BEOSIN
Blockchain Security

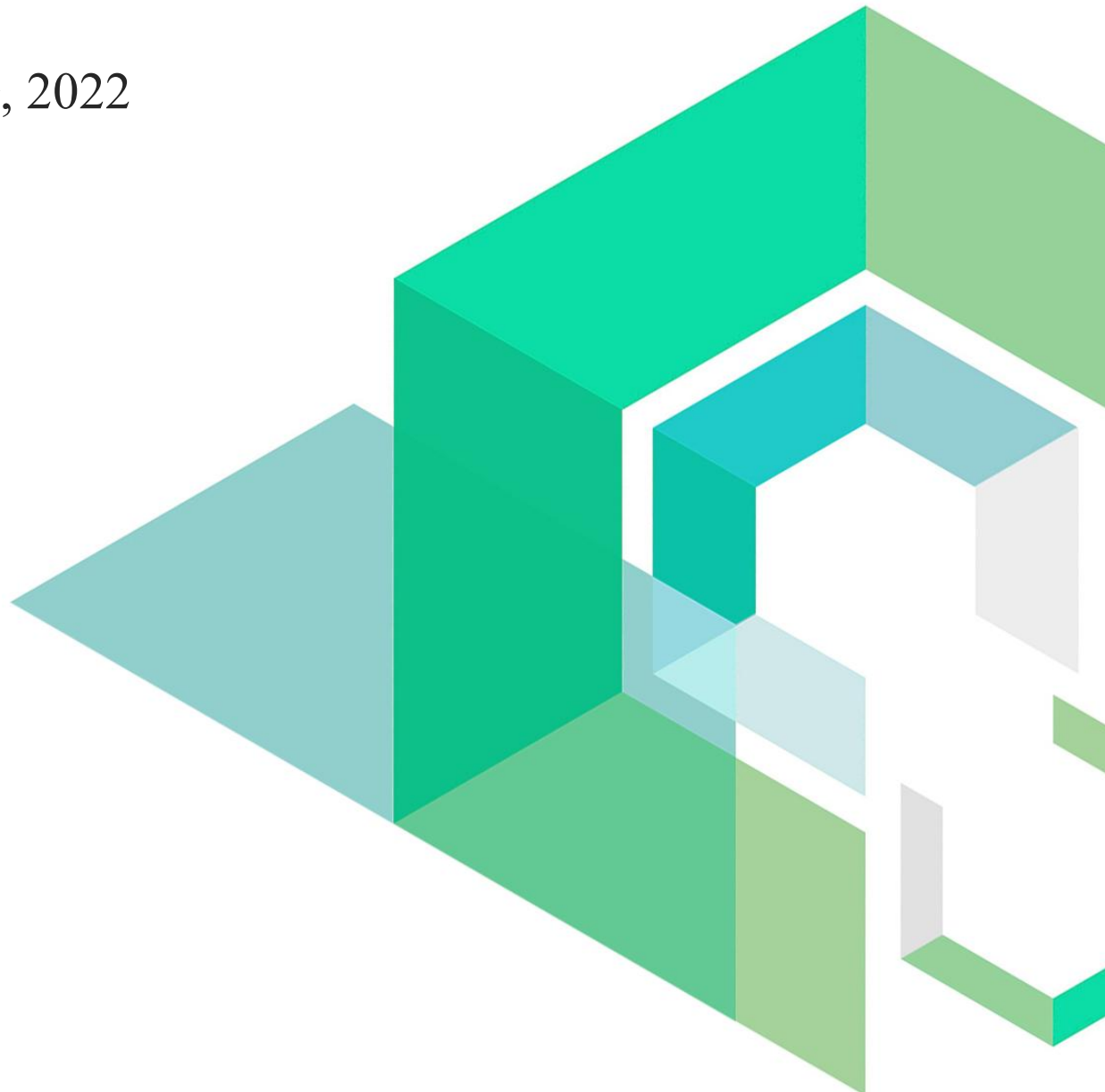
LeverFi

Smart Contract Security Audit

V1.0

No. 202209141908

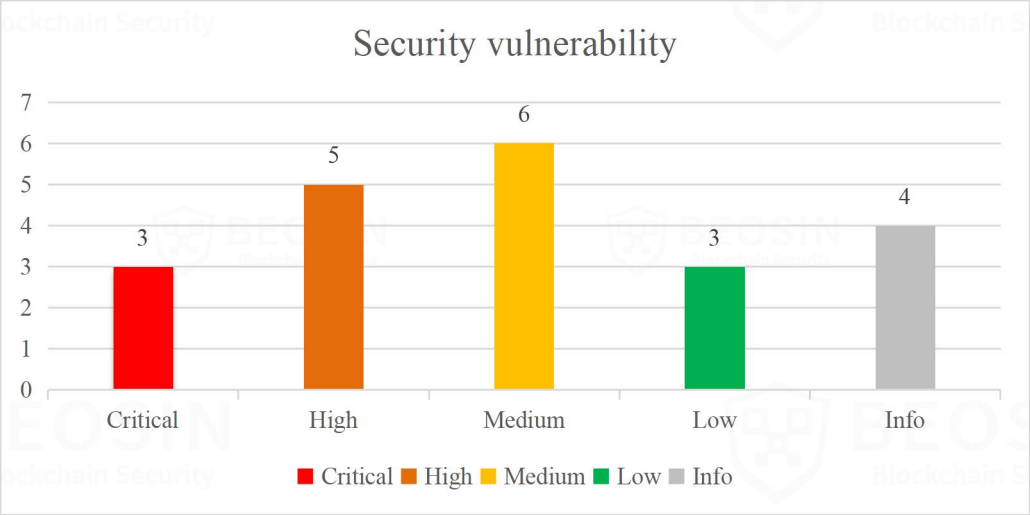
Sept 14th, 2022



Contents

Summary of audit results

After auditing, 3 Critical-risk, 5 High-risk, 6 Medium-risk, 3 Low-risk and 4 Info-risk items were identified in the LeverFi project. Specific audit details will be presented in the **Findings section**. Users should pay attention to the following aspects when interacting with this project:



*Notes:

- **Risk Description:**

1. This project only audits the internal code of the project, and does not audit the interactive contract. Audited contracts include CollateralLogic.sol, GeneralLogic.sol, LiquidationLogic.sol, ReserveLogic.sol, ShareBaseAccounting.sol, TradeLogic.sol, ValidationLogic.sol, InterestUtils.sol, MathUtils.sol, UserConfiguration.sol, ChainLinkPriceOracleGetter.sol, CurveLPOracleGetter. sol, MockPriceOracle.sol, TriCryptoOracle.sol, AaveReinvestmentLogic.sol, ConvexReinvestmentLogic.sol, ReinvestmentProxy.sol, ZeroexSwapAdapter.sol, DataTypes.sol, UnwrapLp.sol and Ledger.sol

2. There is test code in the project, please do not use the test code, such as MockPriceOracle.sol, etc.

3. This audit report is only for the current code, but the business logic contract of the current project can be upgraded, and the code after the upgrade cannot be determined. After the upgrade, the risk of loss of funds and data may be introduced. When users interact with this project, they need to pay attention to whether the upgraded logic contract is consistent with the audit code.

- **Project Description:**

1. Business overview

The LeverFi project is a revenue aggregation project that allows liquidity providers to provide funds to earn interest. Liquidity providers deposit funds into lending pools, and lenders earn interest from borrowing leveraged traders. Idle assets in lending pools that are not used (unused liquidity) by traders will be deployed to other DeFi protocols for yield. When a trader deposits BTC, ETH, Curve-LP, Uni-LP and other collaterals in the collateral pool, the contract will deposit the collateral into the DeFi protocol to obtain benefits, while allowing traders to perform leveraged transactions. Leveraged trades are only entered, stored and settled within the LeverFi platform. If a trade loses close to the value of the deposited collateral, the liquidator system will make a margin call to the trader.

1 Overview

1.1 Project Overview

Project Name	LeverFi
Platform	Ethereum
Github	https://github.com/LeverFi/main-contracts
Commit	ed904fe9caabab160fdaed965094613236ba9308(original) 594a866a7b26d337d52c5e286f4f2b9d78dbb85b(fixed)

1.2 Audit Overview

Audit work duration: August 4, 2022 – September 14, 2022

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Security Team

2 Findings

Index	Risk description	Severity level	Status
LeverFi-1	The *initialize* function lacks permission check	Critical	Fixed
LeverFi-2	The *swapPosition* function lacks permission check	Critical	Fixed
LeverFi-3	The *swap* function is improperly designed	Critical	Fixed
LeverFi-4	DoS attack	High	Fixed
LeverFi-5	The *_calculateRewards* function design flaw	High	Fixed
LeverFi-6	The *_depositReserve* function is improperly designed	High	Fixed
LeverFi-7	The *configureAsset* function is improperly designed	High	Fixed
LeverFi-8	Clearing mechanism error	High	Fixed
LeverFi-9	The *getNormalizedDebt* function is improperly designed	Medium	Fixed
LeverFi-10	Implementation flaws in the *reinvestReserveSupply* and *reinvestCollateralSupply* functions	Medium	Fixed
LeverFi-11	Unsafe call	Medium	Fixed
LeverFi-12	The *configureReserve* function design flaw	Medium	Fixed
LeverFi-13	Centralization risk	Medium	Fixed
LeverFi-14	The *_repayShort* function is improperly designed	Medium	Fixed
LeverFi-15	Improperly designed *_depositCollateral* function	Low	Fixed
LeverFi-16	The *validateTrade* function check error	Low	Fixed
LeverFi-17	The *configureCollateral* function cannot modify reinvestment	Low	Acknowledged
LeverFi-18	lack of judgment on value	Info	Fixed
LeverFi-19	The *getUserLiquidity* function design flaw	Info	Fixed
LeverFi-20	Lack of judgment on whether to add assets	Info	Fixed
LeverFi-21	Admin permission not initialized	Info	Fixed

Risk Details Description:

- LeverFi-16 is unfixed and will result in the inability to set up a new reinvestment.

Finding Details:

[LeverFi-1] The *initialize* function lacks permission check

Severity Level	**Critical**
Type	Business Security
Lines	AaveReinvestmentLogic.sol,ConvexReinvestmentLogic.sol#L47,79
Description	In the initialize of the ConvexReinvestmentLogic and AaveReinvestmentLogic contracts, there is no permission to check and use the Initializable contract, which will cause anyone to call this function to modify it, thereby withdrawing the funds in the contract.

```
37
38    function initialize(
39        address asset_,
40        address receipt_,
41        address platform_,
42        address[] memory,
43        address treasury_,
44        address ledger_,
45        uint256,
46        bytes memory
47    ) public {
48        asset = asset_;
49        addressStorage[RECEIPT] = receipt_;
50        addressStorage[PLATFORM] = platform_;
51        treasury = treasury_;
52        ledger = ledger_;
53
54        // Fees does not applied to this type of reinvestment (auto-accruing reward).
55        feeMantissa = 0;
56    }
```

Figure 1 Source code of AaveReinvestmentLogic contract(Unfixed)

```
69     **/
70    function initialize(
71        address asset_,
72        address receipt_,
73        address platform_,
74        address[] memory rewards_,
75        address treasury_,
76        address ledger_,
77        uint256 feeMantissa_,
78        bytes memory data
79    ) public {
80        asset = asset_;
81        addressStorage[RECEIPT] = receipt_;
82        addressStorage[PLATFORM] = platform_;
83        treasury = treasury_;
84        ledger = ledger_;
85        feeMantissa = feeMantissa_;
86
87        (uint256 poolId_, address rewardPool_) = abi.decode(data, (uint256, address));
88        uintStorage[POOL_ID] = poolId_;
89        addressStorage[REWARD_POOL] = rewardPool_;
90
91        uintStorage[REWARD_LENGTH] = rewards_.length;
92
93        for (uint256 i = 0; i < rewards_.length; i++) {
94            setRewards(i, RewardConfig(i, rewards_[i], 0, 0));
95        }
96    }
```

Figure 2 Source code of ConvexReinvestmentLogic contract(Unfixed)

Recommendations	It is recommended to use the Initializable contract.
Status	Fixed.

```
39  function initialize(
40      address asset_,
41      address receipt_,
42      address platform_,
43      address[] memory,
44      address treasury_,
45      address ledger_,
46      uint256,
47      bytes memory
48  ) external initializer onlyOwner {
49      asset = asset_;
50      addressStorage[RECEIPT] = receipt_;
51      addressStorage[PLATFORM] = platform_;
52      treasury = treasury_;
53      ledger = ledger_;
54
55      // Fees does not applied to this type of reinvestment (auto-accruing reward).
56      feeMantissa = 0;
57  }
```

Figure 3 Source code of AaveReinvestmentLogic contract(Fixed)

```
70   **/
71  function initialize(
72      address asset_,
73      address receipt_,
74      address platform_,
75      address[] memory rewards_,
76      address treasury_,
77      address ledger_,
78      uint256 feeMantissa_,
79      bytes memory data
80  ) external initializer onlyOwner {
81      asset = asset_;
82      addressStorage[RECEIPT] = receipt_;
83      addressStorage[PLATFORM] = platform_;
84      treasury = treasury_;
85      ledger = ledger_;
86      feeMantissa = feeMantissa_;
87
88      (uint256 poolId_, address rewardPool_) = abi.decode(data, (uint256, address));
89      uintStorage[POOL_ID] = poolId_;
90      addressStorage[REWARD_POOL] = rewardPool_;
91
92      uintStorage[REWARD_LENGTH] = rewards_.length;
93
94      for (uint256 i = 0; i < rewards_.length; i++) {
95          setRewards(i, RewardConfig(i, rewards_[i], 0, 0));
96      }
97  }
```

Figure 4 Source code of ConvexReinvestmentLogic contract(Fixed)

[LeverFi-2] The *swapPosition* function lacks permission check

Severity Level	**Critical**
Type	General Vulnerability
Lines	Ledger.sol#L1147
Description	The *swapPosition* function lacks permission verification. Anyone can execute this function and use the contract's tokens to swap, which will result in a loss of contract funds.

```
1146
1147    function swapPosition(address assetIn, address assetOut, uint256 amount, bytes memory data) external {
1148        TradeLogic.executeTrade(
1149            _reserveConfig,
1150            assetConfig,
1151            _userPosition[LIQUIDATION_WALLET],
1152            userConfig[LIQUIDATION_WALLET],
1153            DataTypes.ExecuteSwapParams(
1154                LIQUIDATION_WALLET,
1155                treasury,
1156                assetIn,
1157                assetOut,
1158                amount,
1159                tradeFeeMantissa,
1160                0,
1161                0,
1162                0,
1163                data,
1164                false,
1165                false
1166            )
1167        );
1168    }
1169
```

Figure 5 Source code of *swapPosition* function(Unfixed)

```
79      **/
80      function executeTrade(
81          mapping(address => DataTypes.ReserveConfig) storage reserveConfig,
82          mapping(address => DataTypes.AssetConfig) storage assetConfig,
83          mapping(address => DataTypes.UserPosition) storage userPosition,
84          DataTypes.UserConfiguration storage userConfig,
85          DataTypes.ExecuteSwapParams memory params
86      ) external {
87          ExecuteTradeVars memory vars;
88          vars.assetConfigCache[0] = assetConfig[params.shortAsset];
89          vars.assetConfigCache[1] = assetConfig[params.longAsset];
90          executeShorting(
91              reserveConfig[params.shortAsset],
92              vars.assetConfigCache[0],
93              userPosition[params.shortAsset],
94              userConfig,
95              params.amount,
96              true
97          );
98          if (params.applyFees) {
99              vars.feeAmount = params.amount.wadMul(params.tradeFeeMantissa);
100             params.amount = params.amount - vars.feeAmount;
101             IERC20Upgradeable(params.shortAsset).safeTransfer(params.treasury, vars.feeAmount);
102         }
103         if (
104             IERC20Upgradeable(params.shortAsset).allowance(address(this), address(vars.assetConfigCache[0].swapAdapter)) < params.amount
105         ) {
106             IERC20Upgradeable(params.shortAsset).safeApprove(address(vars.assetConfigCache[0].swapAdapter), type(uint256).max);
107         }
108         vars.receivedAmount = vars.assetConfigCache[0].swapAdapter.swap(params.shortAsset, params.longAsset, params.amount, params.data);
109         uint256 increaseShortAmount = GeneralLogic.getAssetUsd(
110             vars.assetConfigCache[0],
```

Figure 6 Source code of *executeTrade* function

Recommendations	It is recommended to add the permission check of the function caller.
Status	Fixed.

```
1175
1176    function swapPosition(address assetIn, address assetOut, uint256 amount, bytes memory data) external onlyLiquidateExecutor {
1177        TradeLogic.executeTrade(
1178            _reserveConfig,
1179            assetConfig,
1180            _userPosition[LIQUIDATION_WALLET],
1181            userConfig[LIQUIDATION_WALLET],
1182            DataTypes.ExecuteSwapParams(
1183                LIQUIDATION_WALLET,
1184                treasury,
1185                assetIn,
1186                assetOut,
1187                amount,
1188                tradeFeeMantissa,
1189                0,
1190                0,
1191                0,
1192                data,
1193                false,
1194                false
1195            )
1196        );
1197    }
1198
```

Figure 7 Source code of *swapPosition* function(Fixed)

[LeverFi-3] The *swap* function is improperly designed

Severity Level	**Critical**
Type	Business Security
Lines	Ledger.sol,ZeroexSwapAdapter.sol#L20-54
Description	When trading, the user can pass in the specified asset for swap, instead of swapping according to the actual asset, so that the user can pass in the fake tokens in exchange for the funds of the protocol.

```
445     function trade(address shortAsset, address longAsset, uint256 amount, bytes memory data) external {
446         _executeTrade(msg.sender, shortAsset, longAsset, amount, data);
447     }
448
449     /**
```

Figure 8 Source code of *trade* function

```
79      **/
80      function executeTrade(
81          mapping(address => DataTypes.ReserveConfig) storage reserveConfig,
82          mapping(address => DataTypes.AssetConfig) storage assetConfig,
83          mapping(address => DataTypes.UserPosition) storage userPosition,
84          DataTypes.UserConfiguration storage userConfig,
85          DataTypes.ExecuteSwapParams memory params
86      ) external {
87          ExecuteTradeVars memory vars;
88          vars.assetConfigCache[0] = assetConfig[params.shortAsset];
89          vars.assetConfigCache[1] = assetConfig[params.longAsset];
90          executeShorting(
91              reserveConfig[params.shortAsset],
92              vars.assetConfigCache[0],
93              userPosition[params.shortAsset],
94              userConfig,
95              params.amount,
96              true
97          );
98          if (params.applyFees) {
99              vars.feeAmount = params.amount.wadMul(params.tradeFeeMantissa);
100             params.amount = params.amount - vars.feeAmount;
101             IERC20Upgradeable(params.shortAsset).safeTransfer(params.treasury, vars.feeAmount);
102         }
103         if (
104             IERC20Upgradeable(params.shortAsset).allowance(address(this), address(vars.assetConfigCache[0].swapAdapter)) < params.amount
105         ) {
106             IERC20Upgradeable(params.shortAsset).safeApprove(address(vars.assetConfigCache[0].swapAdapter), type(uint256).max);
107         }
108         vars.receivedAmount = vars.assetConfigCache[0].swapAdapter.swap(params.shortAsset, params.longAsset, params.amount, params.data);
109         uint256 increaseShortAmount = GeneralLogic.getAssetUsd(
110             vars.assetConfigCache[0],
```

Figure 9 Source code of *executeTrade* function

```
19      **/
20      function swap(address, address, uint256 amountToSwap, bytes memory swapBytesData) external override returns (uint256) {
21
22          (address approveRouter, address sellingAsset, address buyingAsset, bytes memory swapBytes) = abi.decode(swapBytesData, (address,address,address,bytes));
23
24          uint256 buyingAssetBalancePrior = IERC20Upgradeable(buyingAsset).balanceOf(address(this));
25
26          IERC20Upgradeable(sellingAsset).safeTransferFrom(msg.sender, address(this), amountToSwap);
27
28
29
30          // approve to appropriate router (Note: router is the part of swapBytes data. Do not take from the swap func arg. The function arg is given because for other swaps like 1inch
31
32          IERC20Upgradeable(sellingAsset).safeIncreaseAllowance(approveRouter, amountToSwap);
33
34
35          (bool success, bytes memory data) = approveRouter.call(swapBytes);
36
37          if(!success){
38              assembly {
39                  let returndata_size := mload(data)
40                  revert(add(32, data), returndata_size)
41              }
42          }
43
44          // check balance after swap
45          uint256 buyingAssetBalance = IERC20Upgradeable(buyingAsset).balanceOf(address(this));
46          uint256 receivedAmount = buyingAssetBalance - buyingAssetBalancePrior;
47
48
49
50          IERC20Upgradeable(buyingAsset).safeTransfer(msg.sender, receivedAmount);
51
52          return receivedAmount;
53      }
54  }
```

Figure 10 Source code of *swap* function(Unfixed)

Recommendations	It is recommended that the code use the incoming params.shortAsset and params.longAsset instead of relying on the selling asset and buy asset in the data entered by the user.
Status	Fixed.

```
function swap(address sellingAsset, address buyingAsset, uint256 amountToSwap, bytes memory swapBytesData)  external override returns (uint256) {

    (address approveRouter,,, bytes memory swapBytes) = abi.decode(swapBytesData, (address,address,address,bytes));

    require(routersList[approveRouter], "disabled router address");

    uint256 buyingAssetBalancePrior = IERC20Upgradeable(buyingAsset).balanceOf(address(this));

    IERC20Upgradeable(sellingAsset).safeTransferFrom(msg.sender, address(this), amountToSwap);



    // approve to appropriate router (Note: router is the part of swapBytes data. Do not take from the swap func arg. The function arg is given because for other swaps like 1inch

    IERC20Upgradeable(sellingAsset).safeIncreaseAllowance(approveRouter, amountToSwap);


    (bool success, bytes memory data) = approveRouter.call(swapBytes);

    if(!success){
        assembly {
            let returndata_size := mload(data)
            revert(add(32, data), returndata_size)
        }
    }

    // check balance after swap
    uint256 buyingAssetBalance = IERC20Upgradeable(buyingAsset).balanceOf(address(this));
    uint256 receivedAmount = buyingAssetBalance - buyingAssetBalancePrior;



    IERC20Upgradeable(buyingAsset).safeTransfer(msg.sender, receivedAmount);

    return receivedAmount;
  }
}
```

Figure 11 Source code of *swap* function(Fixed)

[LeverFi-4] DoS attack

Severity Level	**High**
Type	General Vulnerability
Lines	AaveReinvestmentLogic.sol,ConvexReinvestmentLogic.sol#L65,70,110,116-120
Description	There is a DoS vulnerability in the *invest* function of the AaveReinvestmentLogic and ConvexReinvestmentLogic contracts. If someone sends a token to the contract in advance, the user will fail to call the contract. Because the judgment here is that the amout input by the user must be equal to the contractBalance in the contract, if someone maliciously transfers money to the contract, the judgment will never be satisfied. Similarly, "IERC20Upgradeable(asset).balanceOf(address(this)) == 0" also has this problem.

```
57
58  function invest(uint256 amount) external override onlyLedger {
59
60      IERC20Upgradeable(asset).safeTransferFrom(msg.sender, address(this), amount);
61
62      uint256 contractBalance = IERC20Upgradeable(asset).balanceOf(address(this));
63
64      require(contractBalance > 0, "reinvestment adapter do not have underlying assets to invest");
65      require(contractBalance == amount, "investing amount is different");
66
67      IERC20Upgradeable(asset).safeApprove(platform(), amount);
68      IAaveLendingPoolV2(platform()).deposit(asset, amount, address(this), 0);
69
70      require(IERC20Upgradeable(asset).balanceOf(address(this)) == 0, "reinvestment adapter still has underlying assets to invest");
71  }
```

Figure 12 Source code of *invest* function (AaveReinvestmentLogic.sol)(Unfixed)

```
102     function invest(
103         uint256 amount
104     ) external override onlyLedger {
105         IERC20Upgradeable(asset).safeTransferFrom(msg.sender, address(this), amount);
106
107         uint256 contractBalance = IERC20Upgradeable(asset).balanceOf(address(this));
108
109         require(contractBalance > 0, "reinvestment adapter do not have underlying assets to invest");
110         require(contractBalance == amount, "investing amount is different");
111
112         IERC20Upgradeable(asset).safeApprove(platform(), amount);
113
114         IConvexBooster(platform()).deposit(poolId(), amount, true);
115
116         require(
117             IERC20Upgradeable(asset).balanceOf(address(this)) == 0,
118             "reinvestment adapter still has underlying assets to invest"
119         );
120     }
121
```

Figure 13 Source code of *invest* function (ConvexReinvestmentLogic.sol)(Unfixed)

Recommendations	It is recommended to delete the code.
Status	Fixed.

```
58
59      function invest(uint256 amount) external override onlyLedger {
60
61          IERC20Upgradeable(asset).safeTransferFrom(msg.sender, address(this), amount);
62
63          require(IERC20Upgradeable(asset).balanceOf(address(this)) >= amount, "not enough underlying amount to invest");
64
65          IERC20Upgradeable(asset).safeApprove(platform(), amount);
66          IAaveLendingPoolV2(platform()).deposit(asset, amount, address(this), 0);
67      }
68
```

Figure 14 Source code of *invest* function (AaveReinvestmentLogic.sol)(Fixed)

```
102     **/
103     function invest(
104         uint256 amount
105     ) external override onlyLedger {
106         IERC20Upgradeable(asset).safeTransferFrom(msg.sender, address(this), amount);
107
108         require(IERC20Upgradeable(asset).balanceOf(address(this)) >= amount, "not enough underlying amount to invest");
109
110         IERC20Upgradeable(asset).safeApprove(platform(), amount);
111         IConvexBooster(platform()).deposit(poolId(), amount, true);
112     }
113
```

Figure 15 Source code of *invest* function (ConvexReinvestmentLogic.sol)(Fixed)

[LeverFi-5] The *_calculateRewards* function design flaw

Severity Level	**High**
Type	Business Security
Lines	ConvexReinvestmentLogic.sol#L189-226
Description	Design flaw in the *_calculateRewards* function of the ConvexReinvestmentLogic contract. Because the user does not update the value of acquiredBalance after receiving the reward, it will result in an error in the calculation of the reward.

```
188
189     function _calculateRewards(
190         address user,
191         uint256 currBalance,
192         RewardConfig memory reward,
193         UserReward memory userReward,
194         bool isClaim
195     ) internal returns (RewardConfig memory, UserReward memory) {
196
197         address crvRewards = IConvexBooster(platform()).poolInfo(poolId()).crvRewards;
198         IConvexRewards(crvRewards).getReward();
199
200         uint256 accruedBalance = IERC20Upgradeable(reward.asset).balanceOf(address(this));
201
202         if (totalSupply() > 0 && accruedBalance > reward.remaining) {
203             reward.integral += (accruedBalance - reward.remaining) * 1e20 / totalSupply();
204         }
205
206         if (isClaim || reward.integral > userReward.integral) {
207             uint256 receivable = (reward.integral - userReward.integral) * currBalance / 1e20;
208
209             if (isClaim) {
210                 receivable += userReward.claimable;
211
212
213                 IERC20Upgradeable(reward.asset).safeTransfer(user, receivable);
214                 userReward.claimable = 0;
215
216             } else {
217                 userReward.claimable += receivable;
218             }
219             userReward.integral = reward.integral;
220         }
221
222         if (accruedBalance != reward.remaining) {
223             reward.remaining = accruedBalance;
224         }
225
226         return (reward, userReward);
227     }
228
```

Figure 16 Source code of *_calculateRewards* function(Unfixed)

Recommendations	It is recommended to update the acquiredBalance after claiming the reward.
Status	Fixed.

```
181     function _calculateRewards(
182         address user,
183         uint256 currBalance,
184         RewardConfig memory reward,
185         UserReward memory userReward,
186         bool isClaim
187     ) internal returns (RewardConfig memory, UserReward memory) {
188
189         address crvRewards = IConvexBooster(platform()).poolInfo(poolId()).crvRewards;
190         IConvexRewards(crvRewards).getReward();
191
192         uint256 accruedBalance = IERC20Upgradeable(reward.asset).balanceOf(address(this));
193
194         if (totalSupply() > 0 && accruedBalance > reward.remaining) {
195             reward.integral += (accruedBalance - reward.remaining) * 1e20 / totalSupply();
196         }
197
198         if (isClaim || reward.integral > userReward.integral) {
199             uint256 receivable = (reward.integral - userReward.integral) * currBalance / 1e20;
200
201             if (isClaim) {
202                 receivable += userReward.claimable;
203
204
205                 IERC20Upgradeable(reward.asset).safeTransfer(user, receivable);
206                 userReward.claimable = 0;
207                 accruedBalance -= receivable;
208             } else {
209                 userReward.claimable += receivable;
210             }
211             userReward.integral = reward.integral;
212         }
213
214         if (accruedBalance != reward.remaining) {
215             reward.remaining = accruedBalance;
216         }
217
218         return (reward, userReward);
219     }
```

Figure 17 Source code of *_calculateRewards* function (ConvexReinvestmentLogic.sol)(Fixed)

[LeverFi-6] The *_depositReserve* function is improperly designed

Severity Level	**High**
Type	Business Security
Lines	Ledger.sol#L812-825
Description	First of all, in the *getReserveSupply* function, if the function uses true, then poolAmount will not use the UtilizedSupply part. Go back to the *_depositReserve* function of the Ledger contract and calculate the part of shareAmountRay that does not use UtilizedSupply, then the calculated share will be wrong. In the *_withdrawReserve* function, the poolAmount is added with UtilizedSupply. The calculated shareAmountRay is smaller than expected, which will lead to arbitrage space. Similarly, the part of UtilizedSupply should also be used when calculating currUserReserveBalance.

```
798     /********************* INTERNAL ****************************/
799     function _depositReserve(address user, address asset, uint256 amount) internal {
800         DataTypes.AssetConfig memory assetConfigCache = assetConfig[asset];
801         DataTypes.ReserveConfig storage reserve = _reserveConfig[asset];
802         uint256 unit = assetConfigCache.decimals;
803
804         ValidationLogic.validateDepositReserve(reserve, assetConfigCache, amount);
805
806         reserve.updateIndex();
807
808         uint256 currUserReserveBalance;
809         // is first deposit
810         if (userReserveShareRay[user][asset] == 0) {
811             userConfig[user].setUsingReserve(reserve.assetId, true);
812         } else {
813             currUserReserveBalance = ShareBaseAccounting.getAbsoluteAmount(
814                 userReserveShareRay[user][asset],
815                 reserve.scaledTotalSupplyRay,
816                 reserve.getReserveSupply(assetConfigCache, true).unitToRay(unit)
817             ).rayToUnit(unit);
818         }
819
820         uint256 shareAmountRay = ShareBaseAccounting.getShareAmount(
821             amount.unitToRay(unit),
822             reserve.scaledTotalSupplyRay,
823             reserve.getReserveSupply(assetConfigCache, true).unitToRay(unit)
824         );
825
826         // update user info
827         userReserveShareRay[user][asset] += shareAmountRay;
828         reserve.scaledTotalSupplyRay += shareAmountRay;
829
830         IERC20Upgradeable(asset).safeTransferFrom(user, address(this), amount);
831
832         if (reserve.reinvestment != address(0)) {
833             if (IERC20Upgradeable(reserve.asset).allowance(address(this), reserve.reinvestment) < amount) {
834                 IERC20Upgradeable(reserve.asset).safeApprove(reserve.reinvestment, type(uint256).max);
835             }
836
837             IReinvestment(reserve.reinvestment).checkpoint(user, currUserReserveBalance);
838             IReinvestment(reserve.reinvestment).invest(amount);
839         } else {
840             reserve.liquidSupply += amount;
841         }
842
843         emit DepositedReserve(user, asset, reserve.reinvestment, amount);
844     }
845
```

Figure 18 Source code of *_depositReserve* function(Unfixed)

Recommendations	It is recommended to set the *getReserveSupply* function to false.
Status	Fixed.

```
844  function _depositReserve(address user, address asset, uint256 amount) internal {
845      DataTypes.AssetConfig memory assetConfigCache = assetConfig[asset];
846      DataTypes.ReserveConfig storage reserve = _reserveConfig[asset];
847      uint256 unit = assetConfigCache.decimals;
848
849      ValidationLogic.validateDepositReserve(reserve, assetConfigCache, amount);
850
851      reserve.updateIndex();
852      uint256 currReserveSupply = reserve.getReserveSupply(assetConfigCache, false);
853      uint256 currUserReserveBalance;
854
855      // is first deposit
856      if (userReserveShareRay[user][asset] == 0) {
857          userConfig[user].setUsingReserve(reserve.assetId, true);
858      } else {
859          currUserReserveBalance = ShareBaseAccounting.getAbsoluteAmount(
860              userReserveShareRay[user][asset],
861              reserve.scaledTotalSupplyRay,
862              currReserveSupply.unitToRay(unit)
863          ).rayToUnit(unit);
864      }
865
866      uint256 shareAmountRay = ShareBaseAccounting.getShareAmount(
867          amount.unitToRay(unit),
868          reserve.scaledTotalSupplyRay,
869          currReserveSupply.unitToRay(unit)
870      );
871
872      // update user info
873      userReserveShareRay[user][asset] += shareAmountRay;
874      reserve.scaledTotalSupplyRay += shareAmountRay;
875
876      IERC20Upgradeable(asset).safeTransferFrom(user, address(this), amount);
877
878      if (reserve.reinvestment != address(0)) {
879          if (IERC20Upgradeable(reserve.asset).allowance(address(this), reserve.reinvestment) < amount) {
880              IERC20Upgradeable(reserve.asset).safeApprove(reserve.reinvestment, 0);
881              IERC20Upgradeable(reserve.asset).safeApprove(reserve.reinvestment, type(uint256).max);
882          }
883
884          IReinvestment(reserve.reinvestment).checkpoint(user, currUserReserveBalance);
885          IReinvestment(reserve.reinvestment).invest(amount);
886      } else {
887          reserve.liquidSupply += amount;
888      }
889
890      emit DepositedReserve(user, asset, reserve.reinvestment, amount);
891  }
```

Figure 19 Source code of *_depositReserve* function(Fixed)

[LeverFi-7] The *configureAsset* function is improperly designed

Severity Level	**High**
Type	Business Security
Lines	Ledger.sol#L231
Description	In the Ledger contract, when the *configureAsset* function modifies the configuration, it will replace assetLength with the latest index, which will cause data confusion.

```
220     */
221     function configureAsset(
222         address asset,
223         uint256 decimals,
224         address longReinvestment,
225         ISwapAdapter swapAdapter,
226         IPriceOracleGetter oracle,
227         DataTypes.AssetState state,
228         DataTypes.AssetMode mode
229     ) public onlyOperator {
230         assetConfig[asset] = DataTypes.AssetConfig(
231             assetLength,
232             decimals,
233             longReinvestment,
234             swapAdapter,
235             oracle,
236             state,
237             mode
238         );
239
240         emit ConfiguredAsset(asset, decimals, longReinvestment, address(swapAdapter), address(oracle), state, mode);
241     }
242
```

Figure 20 Source code of *configureAsset* function(Unfixed)

Recommendations	It is recommended that assetId should remain unchanged when set.
Status	Fixed.

```
234     function configureAsset(
235         address asset,
236         uint256 decimals,
237         address longReinvestment,
238         ISwapAdapter swapAdapter,
239         IPriceOracleGetter oracle,
240         DataTypes.AssetState state,
241         DataTypes.AssetMode mode
242     ) public onlyOperator {
243         DataTypes.AssetConfig storage _assetConfig = assetConfig[asset];
244
245         _assetConfig.decimals = decimals;
246         _assetConfig.longReinvestment = longReinvestment;
247         _assetConfig.swapAdapter = swapAdapter;
248         _assetConfig.oracle = oracle;
249         _assetConfig.state = state;
250         _assetConfig.mode = mode;
251
252         emit ConfiguredAsset(asset, decimals, longReinvestment, address(swapAdapter), address(oracle), state, mode);
253     }
254
```

Figure 21 Source code of *configureAsset* function(Fixed)

[LeverFi-8] Clearing mechanism error

Severity Level	**High**
Type	Business Security
Lines	LiquidationLogic.sol#L189-221
Description	When liquidation is performed in the *executeForeclosure* function, the debt repaid by the liquidation wallet is not recorded correctly, which will cause the liquidation wallet to use a small amount of funds to pay off a large debt.

```
188                 } else {
189                     uint256 userShortAbsoluteAmount = userAssetPosition.amount
190                     .rayMul(vars.reserveConfigCache.currBorrowIndexRay)
191                     .rayToUnit(vars.assetConfig.decimals);
192
193                     uint256 amountToRepay = userShortAbsoluteAmount;
194                     // repay user short with liquidation wallet if has any
195                     if (liquidationAssetPosition.amount > 0 && liquidationAssetPosition._type == DataTypes.PositionType.Long) {
196                         if (userShortAbsoluteAmount >= liquidationAssetPosition.amount) {
197                             amountToRepay = liquidationAssetPosition.amount;
198                         }
199                         TradeLogic.executeLonging(
200                             reserveConfigs[vars.asset],
201                             vars.assetConfig,
202                             userAssetPosition,
203                             userConfigs[vars.user],
204                             params.treasury,
205                             amountToRepay,
206                             false
207                         );
208                     }
209                     vars.newPosition = TradeLogic.getNewPosition(
210                         vars.assetConfig,
211                         liquidationAssetPosition,
212                         DataTypes.UserPosition(amountToRepay, DataTypes.PositionType.Short),
213                         vars.reserveConfigCache.currBorrowIndexRay
214                     );
215                     liquidationAssetPosition.amount = vars.newPosition.amount;
216                     liquidationAssetPosition._type = vars.newPosition._type;
217                     userConfigs[params.liquidationWallet].setUsingPosition(vars.poolId, vars.newPosition.amount > 0);
218                     userAssetPosition.amount = 0;
219                     userAssetPosition._type = DataTypes.PositionType.Long;
220                     userConfigs[vars.user].setUsingPosition(vars.poolId, false);
221                 }
222             }
```

Figure 22 Source code of *executeForeclosure* function(Unfixed)

Recommendations	It is recommended to use the actual amount, not the amount paid for the user.
Status	Fixed.

```
191                 } else {
192                     uint256 userShortAbsoluteAmount = userAssetPosition.amount
193                     .rayMul(vars.reserveConfigCache.currBorrowIndexRay)
194                     .rayToUnit(vars.assetConfig.decimals);
195
196                     uint256 amountToRepay = userShortAbsoluteAmount;
197                     // repay user short with liquidation wallet if has any
198                     if (liquidationAssetPosition.amount > 0 && liquidationAssetPosition._type == DataTypes.PositionType.Long) {
199                         if (userShortAbsoluteAmount >= liquidationAssetPosition.amount) {
200                             amountToRepay = liquidationAssetPosition.amount;
201                         }
202                         TradeLogic.executeLonging(
203                             reserveConfigs[vars.asset],
204                             vars.assetConfig,
205                             userAssetPosition,
206                             userConfigs[vars.user],
207                             params.treasury,
208                             amountToRepay,
209                             false
210                         );
211                     }
212                     vars.newPosition = TradeLogic.getNewPosition(
213                         vars.assetConfig,
214                         liquidationAssetPosition,
215                         DataTypes.UserPosition(userShortAbsoluteAmount, DataTypes.PositionType.Short),
216                         vars.reserveConfigCache.currBorrowIndexRay
217                     );
```

Figure 23 Source code of *executeForeclosure* function(Fixed)

[LeverFi-9] The *getNormalizedDebt* function is improperly designed

Severity Level	Medium
Type	Business Security
Lines	ReserveLogic.sol#L59-64
Description	When querying currBorrowIndexRay through the *getNormalizedDebt* function, if user B borrows money right after user A at the same time. At this time, according to the code logic, their currBorrowIndexRay is the same, but when user B borrows money, the utilization rate changes. Theoretically currBorrowIndexRay will be different, which will result in less interest repaid by users who borrow later.

```
55  function getNormalizedDebt(
56      DataTypes.ReserveConfig memory reserve
57  ) internal view returns (uint256, uint256, uint256) {
58      uint256 timestamp = reserve.lastUpdatedTimestamp;
59      if (timestamp == block.timestamp) {
60          return (
61          reserve.reserveIndexRay,
62          reserve.protocolIndexRay,
63          reserve.reserveIndexRay + reserve.protocolIndexRay
64          );
65      } else {
66          uint256 currBorrowIndexRay = reserve.reserveIndexRay + reserve.protocolIndexRay;
67          uint256 interestRateRay = getInterestRate(
68              reserve.scaledUtilizedSupplyRay,
69              reserve.scaledTotalSupplyRay,
70              reserve.protocolRateMantissaRay,
71              reserve.utilizationBaseRateMantissaRay,
72              reserve.kinkMantissaRay,
73              reserve.multiplierAnnualRay,
74              reserve.jumpMultiplierAnnualRay
75          );
76          if (interestRateRay == 0) {
77              return (
78              reserve.reserveIndexRay,
79              reserve.protocolIndexRay,
80              reserve.reserveIndexRay + reserve.protocolIndexRay
81              );
82          }
```

Figure 24 Source code of *getNormalizedDebt* function(Unfixed)

Recommendations	It is recommended to delete the if judgment.
Status	Fixed.

```
function getNormalizedDebt(
    DataTypes.ReserveConfig memory reserve
) internal view returns (uint256, uint256, uint256) {
    uint256 currBorrowIndexRay = reserve.reserveIndexRay + reserve.protocolIndexRay;
    uint256 interestRateRay = getInterestRate(
        reserve.scaledUtilizedSupplyRay,
        reserve.scaledTotalSupplyRay,
        reserve.protocolRateMantissaRay,
        reserve.utilizationBaseRateMantissaRay,
        reserve.kinkMantissaRay,
        reserve.multiplierAnnualRay,
        reserve.jumpMultiplierAnnualRay
    );
    if (interestRateRay == 0) {
        return (
        reserve.reserveIndexRay,
        reserve.protocolIndexRay,
        reserve.reserveIndexRay + reserve.protocolIndexRay
        );
    }
    uint256 cumulatedInterestIndexRay = InterestUtils.getCompoundedInterest(
        interestRateRay, reserve.lastUpdatedTimestamp, block.timestamp
    );
    uint256 growthIndexRay = currBorrowIndexRay.rayMul(cumulatedInterestIndexRay) - currBorrowIndexRay;
    uint256 protocolInterestRatio = reserve.protocolRateMantissaRay.rayDiv(interestRateRay);
    uint256 nextReserveIndexRay = reserve.reserveIndexRay + growthIndexRay.rayMul(MathUtils.RAY - protocolInterestRatio);
    uint256 nextProtocolIndexRay = reserve.protocolIndexRay + growthIndexRay.rayMul(protocolInterestRatio);
    return (
    nextReserveIndexRay,
    nextProtocolIndexRay,
    nextProtocolIndexRay + nextReserveIndexRay
    );
}
```

Figure 25 Source code of *getNormalizedDebt* function(Fixed)

[LeverFi-10] Implementation flaws in the *reinvestReserveSupply* and *reinvestCollateralSupply* functions

Severity Level	**Medium**
Type	Business Security
Lines	Ledger.sol#L759-795
Description	In the *reinvestReserveSupply* and *reinvestCollateralSupply* functions, the lack of approve for New_reinvestment will cause the function call to fail.

```
759  function reinvestReserveSupply(address asset) external onlyOperator {
760
761
762      DataTypes.ReserveConfig storage reserveConfig = _reserveConfig[asset];
763
764      require(reserveConfig.state == DataTypes.AssetState.Disabled, "state is not disabled");
765      require(reserveConfig.reinvestment != address(0), "reinvestment is not set");
766
767      IReinvestment(reserveConfig.reinvestment).invest(reserveConfig.liquidSupply);
768
769      emit ReinvestedReserveSupply(asset, reserveConfig.liquidSupply);
770
771      reserveConfig.liquidSupply = 0;
772
773  }
774
775  /**
776   * @notice Reinvests collateral supply
777   * @param asset Underlying asset address
778   * @param reinvestment Address where the asset is reinvested in
779   */
780  function reinvestCollateralSupply(address asset, address reinvestment) external onlyOperator {
781
782
783      uint256 poolId = collateralPoolList[asset][reinvestment];
784      DataTypes.CollateralConfig storage collateralConfig = _collateralConfig[poolId];
785
786      require(collateralConfig.state == DataTypes.AssetState.Disabled, "state is not disabled");
787      require(collateralConfig.reinvestment != address(0), "reinvestment is not set");
788
789      IReinvestment(collateralConfig.reinvestment).invest(collateralConfig.liquidSupply);
790
791      emit ReinvestedCollateralSupply(asset, reinvestment, collateralConfig.liquidSupply);
792
793      collateralConfig.liquidSupply = 0;
794
795  }
```

Figure 26 Source code of related functions(Unfixed)

Recommendations	It is recommended to increase approve.
Status	Fixed.

```
802     */
803     function reinvestReserveSupply(address asset) external onlyOperator {
804
805
806         DataTypes.ReserveConfig storage reserveConfig = _reserveConfig[asset];
807
808         require(reserveConfig.state == DataTypes.AssetState.Disabled, "state is not disabled");
809         require(reserveConfig.reinvestment != address(0), "reinvestment is not set");
810
811         IERC20Upgradeable(asset).safeApprove(reserveConfig.reinvestment, reserveConfig.liquidSupply);
812         IReinvestment(reserveConfig.reinvestment).invest(reserveConfig.liquidSupply);
813
814         emit ReinvestedReserveSupply(asset, reserveConfig.liquidSupply);
815
816         reserveConfig.liquidSupply = 0;
817
818     }
819
820     /**
821      * @notice Reinvests collateral supply
822      * @param asset Underlying asset address
823      * @param reinvestment Address where the asset is reinvested in
824      */
825     function reinvestCollateralSupply(address asset, address reinvestment) external onlyOperator {
826
827
828         uint256 poolId = collateralPoolList[asset][reinvestment];
829         DataTypes.CollateralConfig storage collateralConfig = _collateralConfig[poolId];
830
831         require(collateralConfig.state == DataTypes.AssetState.Disabled, "state is not disabled");
832         require(collateralConfig.reinvestment != address(0), "reinvestment is not set");
833
834         IERC20Upgradeable(asset).safeApprove(collateralConfig.reinvestment, collateralConfig.liquidSupply);
835         IReinvestment(collateralConfig.reinvestment).invest(collateralConfig.liquidSupply);
836
837         emit ReinvestedCollateralSupply(asset, reinvestment, collateralConfig.liquidSupply);
838
839         collateralConfig.liquidSupply = 0;
840
841     }
```

Figure 27 Source code of related functions(Fixed)

[LeverFi-11] Unsafe call

Severity Level	Medium
Type	General Vulnerability
Lines	ZeroexSwapAdapter.sol#L35
Description	Unsafe call method. In the *swap* function of the ZeroexSwapAdapter contract, because the swapBytesData parameter is controllable, the approveRouter and swapBytes are also controllable. Then an attacker can use call to perform arbitrary operations. For example, approveRouter is specified as a token, and the swapBytes parameter is the token authorization behavior (the contract is authorized to the attacker), then the attacker can withdraw the tokens in the contract.

```
function swap(address, address, uint256 amountToSwap, bytes memory swapBytesData)  external override returns (uint256) {

    (address approveRouter, address sellingAsset, address buyingAsset, bytes memory swapBytes) = abi.decode(swapBytesData, (address,address,address,bytes));

    uint256 buyingAssetBalancePrior = IERC20Upgradeable(buyingAsset).balanceOf(address(this));

    IERC20Upgradeable(sellingAsset).safeTransferFrom(msg.sender, address(this), amountToSwap);


    // approve to appropriate router (Note: router is the part of swapBytes data. Do not take from the swap func arg. The function arg is given because for other swaps like 1inch

    IERC20Upgradeable(sellingAsset).safeIncreaseAllowance(approveRouter, amountToSwap);

    (bool success, bytes memory data) = approveRouter.call(swapBytes);

    if(!success){
        assembly {
            let returndata_size := mload(data)
            revert(add(32, data), returndata_size)
        }
    }

    // check balance after swap
    uint256 buyingAssetBalance = IERC20Upgradeable(buyingAsset).balanceOf(address(this));
    uint256 receivedAmount = buyingAssetBalance - buyingAssetBalancePrior;


    IERC20Upgradeable(buyingAsset).safeTransfer(msg.sender, receivedAmount);

    return receivedAmount;
}
}
```

Figure 28 Source code of *swap* function (ZeroexSwapAdapter.sol)(Unfixed)

Recommendations	It is recommended to directly call this interface for swap instead of using the call method.
Status	Fixed.

```
    /// @param router Router address to enable
    function configureRoutersList(address router, bool value) external onlyOwner {
        emit ConfiguredRoutersList(router, routersList[router], value);
        routersList[router] = value;
    }
```

Figure 29 Source code of *configureRoutersList* function (ZeroexSwapAdapter.sol)(Fixed)

[LeverFi-12] The *configureReserve* function design flaw

Severity Level	Medium
Type	Business Security
Lines	Ledger.sol#L304
Description	First, in the *configureReserve* function, the operator can modify the reinvestment arbitrarily. It is assumed that the operator initially sets the reinvestment to a non-zero address, and the user stakes during this period, and then the operator sets the reinvestment to zero. At this time, when the user stakes in, the calculation of the *getShareAmount* function in the *_depositReserve* function will be wrong, because the *getReserveAvailableSupply* function here can only obtain the total amount of reserve.reinvestment that is zero, and does not obtain the amount of reinvestment that is not zero, so the calculation error is caused.

```
290  function configureReserve(
291      address asset,
292      address reinvestment,
293      uint256 feeMantissaRay,
294      uint256 protocolRateMantissaRay,
295      uint256 utilizationBaseRateMantissaRay,
296      uint256 kinkMantissaRay,
297      uint256 multiplierPerAnnualRay,
298      uint256 jumpMultiplierPerAnnualRay,
299      DataTypes.AssetState state
300  ) public onlyOperator {
301      DataTypes.ReserveConfig storage reserve = _reserveConfig[asset];
302
303      require(jumpMultiplierPerAnnualRay >= multiplierPerAnnualRay, "Jump multiplier must be greater or equal normal multiplier");
304      reserve.reinvestment = reinvestment;
305      reserve.feeMantissaRay = feeMantissaRay;
306      reserve.protocolRateMantissaRay = protocolRateMantissaRay;
307      reserve.utilizationBaseRateMantissaRay = utilizationBaseRateMantissaRay;
308      reserve.kinkMantissaRay = kinkMantissaRay;
309      reserve.multiplierAnnualRay = multiplierPerAnnualRay;
310      reserve.jumpMultiplierAnnualRay = jumpMultiplierPerAnnualRay;
311      reserve.state = state;
312
313      emit ConfiguredReserve(
314          asset,
315          reinvestment,
316          feeMantissaRay,
317          utilizationBaseRateMantissaRay,
318          kinkMantissaRay,
319          multiplierPerAnnualRay,
320          jumpMultiplierPerAnnualRay,
321          state
322      );
323  }
```

Figure 30 Source code of *configureReserve* function (Ledger.sol)(Unfixed)

```
797     /********************* INTERNAL ****************************/
798     function _depositReserve(address user, address asset, uint256 amount) internal {
799         DataTypes.AssetConfig memory assetConfigCache = assetConfig[asset];
800         DataTypes.ReserveConfig storage reserve = _reserveConfig[asset];
801         uint256 unit = assetConfigCache.decimals;
802
803         ValidationLogic.validateDepositReserve(reserve, assetConfigCache, amount);
804
805         reserve.updateIndex();
806
807         uint256 currUserReserveBalance;
808         // is first deposit
809         if (userReserveShareRay[user][asset] == 0) {
810             userConfig[user].setUsingReserve(reserve.assetId, true);
811         } else {
812             currUserReserveBalance = ShareBaseAccounting.getAbsoluteAmount(
813                 userReserveShareRay[user][asset],
814                 reserve.scaledTotalSupplyRay,
815                 reserve.getReserveSupply(assetConfigCache, true).unitToRay(unit)
816             ).rayToUnit(unit);
817         }
818
819         uint256 shareAmountRay = ShareBaseAccounting.getShareAmount(
820             amount.unitToRay(unit),
821             reserve.scaledTotalSupplyRay,
822             reserve.getReserveSupply(assetConfigCache, true).unitToRay(unit)
823         );
824
```

Figure 31 Source code of *_depositReserve* function (Ledger.sol)(Unfixed)

```
21      */
22      function getReserveSupply(
23          DataTypes.ReserveConfig storage reserve,
24          DataTypes.AssetConfig memory assetConfig,
25          bool claimable
26      ) internal view returns (uint256 poolAmount) {
27          if (!claimable) {
28              poolAmount += getUtilizedSupply(reserve, assetConfig);
29          }
30          poolAmount += GeneralLogic.getReserveAvailableSupply(reserve);
31      }
32
```

Figure 32 Source code of *getReserveSupply* function (ReserveLogic.sol)(Unfixed)

```
24
25      function getReserveAvailableSupply(
26          DataTypes.ReserveConfig memory reserve
27      ) public view returns (uint256 supply) {
28          if (reserve.reinvestment == address(0)) {
29              supply = reserve.liquidSupply;
30          } else {
31              supply = IReinvestment(reserve.reinvestment).totalSupply();
32          }
33      }
```

Figure 33 Source code of *getReserveAvailableSupply* function (GeneralLogic.sol)(Unfixed)

Recommendations	It is recommended that the project side add a state in which users cannot stake.
Status	Fixed. The project party has added this function to avoid the situation that users can still stake when modifying the reinvestment.

```
338     */
339     function configureReserveReinvestment(
340         address asset,
341         address reinvestment
342     ) public onlyOperator {
343         DataTypes.ReserveConfig storage reserve = _reserveConfig[asset];
344 
345         require(reserve.assetId != 0, "asset reserve is not initialized");
346         require(reserve.state == DataTypes.AssetState.Disabled);
347 
348         emit ConfiguredReserveReinvestment(asset, reserve.reinvestment, reinvestment);
349 
350         reserve.reinvestment = reinvestment;
351     }
352 
```

Figure 34 Source code of *configureReserveReinvestment* function (Ledger.sol)(Fixed)

[LeverFi-13] Centralization risk

Severity Level	Medium
Type	Business Security
Lines	Ledger.sol
Description	In the Ledger contract, the operator permission can modify the relevant parameters in the contract, which is high permission.

```
146     */
147     function setLeverageFactor(uint256 leverageFactor_) external onlyOperator {
148         require(leverageFactor_ >= 1e18, "leverage factor cannot be lower than 1x");
149         require(leverageFactor_ <= 10e18, "leverage factor cannot be lower than 10x");
150
151         emit LeverageFactorUpdated(leverageFactor, leverageFactor_);
152
153         leverageFactor = leverageFactor_;
154     }
155
156     /**
157      * @notice Setter for tradeFeeMantissa
158      * @param tradeFeeMantissa_ The new tradeFeeMantissa
159      */
160     function setTradeFee(uint256 tradeFeeMantissa_) external onlyOperator {
161         require(tradeFeeMantissa_ <= 0.1e18, "fee cannot be more than 10%");
162
163         emit TradeFeeUpdated(tradeFeeMantissa, tradeFeeMantissa_);
164
165         tradeFeeMantissa = tradeFeeMantissa_;
166     }
167
168     /**
169      * @notice Setter for liquidationRatioMantissa
170      * @param liquidationRatioMantissa_ The new liquidationRatioMantissa
171      */
172     function setLiquidationRatio(uint256 liquidationRatioMantissa_) external onlyOperator {
173         require(liquidationRatioMantissa_ >= 0.5e18, "fee cannot be less than 50%");
174         require(liquidationRatioMantissa_ <= 0.9e18, "fee cannot be more than 90%");
175
176         emit LiquidationRatioUpdated(liquidationRatioMantissa, liquidationRatioMantissa_);
177
178         liquidationRatioMantissa = liquidationRatioMantissa_;
179     }
180
181     /********************** CORE FUNCTIONS ****************************/
```

Figure 35 Source code of related functions

```
234 function configureAsset(
235     address asset,
236     uint256 decimals,
237     address longReinvestment,
238     ISwapAdapter swapAdapter,
239     IPriceOracleGetter oracle,
240     DataTypes.AssetState state,
241     DataTypes.AssetMode mode
242 ) public onlyOperator {
243     DataTypes.AssetConfig storage _assetConfig = assetConfig[asset];
244
245     _assetConfig.decimals = decimals;
246     _assetConfig.longReinvestment = longReinvestment;
247     _assetConfig.swapAdapter = swapAdapter;
248     _assetConfig.oracle = oracle;
249     _assetConfig.state = state;
250     _assetConfig.mode = mode;
251
252     emit ConfiguredAsset(asset, decimals, longReinvestment, address(swapAdapter), address(oracle), state, mode);
253 }
254
```

Figure 36 Source code of *configureAsset* function

Recommendations	It is recommended to use a multi-signature wallet to manage operator permissions.
Status	Fixed. The project party is planned that a Gnosis multi-signature wallet will be used as

operator for the Ledger.

[LeverFi-14] The *_repayShort* function is improperly designed

Severity Level	Medium
Type	Business Security
Lines	Ledger.sol#1074-1107
Description	The essence of the *_repayShort* function is that the user repays the debt, but the user can use the *_repayShort* function to add positions. Because the function does not determine whether the user's position is long, and the money transferred by the user is not transferred to the investment contract. When the last user wants to withdraw funds, it will cause the call to fail because the user withdraws funds from the investment contract.

```
1074    function _repayShort(address user, address asset, uint256 amount) internal {
1075        _reserveConfig[asset].updateIndex();
1076
1077        (uint256 positionAbsoluteAmount,) = GeneralLogic.getUserPosition(
1078            assetConfig[asset],
1079            _userPosition[user][asset],
1080            _reserveConfig[asset].getBorrowIndex()
1081        );
1082
1083        // cap amount to max repayable
1084        if (amount > positionAbsoluteAmount) {
1085            amount = positionAbsoluteAmount;
1086        }
1087
1088        ValidationLogic.validateRepayShort(
1089            userLastTradeBlock[user],
1090            DataTypes.ValidateRepayShortParams(
1091                user,
1092                asset,
1093                amount
1094            )
1095        );
1096
1097        IERC20Upgradeable(asset).safeTransferFrom(user, address(this), amount);
1098
1099        TradeLogic.executeLonging(
1100            _reserveConfig[asset],
1101            assetConfig[asset],
1102            _userPosition[user][asset],
1103            userConfig[user],
1104            treasury,
1105            amount,
1106            false
1107        );
1108
```

Figure 37 Source code of *_repayShort* function(Unfixed)

Recommendations	It is recommended to judge whether the user's position is a short type in *validateRepayShort*.
Status	Fixed.

```
1074    function _repayShort(address user, address asset, uint256 amount) internal {
1075        _reserveConfig[asset].updateIndex();
1076
1077        (uint256 positionAbsoluteAmount,) = GeneralLogic.getUserPosition(
1078            assetConfig[asset],
1079            _userPosition[user][asset],
1080            _reserveConfig[asset].getBorrowIndex()
1081        );
1082
1083        // cap amount to max repayable
1084        if (amount > positionAbsoluteAmount) {
1085            amount = positionAbsoluteAmount;
1086        }
1087
1088        ValidationLogic.validateRepayShort(
1089            userLastTradeBlock[user],
1090            DataTypes.ValidateRepayShortParams(
1091                user,
1092                asset,
1093                amount
1094            )
1095        );
1096
1097        IERC20Upgradeable(asset).safeTransferFrom(user, address(this), amount);
1098
1099        TradeLogic.executeLonging(
1100            _reserveConfig[asset],
1101            assetConfig[asset],
1102            _userPosition[user][asset],
1103            userConfig[user],
1104            treasury,
1105            amount,
1106            true
1107        );
1108
1109        emit RepaidShort(user, asset, amount);
```

Figure 38 Source code of *_repayShort* function(Fixed)

[LeverFi-15] Improperly designed *_depositCollateral* function

Severity Level	**Low**
Type	Business Security
Lines	Ledger.sol#L914-916
Description	In the *_depositCollateral* function, when verifying whether the stake meets the requirements, the current user's share (currUserDepositShareRay) plus the amount is incorrectly used to determine whether the minimum stake amount is met.

```
908  function _depositCollateral(address user, address asset, address reinvestment, uint256 amount) internal {
909      uint256 collateralPoolId = collateralPoolList[asset][reinvestment];
910      DataTypes.AssetConfig memory assetConfigCache = assetConfig[asset];
911      DataTypes.CollateralConfig storage collateral = _collateralConfig[collateralPoolId];
912      uint256 unit = assetConfigCache.decimals;
913
914      uint256 currUserDepositShareRay = userCollateralShareRay[user][collateralPoolId];
915
916      ValidationLogic.validateDepositCollateral(collateral, assetConfigCache, userLastTradeBlock[user], amount, currUserDepositShareRay);
917
```

Figure 39 Source code of *_depositCollateral* function(Unfixed)

Recommendations	It is recommended to use the current user's stake amount.
Status	Fixed.

```
939  function _depositCollateral(address user, address asset, address reinvestment, uint256 amount) internal {
940      uint256 collateralPoolId = collateralPoolList[asset][reinvestment];
941      DataTypes.AssetConfig memory assetConfigCache = assetConfig[asset];
942      DataTypes.CollateralConfig storage collateral = _collateralConfig[collateralPoolId];
943      uint256 unit = assetConfigCache.decimals;
944
945      uint256 currUserDepositShareRay = userCollateralShareRay[user][collateralPoolId];
946
947      uint256 currCollateralBalance;
948      // is first deposit
949      if (currUserDepositShareRay == 0) {
950          userConfig[user].setUsingCollateral(collateralPoolId, true);
951      } else {
952          currCollateralBalance = ShareBaseAccounting.getAbsoluteAmount(
953              currUserDepositShareRay,
954              collateral.scaledTotalSupplyRay,
955              GeneralLogic.getCollateralSupply(collateral).unitToRay(unit)
956          ).rayToUnit(unit);
957      }
958
959      ValidationLogic.validateDepositCollateral(collateral, assetConfigCache, userLastTradeBlock[user], amount, currCollateralBalance);
960
```

Figure 40 Source code of *_depositCollateral* function(Fixed)

[LeverFi-16] The *validateTrade* function check error

Severity Level	**Low**
Type	Business Security
Lines	ValidationLogic.sol#L170-174
Description	In the *validateTrade* function, the state of reserve is repeatedly judged, and the state of assetConfig should be judged here.

```
162  function validateTrade(
163      mapping(address => DataTypes.ReserveConfig) storage reserve,
164      mapping(address => DataTypes.AssetConfig) storage assetConfig,
165      mapping(address => DataTypes.UserPosition) storage userPosition,
166      uint256 userLastTradeBlock,
167      DataTypes.ValidateTradeParams memory params
168  ) external view {
169      ValidateTradeVars memory vars;
170      require(reserve[params.shortAsset].state == DataTypes.AssetState.Active, "selling asset not active");
171      require(reserve[params.longAsset].state == DataTypes.AssetState.Active, "buying asset not active");
172      require(reserve[params.shortAsset].state == DataTypes.AssetState.Active, "selling asset not active");
173      require(reserve[params.longAsset].state == DataTypes.AssetState.Active, "buying asset not active");
174      require(userLastTradeBlock != block.number, "cannot execute trade with other actions");
175      require(params.tradeAmount != 0, "amount is zero");
176      if (userPosition[params.shortAsset]._type == DataTypes.PositionType.Short) {
177          vars.amountToShort = params.tradeAmount;
```

Figure 41 Source code of *validateTrade* function(Unfixed)

Recommendations	It is recommended to modify reserve to assetConfig.
Status	Fixed.

```
160   **/
161  function validateTrade(
162      mapping(address => DataTypes.ReserveConfig) storage reserve,
163      mapping(address => DataTypes.AssetConfig) storage assetConfig,
164      mapping(address => DataTypes.UserPosition) storage userPosition,
165      uint256 userLastTradeBlock,
166      DataTypes.ValidateTradeParams memory params
167  ) external view {
168      ValidateTradeVars memory vars;
169      require(assetConfig[params.shortAsset].state == DataTypes.AssetState.Active, "selling asset not active");
170      require(assetConfig[params.longAsset].state == DataTypes.AssetState.Active, "buying asset not active");
171      require(reserve[params.shortAsset].state == DataTypes.AssetState.Active, "selling asset not active");
172      require(reserve[params.longAsset].state == DataTypes.AssetState.Active, "buying asset not active");
173      require(userLastTradeBlock != block.number, "cannot execute trade with other actions");
174      require(params.tradeAmount != 0, "amount is zero");
175      if (userPosition[params.shortAsset]._type == DataTypes.PositionType.Short) {
176          vars.amountToShort = params.tradeAmount;
177      } else {
178          if (userPosition[params.shortAsset].amount < params.tradeAmount) {
```

Figure 42 Source code of *validateTrade* function(Fixed)

[LeverFi-17] The *configureCollateral* function cannot modify reinvestment

Severity Level	**Low**
Type	Business Security
Lines	Ledger.sol#
Description	Because reinvestment cannot be set in the *configureCollateral* function, but there is a function for reinvesting new reinvestment in the contract, the reinvestment function will not be available.

```
361  function initializeCollateral(
362      address asset,
363      address reinvestment,
364      uint256 feeMantissa,
365      uint256 ltv,
366      uint256 minDeposit
367  ) external onlyOperator {
368      require(collateralPoolList[asset][reinvestment] == 0, "already initialized");
369
370      collateralPoolLength++;
371      collateralPoolList[asset][reinvestment] = collateralPoolLength;
372
373      DataTypes.CollateralConfig storage collateral = _collateralConfig[collateralPoolLength];
374
375      configureCollateral(asset, reinvestment, feeMantissa, ltv, minDeposit, DataTypes.AssetState.Active);
376
377      collateral.poolId = collateralPoolLength;
378      collateral.asset = asset;
379      collateral.reinvestment = reinvestment;
380
381      emit InitializedCollateral(collateral.poolId, asset, reinvestment);
382  }
383
```

Figure 43 Source code of *initializeCollateral* function

```
393  function configureCollateral(
394      address asset,
395      address reinvestment,
396      uint256 feeMantissa,
397      uint256 ltv,
398      uint256 minDeposit,
399      DataTypes.AssetState state
400  ) public onlyOperator {
401      DataTypes.CollateralConfig storage collateral = _collateralConfig[collateralPoolList[asset][reinvestment]];
402
403      collateral.feeMantissa = feeMantissa;
404      collateral.ltv = ltv;
405      collateral.minBalance = minDeposit;
406      collateral.state = state;
407
408      emit ConfiguredCollateral(asset, reinvestment, feeMantissa, ltv, minDeposit, state);
409  }
410
```

Figure 44 Source code of *configureCollateral* function

```
825  function reinvestCollateralSupply(address asset, address reinvestment) external onlyOperator {
826
827
828      uint256 poolId = collateralPoolList[asset][reinvestment];
829      DataTypes.CollateralConfig storage collateralConfig = _collateralConfig[poolId];
830
831      require(collateralConfig.state == DataTypes.AssetState.Disabled, "state is not disabled");
832      require(collateralConfig.reinvestment != address(0), "reinvestment is not set");
833
834      IERC20Upgradeable(asset).safeApprove(collateralConfig.reinvestment, collateralConfig.liquidSupply);
835      IReinvestment(collateralConfig.reinvestment).invest(collateralConfig.liquidSupply);
836
837      emit ReinvestedCollateralSupply(asset, reinvestment, collateralConfig.liquidSupply);
838
839      collateralConfig.liquidSupply = 0;
840
841  }
```

Figure 45 Source code of *reinvestCollateralSupply* function

Recommendations	It is recommended to add collateralPoolList[asset][new_reinvestment] = collateralPoolList[asset][reinvestment]; del collateralPoolList[asset][reinvestment] to the configureCollateral function, and then update the reinvestment in the new collateral. When setting, you should also judge whether collateralPoolList[asset][new_reinvestment] has been set. Of course, when setting, also judge the state, which can only be set when Disabled.
Status	Acknowledged. Project party description: Setting new reinvestment to an existing collateral should not be possible by design. To replace a collateral with new reinvestment, we will initialize a collateral of the same asset with new reinvestment address treated as a new pool, and make the old one to `Withdrawing` state.

[LeverFi-18] lack of judgment on value

Severity Level	Info
Type	Business Security
Lines	Ledger.sol#L137-161
Description	The following function lacks judgment on the input value, resulting in any value can be set.

```
136     */
137     function setLeverageFactor(uint256 leverageFactor_) external onlyOperator {
138         emit LeverageFactorUpdated(leverageFactor, leverageFactor_);
139
140         leverageFactor = leverageFactor_;
141     }
142
143     /**
144      * @notice Setter for tradeFeeMantissa
145      * @param tradeFeeMantissa_ The new tradeFeeMantissa
146      */
147     function setTradeFee(uint256 tradeFeeMantissa_) external onlyOperator {
148         emit TradeFeeUpdated(tradeFeeMantissa, tradeFeeMantissa_);
149
150         tradeFeeMantissa = tradeFeeMantissa_;
151     }
152
153     /**
154      * @notice Setter for liquidationRatioMantissa
155      * @param liquidationRatioMantissa_ The new liquidationRatioMantissa
156      */
157     function setLiquidationRatio(uint256 liquidationRatioMantissa_) external onlyOperator {
158         emit LiquidationRatioUpdated(liquidationRatioMantissa, liquidationRatioMantissa_);
159
160         liquidationRatioMantissa = liquidationRatioMantissa_;
161     }
162
```

Figure 46 Source code of related functions(Unfixed)

```
118     function initialize(
119         address treasury_,
120         uint256 leverageFactor_,
121         uint256 liquidationRatioMantissa_,
122         uint256 tradeFeeMantissa_
123     ) public initializer {
124         treasury = treasury_;
125         leverageFactor = leverageFactor_;
126         liquidationRatioMantissa = liquidationRatioMantissa_;
127         tradeFeeMantissa = tradeFeeMantissa_;
128
129         _grantRole(OPERATOR_ROLE, msg.sender);
130         _grantRole(LIQUIDATE_EXECUTOR, msg.sender);
131     }
132
```

Figure 47 Source code of *initialize* function(Unfixed)

Recommendations	It is recommended to judge the range of the input value.
Status	Fixed.

```
137     */
138     function setLeverageFactor(uint256 leverageFactor_) external onlyOperator {
139         require(leverageFactor_ >= 1e18, "leverage factor cannot be lower than 1x");
140         require(leverageFactor_ <= 10e18, "leverage factor cannot be lower than 10x");
141
142         emit LeverageFactorUpdated(leverageFactor, leverageFactor_);
143
144         leverageFactor = leverageFactor_;
145     }
146
147     /**
148      * @notice Setter for tradeFeeMantissa
149      * @param tradeFeeMantissa_ The new tradeFeeMantissa
150      */
151     function setTradeFee(uint256 tradeFeeMantissa_) external onlyOperator {
152         require(tradeFeeMantissa_ <= 0.1e18, "fee cannot be more than 10%");
153
154         emit TradeFeeUpdated(tradeFeeMantissa, tradeFeeMantissa_);
155
156         tradeFeeMantissa = tradeFeeMantissa_;
157     }
158
159     /**
160      * @notice Setter for liquidationRatioMantissa
161      * @param liquidationRatioMantissa_ The new liquidationRatioMantissa
162      */
163     function setLiquidationRatio(uint256 liquidationRatioMantissa_) external onlyOperator {
164         require(liquidationRatioMantissa_ >= 0.5e18, "fee cannot be less than 50%");
165         require(liquidationRatioMantissa_ <= 0.9e18, "fee cannot be more than 90%");
166
167         emit LiquidationRatioUpdated(liquidationRatioMantissa, liquidationRatioMantissa_);
168
169         liquidationRatioMantissa = liquidationRatioMantissa_;
170     }
```

Figure 48 Source code of related functions(Fixed)

```
119     function initialize(
120         address treasury_,
121         uint256 leverageFactor_,
122         uint256 liquidationRatioMantissa_,
123         uint256 tradeFeeMantissa_
124     ) public initializer {
125         require(leverageFactor_ >= 1e18, "leverage factor cannot be lower than 1x");
126         require(leverageFactor_ <= 10e18, "leverage factor cannot be lower than 10x");
127         require(liquidationRatioMantissa_ >= 0.5e18, "fee cannot be less than 50%");
128         require(liquidationRatioMantissa_ <= 0.9e18, "fee cannot be more than 90%");
129         require(tradeFeeMantissa_ <= 0.1e18, "fee cannot be more than 10%");
130
131         treasury = treasury_;
132         leverageFactor = leverageFactor_;
133         liquidationRatioMantissa = liquidationRatioMantissa_;
134         tradeFeeMantissa = tradeFeeMantissa_;
135
136         _setRoleAdmin(OPERATOR_ROLE, OPERATOR_ROLE);
137         _setRoleAdmin(LIQUIDATE_EXECUTOR, OPERATOR_ROLE);
138
139         _grantRole(OPERATOR_ROLE, msg.sender);
140         _grantRole(LIQUIDATE_EXECUTOR, msg.sender);
141     }
```

Figure 49 Source code of *initialize* function(Fixed)

[LeverFi-19] The *getUserLiquidity* function design flaw

Severity Level	Info
Type	Business Security
Lines	GeneralLogic.sol#L185-188
Description	When calculating availableLeverageUsd, whether vars.pnlUsd>0 or vars.pnlUsd<0, should use (int256(vars.totalCollateralUsdPostLtv) + vars.pnlUsd) * int256(params.leverageFactor) / int256(1e18) - int(vars.totalShortUsd)), because the part of the revenue can still be used as part of the user.

```
176                 }
177             }
178             vars.userConfigCache.collateral = vars.userConfigCache.collateral >> 1;
179             vars.userConfigCache.position = vars.userConfigCache.position >> 1;
180             vars.i++;
181         }
182         vars.pnlUsd = int256(vars.totalLongUsd) - int256(vars.totalShortUsd);
183         vars.isLiquidatable = (int256(vars.totalCollateralUsdPreLtv.wadMul(params.liquidationRatioMantissa)) + vars.pnlUsd) < 0;
184         vars.totalLeverageUsd = vars.totalCollateralUsdPostLtv.wadMul(params.leverageFactor);
185         vars.availableLeverageUsd = vars.pnlUsd > 0
186         ? (int256(vars.totalCollateralUsdPostLtv)) * int256(params.leverageFactor) / int256(1e18) - int(vars.totalShortUsd)
187         : (int256(vars.totalCollateralUsdPostLtv) + vars.pnlUsd) * int256(params.leverageFactor) / int256(1e18) - int(vars.totalShortUsd);
188         return (
189         vars.totalCollateralUsdPreLtv,
190         vars.totalCollateralUsdPostLtv,
191         vars.totalLongUsd,
192         vars.totalShortUsd,
193         vars.pnlUsd,
194         vars.totalLeverageUsd,
195         vars.availableLeverageUsd,
196         vars.isLiquidatable
197         );
198     }
```

Figure 50 Source code of *getUserLiquidity* function(Unfixed)

Recommendations	It is recommended to treat the revenue part as the part used by the user.
Status	Fixed.

```
180     }
181     vars.pnlUsd = int256(vars.totalLongUsd) - int256(vars.totalShortUsd);
182     vars.isLiquidatable = (int256(vars.totalCollateralUsdPreLtv.wadMul(params.liquidationRatioMantissa)) + vars.pnlUsd) < 0;
183     vars.totalLeverageUsd = vars.totalCollateralUsdPostLtv.wadMul(params.leverageFactor);
184     vars.availableLeverageUsd = (int256(vars.totalCollateralUsdPostLtv) + vars.pnlUsd) * int256(params.leverageFactor) / int256(1e18) - int(vars.totalShortUsd);
185
186     return (
187     vars.totalCollateralUsdPreLtv,
188     vars.totalCollateralUsdPostLtv,
189     vars.totalLongUsd,
190     vars.totalShortUsd,
191     vars.pnlUsd,
192     vars.totalLeverageUsd,
193     vars.availableLeverageUsd,
194     vars.isLiquidatable
195     );
196 }
```

Figure 51 Source code of *getUserLiquidity* function(Fixed)

[LeverFi-20] Lack of judgment on whether to add assets

Severity Level	Info
Type	Business Security
Lines	Ledger.sol#L175-197,244-275
Description	In the *initializeAsset* function of the Ledger contract, it is not determined whether the asset has been added. If the operator is not set properly, it will cause an error in the assetList record. Similarly, the *initializeCollateral* function should also determine whether the collateralPoolList[asset][reinvestment] has been added.

```
175    function initializeAsset(
176        address asset,
177        uint256 decimals,
178        address longReinvestment,
179        ISwapAdapter swapAdapter,
180        IPriceOracleGetter oracle,
181        DataTypes.AssetState state,
182        DataTypes.AssetMode mode
183    ) external onlyOperator {
184        assetLength++;
185        assetList[assetLength] = asset;
186
187        configureAsset(
188            asset,
189            decimals,
190            longReinvestment,
191            swapAdapter,
192            oracle,
193            state,
194            mode
195        );
196
197        emit InitializedAssetConfig(assetLength, asset);
198    }
199
```

Figure 52 Source code of *initializeAsset* function(Unfixed)

```
333  function initializeCollateral(
334      address asset,
335      address reinvestment,
336      uint256 feeMantissa,
337      uint256 ltv,
338      uint256 minDeposit
339  ) external onlyOperator {
340      collateralPoolLength++;
341      collateralPoolList[asset][reinvestment] = collateralPoolLength;
342
343      DataTypes.CollateralConfig storage collateral = _collateralConfig[collateralPoolLength];
344
345      configureCollateral(asset, reinvestment, feeMantissa, ltv, minDeposit, DataTypes.AssetState.Active);
346
347      collateral.poolId = collateralPoolLength;
348      collateral.asset = asset;
349      collateral.reinvestment = reinvestment;
350
351      emit InitializedCollateral(collateral.poolId, asset, reinvestment);
352  }
353
```

Figure 53 Source code of *initializeCollateral* function(Unfixed)

Recommendations	It is recommended to determine whether the asset has been added.
Status	Fixed.

```
183     */
184     function initializeAsset(
185         address asset,
186         uint256 decimals,
187         address longReinvestment,
188         ISwapAdapter swapAdapter,
189         IPriceOracleGetter oracle,
190         DataTypes.AssetState state,
191         DataTypes.AssetMode mode
192     ) external onlyOperator {
193         require(assetConfig[asset].assetId == 0, "already initialized");
194
195         assetLength++;
196         assetList[assetLength] = asset;
197
198         configureAsset(
199             asset,
200             decimals,
201             longReinvestment,
202             swapAdapter,
203             oracle,
204             state,
205             mode
206         );
207
208         emit InitializedAssetConfig(assetLength, asset);
209     }
210
```

Figure 54 Source code of *initializeAsset* function(Fixed)

```
361     */
362     function initializeCollateral(
363         address asset,
364         address reinvestment,
365         uint256 feeMantissa,
366         uint256 ltv,
367         uint256 minDeposit
368     ) external onlyOperator {
369         require(collateralPoolList[asset][reinvestment] == 0, "already initialized");
370
371         collateralPoolLength++;
372         collateralPoolList[asset][reinvestment] = collateralPoolLength;
373
374         DataTypes.CollateralConfig storage collateral = _collateralConfig[collateralPoolLength];
375
376         configureCollateral(asset, reinvestment, feeMantissa, ltv, minDeposit, DataTypes.AssetState.Active);
377
378         collateral.poolId = collateralPoolLength;
379         collateral.asset = asset;
380         collateral.reinvestment = reinvestment;
381
382         emit InitializedCollateral(collateral.poolId, asset, reinvestment);
383     }
384
```

Figure 55 Source code of *initializeCollateral* function(Fixed)

[LeverFi-21] Admin permission not initialized

Severity Level	**Info**
Type	Business Security
Lines	Ledger.sol#L
Description	In the initialize function, there is no initial admin permission, which will result in the subsequent failure to change the permissions of OPERATOR_ROLE and LIQUIDATE_EXECUTOR.

```
117         */
118         function initialize(
119             address treasury_,
120             uint256 leverageFactor_,
121             uint256 liquidationRatioMantissa_,
122             uint256 tradeFeeMantissa_
123         ) public initializer {
124             treasury = treasury_;
125             leverageFactor = leverageFactor_;
126             liquidationRatioMantissa = liquidationRatioMantissa_;
127             tradeFeeMantissa = tradeFeeMantissa_;
128
129             _grantRole(OPERATOR_ROLE, msg.sender);
130             _grantRole(LIQUIDATE_EXECUTOR, msg.sender);
131         }
```

Figure 56 Source code of *initialize* function(Unfixed)

Recommendations	Initial admin privileges are recommended.
Status	Fixed.

```
119     function initialize(
120         address treasury_,
121         uint256 leverageFactor_,
122         uint256 liquidationRatioMantissa_,
123         uint256 tradeFeeMantissa_
124     ) public initializer {
125         require(leverageFactor_ >= 1e18, "leverage factor cannot be lower than 1x");
126         require(leverageFactor_ <= 10e18, "leverage factor cannot be lower than 10x");
127         require(liquidationRatioMantissa_ >= 0.5e18, "fee cannot be less than 50%");
128         require(liquidationRatioMantissa_ <= 0.9e18, "fee cannot be more than 90%");
129         require(tradeFeeMantissa_ <= 0.1e18, "fee cannot be more than 10%");
130
131         treasury = treasury_;
132         leverageFactor = leverageFactor_;
133         liquidationRatioMantissa = liquidationRatioMantissa_;
134         tradeFeeMantissa = tradeFeeMantissa_;
135
136         _setRoleAdmin(OPERATOR_ROLE, OPERATOR_ROLE);
137         _setRoleAdmin(LIQUIDATE_EXECUTOR, OPERATOR_ROLE);
138
139         _grantRole(OPERATOR_ROLE, msg.sender);
140         _grantRole(LIQUIDATE_EXECUTOR, msg.sender);
141     }
```

Figure 57 Source code of *initialize* function(Fixed)

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	High	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in Blockchain.

3.4 About BEOSIN

BEOSIN is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions.BEOSIN has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, BEOSIN has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website

https://www.beosin.com

Telegram

https://t.me/+dD8Bnqd133RmNWNl

Twitter

https://twitter.com/Beosin_com

Email

Contact@beosin.com