SMART CONTRACT AUDIT REPORT

for

LeverFi

Prepared By: Xiaomi Huang

PeckShield

September 13, 2022

Document Properties

Client	LeverFi
Title	Smart Contract Audit Report
Target	LeverFi
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 13, 2022	Jing Wang	Final Release
1.0-rc	August 30, 2022	Jing Wang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the LeverFi protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About LeverFi

LeverFi is a decentralized leveraged trading platforms which offers a leveraged solution for traders. With the LeverFi platform, traders could deposit collateral such as BTC, ETH, Curve-LP, Uni-LP and more to trade changes in asset prices at up to 10x leverage. Collaterals are deposited to farming protocols, allowing traders to earn yields and leverage trade at the same time. The basic information of the LeverFi protocol is as follows:

Table 1.1: Basic Information of The LeverFi Protocol

Item	Description
Issuer	LeverFi
Website	https://www.leverfi.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 13, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/LeverFi/leverfi-contracts (ed904fe)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/LeverFi/leverfi-contracts (5ab558f)

1.2 About PeckShield

PeckShield Inc. [15] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [14]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [13], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `LeverFi` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	3	🟦🟦🟦
High	0	
Medium	2	🟦🟦
Low	2	🟦🟦
Informational	0	
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 critical-severity vulnerabilities, 2 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key LeverFi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Possible Fund Stealing From Approving Users	Business Logic	Fixed
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Business Logics	Confirmed
PVE-004	Critical	Necessity of Single-Shot Initialization of AaveReinvestmentLogic	Time and State	Fixed
PVE-005	Low	Reentrancy Risk in Ledger	Time and State	Partially Fixed
PVE-006	Critical	Manipulated Utilized Supply For Inflated Reserve Balance	Time and State	Fixed
PVE-007	Medium	Exposure Of Permissioned Ledger::swapPosition()	Business Logic	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Possible Fund Stealing From Approving Users

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `ZeroexSwapAdapter`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

To facilitate the token swap when an user executes trade in the `LeverFi` protocol, the `ZeroexSwapAdapter` contract has a helper routine `swap()`. This routine is developed to transfer user funds into this contract and then use an external router to swap the funds.

To elaborate, we show below the `swap()` routine implementation. This routine allows the user to provide an arbitrary `swapBytesData` which is parsed into `approveRouter`, `sellingAsset`, `buyingAsset`, and `swapBytes`. These parameters are directly used in `approveRouter.call(swapBytes)` (line 24). However, the arbitrary `approveRouter` and `swapBytes` may be exploited to transfer all funds from the users to the hacker.

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

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

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

15        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);

        ...

    }
```

Listing 3.1: `ZeroexSwapAdapter::swap()`

Specifically, the `Ledger` contract needs to give allowance to the `ZeroexSwapAdapter` contract to facilitate the token swap. To save gas fee, the `ledger` gives a maximum allowance as `type(uint256).max` to the `ZeroexSwapAdapter` contract! In this case, if the `approveRouter` value is changed to the `token` address and the `swapBytes` is encoded to call function `token.transferFrom(ledgerAddress, hackerAddress, amount)` where `amount = token.balanceOf(ledgerAddress)`, all `token` of the `Ledger` contract would be withdrawn by the malicious actor.

```
    function executeTrade(
        mapping(address => DataTypes.ReserveConfig) storage reserveConfig,
        mapping(address => DataTypes.AssetConfig) storage assetConfig,
        mapping(address => DataTypes.UserPosition) storage userPosition,
        DataTypes.UserConfiguration storage userConfig,
        DataTypes.ExecuteSwapParams memory params
    ) external {
        ExecuteTradeVars memory vars;
        vars.assetConfigCache[0] = assetConfig[params.shortAsset];
        vars.assetConfigCache[1] = assetConfig[params.longAsset];
        ...
        if (
            IERC20Upgradeable(params.shortAsset).allowance(address(this), address(vars.
                assetConfigCache[0].swapAdapter)) < params.amount
        ) {
            IERC20Upgradeable(params.shortAsset).safeApprove(address(vars.
                assetConfigCache[0].swapAdapter), type(uint256).max);
        }
        vars.receivedAmount = vars.assetConfigCache[0].swapAdapter.swap(params.
            shortAsset, params.longAsset, params.amount, params.data);

        ...
    }
```

Listing 3.2: `TradeLogic::executeTrade()`

Recommendation Apply necessary rigorous validity checks on the untrusted user input. Also raise the community awareness of not giving token allowance to the `ZeroexSwapAdapter` contract.

Status This issue has been fixed in this commit: d90a0c1.

3.2 Accommodation of approve() Idiosyncrasies

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the approve() routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of approve(), there is a requirement, i.e., require(!((_value != 0) && (allowed[msg.sender][_spender] != 0))). This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling approve(_spender, 0)) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known approve()/ transferFrom() race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.3: USDT Token Contract

Because of that, a normal call to approve() with a currently non-zero allowance may fail. In the following, we use the Ledger::_depositReserve() routine as an example. This routine is designed to transfer tokens from depositor to Ledger contract. To accommodate the specific idiosyncrasy, for each `safeApprove()` (line 806), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
798     function _depositReserve(address user, address asset, uint256 amount) internal {
799         DataTypes.AssetConfig memory assetConfigCache = assetConfig[asset];
800         DataTypes.ReserveConfig storage reserve = _reserveConfig[asset];

802         ...

804         if (reserve.reinvestment != address(0)) {
805             if (IERC20Upgradeable(reserve.asset).allowance(address(this), reserve.
                    reinvestment) < amount) {
806                 IERC20Upgradeable(reserve.asset).safeApprove(reserve.reinvestment, type(
                        uint256).max);
807             }

809             IReinvestment(reserve.reinvestment).checkpoint(user, currUserReserveBalance)
                    ;
810             IReinvestment(reserve.reinvestment).invest(amount);
811         } else {
812             reserve.liquidSupply += amount;
813         }

815         emit DepositedReserve(user, asset, reserve.reinvestment, amount);
816     }
```

Listing 3.4: `Ledger::_depositReserve()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in this commit: `d90a0c1`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Ledger`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `LeverFi` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., system parameter setting and funds withdrawn in emergency). It also has the privilege to control or govern the flow of assets managed by this protocol.

Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function emergencyWithdrawReserve(address asset) external onlyOperator {


        DataTypes.ReserveConfig storage reserveConfig = _reserveConfig[asset];

        require(reserveConfig.state == DataTypes.AssetState.Disabled, "cannot withdraw
            with active pool");
        require(reserveConfig.reinvestment != address(0), "reinvestment is not set");

        uint256 withdrawn = IReinvestment(reserveConfig.reinvestment).emergencyWithdraw
            ();

        reserveConfig.liquidSupply = withdrawn;

        emit EmergencyWithdrawnReserve(asset, withdrawn);

    }
```

Listing 3.5: `Ledger::emergencyWithdrawReserve()`

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed.

3.4 Necessity of Single-Shot Initialization of AaveReinvestmentLogic

- ID: PVE-004
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: Multiple Contracts
- Category: Initialization and Cleanup [12]
- CWE subcategory: CWE-1188 [2]

Description

The LeverFi protocol's collateral and reserve are deposited to farming protocols, allowing traders to earn yields and leverage trade at the same time. The reinvestment logic are implemented via the AaveReinvestmentLogic contract, which has an initialize() function. This function is used to initialize a number of key parameters, including asset, addressStorage, addressStorage, treasury, ledger and feeMantissa. To facilitate our discussion, we show below the related code snippet.

```
    function initialize(
        address asset_,
        address receipt_,
        address platform_,
        address[] memory,
        address treasury_,
        address ledger_,
        uint256,
        bytes memory
    ) public {
        asset = asset_;
        addressStorage[RECEIPT] = receipt_;
        addressStorage[PLATFORM] = platform_;
        treasury = treasury_;
        ledger = ledger_;

        // Fees does not applied to this type of reinvestment (auto-accruing reward).
        feeMantissa = 0;
    }
```

Listing 3.6: AaveReinvestmentLogic::initialize()

Apparently the above logic does not provide the guarantee that the initialize() function can be called only once. What's more, it allows anyone to call the function! A bad actor could call initialize() and set the ledger to his own address, hence transferring all the tokens from the pool to his own wallet. Since multiple initializations could cause critical risk for the entire protocol, we suggest to ensure that the initialize() routine may only be called once.

```
    function divest(uint256 amount) external override onlyLedger {
        IAaveLendingPoolV2(platform()).withdraw(asset, amount, msg.sender);
    }

    modifier onlyLedger() {
        require(ledger == msg.sender, "only ledger");
        _;
    }
```

Listing 3.7: `AaveReinvestmentLogic::divest()`

Note another contract `ConvexReinvestmentLogic` shares the same issue.;

Recommendation Ensure the `initialize()` function could only be called once during the entire lifetime.

Status This issue has been fixed in the commit: 1d52714.

3.5 Reentrancy Risk in Ledger

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Ledger`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `Ledger` as an example, the `executeShorting()` function (see the code snippet below) is provided to externally call a reinvestment contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 169) starts before effecting the update on the internal state (lines 185-193), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
function executeShorting(
    DataTypes.ReserveConfig storage reserveConfig,
    DataTypes.AssetConfig memory assetConfigCache,
    DataTypes.UserPosition storage userAssetPosition,
    DataTypes.UserConfiguration storage userConfig,
    uint256 amount,
    bool doDivest
) public returns (uint256, uint256){
    ExecuteShortingVars memory vars;
    ...
    if (doDivest) {
        if (vars.amountLongToDivest > 0) {
            if (assetConfigCache.longReinvestment != address(0)) {
                IReinvestment(assetConfigCache.longReinvestment).divest(vars.
                    amountLongToDivest);
            }
        }
        if (vars.amountToShort > 0) {
            if (reserveConfig.reinvestment != address(0)) {
                IReinvestment(reserveConfig.reinvestment).divest(vars.amountToShort);
            } else {
                reserveConfig.liquidSupply -= vars.amountToShort;
            }
            reserveConfig.scaledUtilizedSupplyRay += vars.amountToShort.unitToRay(vars.
                unit).rayDiv(vars.reserveCache.currBorrowIndexRay);
        }
    }
    require(
        IERC20Upgradeable(reserveConfig.asset).balanceOf(address(this)) >= amount,
        "not enough gathered amount to proceed"
    );
    vars.newPosition = getNewPosition(
        assetConfigCache,
        userAssetPosition,
        DataTypes.UserPosition(amount, DataTypes.PositionType.Short),
        vars.reserveCache.currBorrowIndexRay
    );
    userAssetPosition.amount = vars.newPosition.amount;
    userAssetPosition._type = vars.newPosition._type;
    userConfig.setUsingPosition(reserveConfig.assetId, vars.newPosition.amount > 0);
    return (amount, vars.amountToShort);
}
```

Listing 3.8: `TradeLogic::executeShorting()`

Note this is a protocol level issue and other routines share the same issue.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been partially fixed by this commit: `1c44876`.

3.6 Manipulated Utilized Supply For Inflated Reserve Balance

- ID: PVE-006
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Ledger`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

As mentioned in Section 3.4, the `LeverFi` protocol's reserve are deposited to farming protocols. During the audit, we notice there is a logic error in the computation for reserve supply. It could be exploited to get more underlying tokens from the `Ledger` contract via simple deposit reserve and withdraw reserve operations. To elaborate, we list the related routines, including `getReserveSupply()`, `_depositReserve()` and `_withdrawReserve()` when user performs the deposit and withdrawal.

```
    function getReserveSupply(
        DataTypes.ReserveConfig storage reserve,
        DataTypes.AssetConfig memory assetConfig,
        bool claimable
    ) internal view returns (uint256 poolAmount) {
        if (!claimable) {


            poolAmount += getUtilizedSupply(reserve, assetConfig);
        }


        poolAmount += GeneralLogic.getReserveAvailableSupply(reserve);
    }


    function _depositReserve(address user, address asset, uint256 amount) internal {

        ...

        uint256 shareAmountRay = ShareBaseAccounting.getShareAmount(
            amount.unitToRay(unit),
            reserve.scaledTotalSupplyRay,
            reserve.getReserveSupply(assetConfigCache, true).unitToRay(unit)
        );

        ...

    }

    function _withdrawReserve(address user, address asset, uint256 amount) internal {

        uint256 currReserveSupply = reserveConfig.getReserveSupply(assetConfigCache,
            false);
        ...

        scaledAmountToWithdrawRay = ShareBaseAccounting.getShareAmount(
            amount.unitToRay(unit),
            currScaledTotalSupplyRay,
            currReserveSupply.unitToRay(unit)
        );

        ...
    }
```

Listing 3.9: `getReserveSupply::getReserveSupply()/Ledger::_depositReserve()/Ledger::_withdrawReserve()`

In particular, the share amount of the user reserve is calculated via the amount of deposited underlying tokens multiples `scaledTotalSupplyRay` and then divides `currReserveSupply`. However, `currReserveSupply` is calculated via `getReserveAvailableSupply()`, which only counts the underlying utilized supply on withdrawal, not deposit. In other words, the value of the `currReserveSupply` is inflated when user doing a withdrawal. A bad actor could artificially manipulate the utilized supply to a big number to drain all the funds in the pool.

Recommendation Make sure the deposit and withdrawal operations using the same reserve supply to compute the share.

Status This issue has been fixed in the following commit: `5ab558f`.

3.7 Exposure Of Permissioned Ledger::swapPosition()

- ID: PVE-007
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Ledger`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `LeverFi` protocol has a liquidator system in place to margin call traders if trading losses approach the value of collateral deposited. When examining the implementation of the liquidation system, we notice the presence of a specific routine, i.e., `swapPosition()`. As the name indicates, this routine is used to swap a position for the liquidation wallet. To elaborate, we show below the code snippet of this function.

```
  function swapPosition(address assetIn, address assetOut, uint256 amount, bytes memory
      data) external {
    TradeLogic.executeTrade(
        _reserveConfig,
        assetConfig,
        _userPosition[LIQUIDATION_WALLET],
        userConfig[LIQUIDATION_WALLET],
        DataTypes.ExecuteSwapParams(
            LIQUIDATION_WALLET,
            treasury,
            assetIn,
            assetOut,
            amount,
            tradeFeeMantissa,
            0,
            0,
            0,
            data,
            false,
            false
        )
    );
}
```

Listing 3.10: `Ledger::swapPosition()`

However, we notice that this routine is currently permissionless, which means it can be invoked by anyone to update change the position to another one according to his wish. A bad actor could make profit by decreasing the `PNL` of the liquidation wallet while satisfy the requirement of `int256(params.totalCollateralUsdPreLtv.wadMul(params.liquidationRatioMantissa))+ params.pnlUsd)>= 0`.

Recommendation Adjust the modifier to constrain the operation of `swapPosition()` for the liquidation wallet.

Status This issue has been fixed in the following commit: `826f589`.

4 Conclusion

In this audit, we have analyzed the `LeverFi` protocol design and implementation. `LeverFi` is a decentralized leveraged trading platforms which offers a leveraged solution for traders. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[11] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[12] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[13] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[14] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[15] PeckShield. PeckShield Inc. https://www.peckshield.com.

[16] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[17] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.