITY;
        accountTokens[address(0)] = MINIMUM_LIQUIDITY;
        mintTokens -= MINIMUM_LIQUIDITY;
    }
```

Resolution
The team has decided to resolve this issue by burning tokens after they mint a small amount directly after deploying a new QiToken.

BQ-007: protocolSeizeShareMantissa is a default of 0, and has no upper cap

Severity: Low
Status: Acknowledged
protocolSeizeShareMantissa should have a max upper limit of the value that can be set, otherwise the protocol can seize up to 100% of the seizeTokens.

Also, do note that unlike the original compound, which set protocolSeizeShareMantissa as a constant 2.8%

```
uint public constant protocolSeizeShareMantissa = 2.8e16; //2.8%
```

The ecosystem market does not have it set and it is a default of 0. So any liquidations done until this is set to a non 0 value will not have any seize going to the protocol.

Recommendations
Consider using the same code as Compound to have protocolSeizeShareMantissa as a constant.

Resolution

The team has acknowledged the issue. protocolSeizeShareMantissa will be set after deployment.

BQ-008: Underlying tokens with more than 18 decimals will revert in the ChainlinkOracle when getting underlying price

Severity: Info
Status: Acknowledged
Underlying tokens with more than 18 decimals are not supported as they revert due to subtraction underflow

```
uint256 decimalDelta = uint256(18).sub(uint256(token.decimals()));
```

Recommendations
Consider adjusting the code to accommodate for tokens with more than 18 decimals, by subtracting the token decimals by 18 to get decimalDelta, and then dividing the price by decimalDelta.

Resolution
The team has acknowledged the issue. Tokens above 18 decimals will not be added.

BQ-009: Upgradable versions of OpenZeppelin contracts should be used for MultiRewardDistributor

Severity: Low
Status: Resolved

MultiRewardDistributor is an upgradable contract, but was using non-upgradable version of Pausable and ReentrancyGuard. Not using the correct upgradable versions of the contracts can result in possible issues such as slot collisions.

Recommendations
The upgradable versions of the OZ contracts should be used.

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/utils/PausableUpgradeable.sol

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/utils/ReentrancyGuardUpgradeable.sol

Also in, initialize, call the upgradable contract's init functions

__ReentrancyGuard_init()
__Pausable_init()

Resolution
The recommended contracts were used.

Note
This issue was fixed before the review commit.

BQ-010: Loss of precision when calculating borrowerDelta

Severity: Low
Status: Acknowledged

In calculateBorrowRewardsForUser, there is loss of precision when calculating borrowerDelta

```
// Calculate change in the cumulative sum of the reward per cToken accrued
Double memory deltaIndex = Double({
    mantissa: sub_(_globalBorrowIndex, userBorrowIndex)
});

uint borrowerAmount = div_(
    _rTokenData.borrowBalanceStored,
    _marketBorrowIndex
);

// Calculate reward accrued: rTokenAmount * accruedPerRToken
uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
```

borrowerAmount does division first, before multiplying by deltaIndex.

Recommendations
To prevent loss of precision, it should be multiplication before division
uint borrowerDelta = div(mul_(_rTokenData.borrowBalanceStored, deltaIndex), _marketBorrowIndex);

Resolution
The team has acknowledged the issue. On a side note, the same issue is present in Compound v2.

https://github.com/compound-finance/compound-protocol/blob/a3214f67b73310d547e00fc578e8355911c9d376/contracts/Comptroller.sol#L1293-L1297

BQ-011: Deprecated markets cannot be liquidated

Severity: Medium
Status: Resolved

Unlike Compound v2, there is no functionality in liquidateBorrowAllowed to liquidate deprecated markets.

https://github.com/compound-finance/compound-protocol/blob/a3214f67b73310d547e00fc578e8355911c9d376/contracts/Comptroller.sol#L489-L508

This can result in it being not possible for accounts in deprecated markets to be liquidated.

Recommendations
Add the following functionality to determine if a QiToken is deprecated.

```
function isDeprecated(QiToken qiToken) public view returns (bool) {
    return
    markets[address(QiToken)].collateralFactorMantissa == 0 &&
    borrowGuardianPaused[address(QiToken)] == true &&
    QiToken.reserveFactorMantissa() == 1e18;
}
```

Also, in liquidateBorrowAllowed to check if a token is deprecated, and if so, ensure that .the borrowBalance is greater than or equal to the repayAmount.

```
    /* allow accounts to be liquidated if the market is deprecated */
    if (isDeprecated(QiToken(QiTokenBorrowed))) {
        require(borrowBalance >= repayAmount, "Can not repay more than the total borrow");
        return uint(Error.NO_ERROR);
    }
```

Resolution
The recommended change was implemented.

Note
This issue was fixed before the review commit.