Benqi Collateral Migrator Audit Report

Prepared by Cyfrin
Version 2.0

Lead Auditors

Immeas

April 6, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Benqi Collateral Migrator is a lightweight contract designed to help users of Benqi Markets seamlessly migrate the collateral of an existing position.
Ordinarily, a user would need to manually:

- Repay their loan,
- Withdraw their collateral,
- Swap it to the desired asset,
- Re-deposit it as collateral, and
- Reopen the loan.

The `CollateralMigrator` contract consolidates this entire process into a single transaction by leveraging flash loans from Trader Joe liquidity pairs and swaps via the 1inch DEX aggregator.

4.1 Actors and Roles

- **1. Actors:**
  - **Benqi**: Deploys the contract with trusted Trader Joe factories, and configures and maintains a list of supported `qiToken` markets.
  - **End users**: Use `CollateralMigrator` to migrate the collateral backing their positions.
- **2. Roles:**
  - **PAUSER_ROLE**: Can pause the `CollateralMigrator`, disabling `migrateCollateral` calls. While paused, the Benqi admin can sweep any stray tokens from the contract.
  - **DEFAULT_ADMIN_ROLE**: Can unpause the contract, manage supported market configurations, and sweep tokens when the contract is paused.

4.2 Key Components

- `CollateralMigrator`: The core contract responsible for executing collateral migrations. It enables users to swap the underlying collateral of their Benqi positions. The contract is not intended to retain any tokens between transactions.

4.3 Collateral Migration Flow

1. The end user obtains swap data from the 1inch API.
2. The end user calls `CollateralMigrator::migrateCollateral` with the required swap parameters.
3. `CollateralMigrator` takes a flash loan from a Trader Joe liquidity pair.
4. Using the flash loan, `CollateralMigrator` opens a position in the target market.
5. With the position now overcollateralized, the original source market position is closed.
6. The source tokens are swapped for target tokens.
7. The target tokens are used to repay the flash loan along with the associated fees.
8. Any remaining target tokens are deposited into the target market on behalf of the user.

5 Audit Scope

```
contracts/CollateralMigrator.sol
contracts/modules/SwapModule.sol
```

6 Executive Summary

Over the course of 4 days, the Cyfrin team conducted an audit on the Benqi Collateral Migrator smart contracts provided by Benqi. In this period, a total of 6 issues were found.

The audit uncovered four low-severity findings. The first relates to an edge case where leftover tokens in the contract may cause a revert. The second highlights that swaps may leave dust amounts of the source asset in the contract. The third notes that migrating positions will always result in a higher loan-to-value (LTV), increasing the user's risk. The fourth concerns an unbounded loop in market removal logic, which could lead to excessive gas consumption or even failed transactions as the number of supported markets grows.

The audit also identified some informational findings related to documentation and imports.

The NatSpec documentation throughout the contract was exemplary. Functionality was thoroughly covered by tests, and overall, the code was well written, clean, and easy to understand.

Summary

Project Name	Benqi Collateral Migrator
Repository	benqi-collateral-migrator
Commit	d91ce3dbf56d...
Fix Commit	afe450e39083...
Audit Timeline	Mar 24th - Mar 27th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	4
Informational	2
Gas Optimizations	0
Total Issues	6

Summary of Findings

[L-1] `CollateralMigrator` reverts when pre-existing funds are used for flash loan fees	Resolved
[L-2] Migrating collateral will leave dust amounts of source tokens behind	Resolved
[L-3] User LTV always worsens after migration	Resolved
[L-4] Unbounded loop in `CollateralMigrator::_removeMigrationConfigData` may hinder contract maintainability	Resolved
[I-1] Unused imports	Resolved
[I-2] NatSpec `@custom:reverts` inconsistencies	Resolved

7 Findings

7.1 Low Risk

7.1.1 `CollateralMigrator` reverts when pre-existing funds are used for flash loan fees

Description: The flow for migrating collateral between two markets is as follows:

1. The end user calls `CollateralMigrator::migrateCollateral` with the required swap parameters.
2. `CollateralMigrator` takes a flash loan from a Trader Joe liquidity pair.
3. Using the flash loan, `CollateralMigrator` opens a position in the target market.
4. With the position now overcollateralized, the original source market position can be closed.
5. The source tokens are swapped for target tokens.
6. The target tokens are used to repay the flash loan along with associated fees.
7. Any remaining target tokens are deposited into the target market on behalf of the user.

The `CollateralMigrator` contract is not intended to retain any funds after execution. However, if tokens remain in the contract (e.g., from a previous failed transaction), a user might attempt to use those to partially cover the flash loan fees.

This approach would cause a revert in `CollateralMigrator::LBFlashLoanCallback`:

```
uint256 contractBalanceBF = IERC20(trgMarketConfig.flashData.baseToken).balanceOf(address(this));

// Swap the underlying asset from the source market to the target market
_swap(
    ...
);

// Return the flash loaned amount to the liquidity book pair
uint256 flashAmountWithFee = flashAmount +
    (trgMarketConfig.flashData.isTokenX ? totalFees.decodeX() : totalFees.decodeY());

// IEIP20NonStandard(flashData.baseToken).transfer(address(flashData.liquidityBookPair),
↪   flashAmountWithFee);
IERC20(trgMarketConfig.flashData.baseToken).safeTransfer(
    address(trgMarketConfig.flashData.liquidityBookPair),
    flashAmountWithFee
);

// Return the remaining underlying asset to the market
uint256 contractBalanceAF = IERC20(trgMarketConfig.flashData.baseToken).balanceOf(address(this));
// @audit if existing funds were used to pay fees, this will underflow
uint256 remainingAmount = contractBalanceAF - contractBalanceAF - contractBalanceBF;
```

If any pre-existing funds are used to pay the flash loan fees, the line `contractBalanceAF - contractBalanceBF` will underflow, since `contractBalanceAF` will be lower than `contractBalanceBF`.

Impact: Using pre-existing tokens in the contract to cover flash loan fees will cause the transaction to revert due to an underflow.

Proof of Concept: Add the following test to `CollateralMigrator.test.ts`, under `context("Migration functions")`:

```
it("Migrate will not work when existing balance is used for fees", async function () {
    const { user, collateralMigrator, swapRouter, qiUSDT, qiDAI, USDT, DAI, usdtDaiLBPair } = await
    ↪   loadFixture(setupEnv);

    const fromMarket = qiUSDT.address;
    const toMarket = qiDAI.address;

    const fromAsset = USDT.address;
    const toAsset = DAI.address;
    const migrationAmount = parseEther("200");

    await usdtDaiLBPair.connect(user).setFeeBps(1000); // 10%

    await USDT.connect(user).mint(user.address, migrationAmount);
    await DAI.connect(user).mint(collateralMigrator.address, parseEther("20"));

    // we simulate that we need the amount of a flash loan for all 100% of the position (exchange rate
    ↪  1:1)
    const flashAmount = parseEther("200");

    const data = swapRouter.interface.encodeFunctionData("swap", [
        fromAsset,
        toAsset,
        parseEther("200"),
        parseEther("200")
    ]);

    const swapParams = {
        fromAsset,
        toAsset,
        fromAssetAmount: parseEther("200"),
        minToAssetAmount: parseEther("200"),
        data: data
    };

    // approve the collateral migrator to spend the collateral
    await USDT.connect(user).approve(qiUSDT.address, MaxUint256);
    await qiUSDT.connect(user).mint(migrationAmount);

    await qiUSDT.connect(user).approve(collateralMigrator.address, migrationAmount);

    await expect(
        collateralMigrator
            .connect(user)
            .migrateCollateral(fromMarket, toMarket, migrationAmount, flashAmount, swapParams)
        ).to.be.reverted; // revert on underflow
});
```

Note: this also requires the following changes to `contracts/mocks/MockLBPair.sol` to support fees:

```
diff --git a/contracts/mocks/MockLBPair.sol b/contracts/mocks/MockLBPair.sol
index 6074fa2..a1d28bd 100644
--- a/contracts/mocks/MockLBPair.sol
+++ b/contracts/mocks/MockLBPair.sol
@@ -16,6 +16,8 @@ contract MockLBPair {
     uint16 private _binStep;
     address private _factory;

+    uint128 feeBps;
+
     constructor(address tokenX, address tokenY, uint16 binStep, address factory) {
         _tokenX = tokenX;
         _tokenY = tokenY;
@@ -42,15 +44,21 @@ contract MockLBPair {
     function flashLoan(ILBFlashLoanCallback receiver, bytes32 amounts, bytes calldata data) external {
         (uint amountX, uint amountY) = amounts.decode();
```

```
+        bytes32 fees;
+
         if (amountX > 0 && amountY == 0) {
             IERC20(_tokenX).transfer(address(receiver), amountX);
+            uint128 feeX = uint128(amountX) * feeBps / 10000;
+            fees = feeX.encode(0);
         } else if (amountX == 0 && amountY > 0) {
             IERC20(_tokenY).transfer(address(receiver), amountY);
+            uint128 feeY = uint128(amountY) * feeBps / 10000;
+            fees = uint128(0).encode(feeY);
         } else {
             revert("MockLBPair: INVALID_AMOUNTS");
         }

-        receiver.LBFlashLoanCallback(msg.sender, _tokenX, _tokenY, amounts, 0, data);
+        receiver.LBFlashLoanCallback(msg.sender, _tokenX, _tokenY, amounts, fees, data);
     }

     function encodeAmounts(uint128 amountX, uint128 amountY) external pure returns (bytes32) {
@@ -86,4 +94,8 @@ contract MockLBPair {
     function getFactory() external view returns (address factory) {
         return _factory;
     }
+
+    function setFeeBps(uint128 _feeBps) external {
+        feeBps = _feeBps;
+    }
 }
```

Recommended Mitigation: Instead of calculating the difference between the pre- and post-swap balances, consider refunding the entire remaining balance directly:

```
- uint256 contractBalanceBF = IERC20(trgMarketConfig.flashData.baseToken).balanceOf(address(this));

  ...

  // Return the remaining underlying asset to the market
- uint256 contractBalanceAF = IERC20(trgMarketConfig.flashData.baseToken).balanceOf(address(this));
- uint256 remainingAmount = contractBalanceAF - contractBalanceAF - contractBalanceBF;
+ uint256 remainingAmount = IERC20(trgMarketConfig.flashData.baseToken).balanceOf(address(this));

  if (remainingAmount > 0) {
      _supplyToMarket(user, targetMarket, remainingAmount);
  }
```

Benqi: Fixed in commit 3828ee7

Cyfrin: Verified. Ingoing balance is now tracked. If the swap doesn't leave enough balance to cover the flashloan and fees execution will revert.

7.1.2 Migrating collateral will leave dust amounts of source tokens behind

Description: When migrating collateral, the user provides two separate amounts related to how much of the source market tokens should be migrated: `migrationAmount` and `swapParams.fromAssetAmount`.

- `migrationAmount` is the number of `qiTokens` (Compound V2 `CTokens`) to redeem. This is handled in `CollateralMigrator::LBFlashLoanCallback#L474-L482`:

```
// Redeem the migration amount from the source market
if (migrationAmount == type(uint256).max) {
    // Get the QiToken balance of the source market
    migrationAmount = IMinimalQiToken(sourceMarket).balanceOf(user);
}
// Transfer the QiTokens to the contract
IERC20(sourceMarket).safeTransferFrom(user, address(this), migrationAmount);
// Redeem the QiTokens from the source market
if (IMinimalQiToken(sourceMarket).redeem(migrationAmount) != 0) revert RedeemFailed();
```

- `swapParams.fromAssetAmount` is the amount of underlying source tokens to be swapped for target market tokens, as seen in `CollateralMigrator::LBFlashLoanCallback#L490-L497`:

```
// Swap the underlying asset from the source market to the target market
_swap(
    _swapParams.fromAsset,
    _swapParams.toAsset,
    _swapParams.fromAssetAmount,
    _swapParams.minToAssetAmount,
    _swapParams.data
);
```

The issue lies in the mismatch between these values: `migrationAmount` represents a quantity of interest-bearing `qiTokens`, not the actual amount of underlying tokens they redeem for. Over time, `qiTokens` accumulate interest and represent a growing amount of the underlying asset.

Because the user must provide `fromAssetAmount` at the time of transaction submission, before the redemption happens, they must guess how much underlying they will receive. If they guess too high, the swap will revert. To avoid this, users are forced to guess conservatively and leave some underlying tokens unutilized in the `CollateralMigrator` contract.

Impact: Due to the interest-bearing nature of `qiTokens`, users cannot accurately determine how much underlying they will receive. As a result, dust amounts of leftover source tokens remain in the `CollateralMigrator` contract after the swap.

While these tokens are not permanently lost (they can be recovered via an admin-only `sweep` function), users are unable to fully migrate their position.

Recommended Mitigation: Consider removing the `fromAssetAmount` parameter entirely, as there's no known use case for performing a partial swap. Instead, swap the entire balance of the source asset:

```
  // Swap the underlying asset from the source market to the target market
  _swap(
      _swapParams.fromAsset,
      _swapParams.toAsset,
-     _swapParams.fromAssetAmount,
+     IERC20(_swapParams.fromAsset).balanceOf(address(this))
      _swapParams.minToAssetAmount,
      _swapParams.data
  );
```

Benqi: Fixed in commit e9525d1

Cyfrin: Verified. Contract balance is now used as input amount to the swap.

7.1.3 User LTV always worsens after migration

Description: The flow for migrating collateral between two markets is as follows:

1. The end user calls `CollateralMigrator::migrateCollateral` with the required swap parameters.
2. `CollateralMigrator` takes a flash loan from a Trader Joe liquidity pair.
3. Using the flash loan, `CollateralMigrator` opens a position in the target market.
4. With the position now overcollateralized, the original source market position can be closed.

5. The source tokens are swapped for target tokens.
6. The target tokens are used to repay the flash loan along with the associated fees.
7. Any remaining target tokens are deposited into the target market on behalf of the user.

The issue lies in how flash loan fees affect the resulting position.

Consider the following example:

You have a position where you've borrowed 50 ETH against 110 USDC, giving you an LTV of ~45%. You want to migrate the collateral from USDC to DAI. For simplicity, assume:

- A 10% flash loan fee, and
- A 1:1 USDC/DAI exchange rate.

The largest flash loan you can take is 100 DAI, since the additional 10 DAI needed to cover the fee must come from the swap proceeds.

You then swap your 110 USDC for 110 DAI and repay the 100 DAI loan plus 10 DAI in fees. This leaves you with 0 DAI left over. Your new collateral position is now just 100 DAI backing 50 ETH, raising your LTV to 50%.

Because the flash loan must be repaid with fees, and those fees must come from the user's existing collateral, every migration necessarily increases the user's LTV, reducing their margin of safety.

Impact: Collateral migration always results in a higher LTV post-migration, increasing the user's liquidation risk.

Recommended Mitigation: Consider adding a parameter allowing users to optionally top up the transaction with additional target tokens to cover the flash loan fees. This would preserve their original LTV.

Benqi: Fixed in commit `afe450e`

Cyfrin: Verified. A parameter `extraFunds` was added to `migrateCollateral` that allows the user to top up, keeping their LTV the same.

7.1.4 Unbounded loop in `CollateralMigrator::_removeMigrationConfigData` may hinder contract maintainability

Description: `CollateralMigrator::_removeMigrationConfigData` contains an unbounded loop when removing migration configurations:

```
function _removeMigrationConfigData(address market) private {
    uint32 length = uint32(_markets.length);
    bool found = false;

    for (uint256 i = 0; i < length; i++) {
        if (_markets[i] == market) {
            found = true;
            // remove the market from the list
            _markets[i] = _markets[length - 1];
            _markets.pop();

            // remove the market data and flash data
            delete _migrationConfigData[market];

            emit MigrationConfigDataRemoved(market);
            break;
        }
    }
    // revert if the market is not found
    if (!found) revert MarketIsNotSupported(market);
}
```

This implementation uses a linear search to find the market to remove. If a large number of markets are added, the loop may consume excessive gas, especially for markets positioned near the end of the array. In the worst-case scenario, the transaction could exceed the block gas limit, making it impossible to remove a market.

Additionally, removing markets located near the end of the array incurs higher gas costs, which may become non-trivial as the list grows.

Impact: In the worst case, removing a market could fail due to running out of gas. Even when successful, removals, especially of markets toward the end of the list, can become unnecessarily expensive.

Recommended Mitigation: Consider changing the function to accept the index of the market to remove rather than performing a linear search. This would eliminate the unbounded loop and reduce gas usage:

```
function _removeMigrationConfigData(uint256 index) external onlyRole(DEFAULT_ADMIN_ROLE) {
    address market = _markets[index];

    _markets[index] = _markets[_markets.length - 1];
    _markets.pop();

    delete _migrationConfigData[market];

    emit MigrationConfigDataRemoved(market);
}
```

Benqi: Fixed in commit `64aba7b`

Cyfrin: Verified. A mapping `_marketIndex` was added to track markets to indexes. This is used to call `_removeMigrationConfigData` and the loop is removed.

7.2 Informational

7.2.1 Unused imports

Description: There are two unused imports:

- `CommonErrors`, `CollateralMigrator.sol#17`:

```
import {CommonErrors} from "./errors/CommonErrors.sol";
```

- `ISwapper`, `SwapModule.sol#L6`:

```
import {ISwapper} from "../interfaces/1Inch-v6/ISwapper.sol";
```

Consider removing these.

Benqi: Fixed in commit `34d2cca`

Cyfrin: Verified.

7.2.2 NatSpec `@custom:reverts` inconsistencies

Description: Below are some inconsistencies in how the NatSpec `@custom:reverts` are formatted:

- `CollateralMigrator::LBFlashLoanCallback`:

```
* @custom:reverts IncorrectToAsset Thrown if the `toAsset` in `swapParams` does not match the
↪  target market's base token.
* @custom:reverts RedeemFailed Thrown if the QiToken redemption process fails.
* - {MintFailed}: Thrown if the minting of tokens to the target market fails.
* @custom:reverts MintFailed Thrown if the minting of tokens to the target market fails.
* @custom:reverts InvalidCallbackHash Thrown if the callback data hash does not match the stored
↪  hash.
```

  `MintFailed` is mentioned twice, once with `@custom:reverts` once without.

- `SwapModule::constructor`

```
/**
 * @notice Initializes the SwapModule with the address of the 1Inch router.
 * @param _swapRouter The address of the 1Inch router contract.
 * @dev Reverts with {InvalidZeroAddress} if `_swapRouter` is the zero address.
 */
constructor(address _swapRouter) {
    if (_swapRouter == address(0)) {
        revert InvalidZeroAddress();
    }
    SWAP_ROUTER = _swapRouter;
}
```

  The custom error `InvalidZeroAddress` is missing the `@custom:reverts` in the documentation.

- `SwapModule::_swap`

```
* - {SwapFailed} if the call to the 1Inch router fails.
* - {IncorrectSwapAmount} if the resulting balance does not increase.
* - {OutputLessThanMinAmount} if the received amount is less than the specified minimum.
```

  The `@custom:reverts` is not used as it is in `CollateralMigrator`

Benqi: Fixed in commit `053f28a`

Cyfrin: Verified.