SMART CONTRACT AUDIT REPORT

for

Over-collateralized Borrowing Protocol

Prepared By: Xiaomi Huang

PeckShield

February 15, 2023

Document Properties

Client	OpenLeverage
Title	Smart Contract Audit Report
Target	Over-collateralized Borrowing Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Final

Version Info

Version	Date	Author(s)	Description
1.0	February 15, 2023	Xuxian Jiang	Final Release
1.0-rc1	February 11, 2023	Xuxian Jiang	Release Candidate #1

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 `Over-collateralized Borrowing` 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 Over-collateralized Borrowing Protocol

The `OpenLeverage` protocol is a permissionless margin trading protocol that enables traders or other applications to be long or short on any trading pair on DEXs efficiently and securely. In particular, it enables margin trading with liquidity on various DEXs, hence connecting traders to trade with the most liquid decentralized markets. It is also designed to have two separated pools for each pair with different risk and interest rate parameters, allowing lenders to invest according to the risk-reward ratio. This audit covers a new over-collateralized borrowing protocol that is integrated with `OpenLeverage` and enables borrowers to collateralize and borrow any pair on DEXs efficiently and securely. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Over-collateralized Borrowing Protocol

Item	Description
Name	OpenLeverage
Website	https://openleverage.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 15, 2023

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

- https://github.com/OpenLeverageDev/overcollateralized-borrowing-contracts.git (af45bb2)

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

- https://github.com/OpenLeverageDev/overcollateralized-borrowing-contracts.git (f7da719)

1.2 About PeckShield

PeckShield Inc. [12] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [11]:

- 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 checklist of 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 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) [10], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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 Logic	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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `over-collateralized borrowing` protocol. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	1	■
Total	5	

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 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 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Over-collateralized Borrowing Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Allowance Management in OPBorrowing	Coding Practices	Resolved
PVE-002	Informational	Possible Unreliable Collateral Ratio Calculation in OPBorrowing	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Resolved
PVE-004	Low	Improved Sanity Checks on Protocol Parameters	Coding Practices	Resolved
PVE-005	Low	Possible Insurance Reduction in Liquidation	Time and State	Resolved

Besides the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Proper Allowance Management in OPBorrowing

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OPBorrowing`
- Category: Business Logic [8]
- CWE subcategory: CWE-770 [4]

Description

The `OpenLeverage` protocol supports permissionless margin trading markets. The integrated over-collateralized borrowing protocol enables borrowers to collateralize and borrow any pair on DEXs efficiently. Note an underwater borrow position may be liquidated and certain portion of collected liquidation fee may be sent to buyback. While examining the related liquidation fee logic, we notice its implementation can be improved.

To elaborate, we show below the related routine `collectLiquidationFee()`. As the name indicates, the routine is used to collect liquidation fee. And certain portion or the `buyBackAmount` will be sent to the `liquidationConf.buyBack` (lines 550-555). It comes to our attention there is an earlier allowance granted to the `liquidationConf.buyBack` contract. However, there is no closing allowance reset after the buyback!

```
525     function collectLiquidationFee(
526         uint16 marketId,
527         bool collateralIndex,
528         uint liquidationFees,
529         address borrowToken,
530         LPoolInterface borrowPool,
531         uint borrowTotalReserve,
532         uint borrowTotalShare
533     ) internal returns (bool buyBackSuccess) {
534         if (liquidationFees > 0) {
535             MarketConf storage marketConf = marketsConf[marketId];
```

```
            uint poolReturns = (liquidationFees * marketConf.liquidatePoolReturnsRatio) / RATIO_DENOMINATOR;
            if (poolReturns > 0) {
                OPBorrowingLib.safeTransfer(IERC20(borrowToken), address(borrowPool), poolReturns);
            }
            uint insurance = (liquidationFees * marketConf.liquidateInsuranceRatio) / RATIO_DENOMINATOR;
            if (insurance > 0) {
                uint increaseInsurance = OPBorrowingLib.amountToShare(insurance, borrowTotalShare, borrowTotalReserve);
                increaseInsuranceShare(insurances[marketId], !collateralIndex, borrowToken, increaseInsurance);
            }
            uint liquidatorReturns = (liquidationFees * marketConf.liquidatorReturnsRatio) / RATIO_DENOMINATOR;
            if (liquidatorReturns > 0) {
                OPBorrowingLib.safeTransfer(IERC20(borrowToken), msg.sender, liquidatorReturns);
            }
            uint buyBackAmount = liquidationFees - poolReturns - insurance - liquidatorReturns;
            if (buyBackAmount > 0) {
                OPBorrowingLib.safeApprove(IERC20(borrowToken), address(liquidationConf.buyBack), buyBackAmount);
                (buyBackSuccess, ) = address(liquidationConf.buyBack).call(
                    abi.encodeWithSelector(liquidationConf.buyBack.transferIn.selector, borrowToken, buyBackAmount)
                );
            }
        }
    }
```

Listing 3.1: `OPBorrowing::collectLiquidationFee()`

Recommendation Revisit the above `collectLiquidationFee()` logic to properly reset the spend allowance after the buyback.

Status The issue has been fixed by this commit: `f7da719`.

3.2 Possible Unreliable Collateral Ratio Calculation in OPBorrowing

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `OPBorrowing`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The audited protocol has a core `OPBorrowing` contract that exports a specific `collateralRatio()` function. This function allows to query the given borrower's collateral ratio. While analyzing its implementation, we notice it is prone to manipulation and requires caution in assessing the returned ratio.

Specifically, if we examine the `collateralRatio()` implementation, it makes use of the spot price of the associated pair from the specified `dexAgg`. The spot price may be volatile and can be easily manipulated from possible flashloans. Therefore, we suggest to exercise caution in using this exported function.

```
    function collateralRatio(uint16 marketId, bool collateralIndex, address borrower)
        external view override returns (uint) {
        BorrowVars memory borrowVars = toBorrowVars(marketId, collateralIndex);
        uint borrowed = borrowVars.borrowPool.borrowBalanceCurrent(borrower);
        uint collateral = activeCollaterals[borrower][marketId][collateralIndex];
        if (borrowed == 0  collateral == 0) {
            return 100 * RATIO_DENOMINATOR;
        }
        uint collateralAmount = OPBorrowingLib.shareToAmount(collateral, borrowVars.
            collateralTotalShare, borrowVars.collateralTotalReserve);
        MarketConf storage marketConf = marketsConf[marketId];
        bytes memory dexData = OPBorrowingLib.uint32ToBytes(markets[marketId].dex);
        (uint price, uint8 decimals) = dexAgg.getPrice(borrowVars.collateralToken,
            borrowVars.borrowToken, dexData);
        return (((collateralAmount * price) / (10 ** uint(decimals))) * marketConf.
            collateralRatio) / borrowed;
    }
```

Listing 3.2: `OPBorrowing::collateralRatio()`

Recommendation Be aware of the implicit assumption or associated risk when calling the above `collateralRatio()` function.

Status This issue has been resolved as the team clarifies that the above function is used only for front-end inquiries, not external integration with other protocols.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the audited protocol, there is a privileged `admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and pool adjustment). 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 migrateOpenLevMarkets(uint16 from, uint16 to) external override onlyAdmin {
        for (uint16 i = from; i <= to; i++) {
            OpenLevInterface.Market memory market = openLev.markets(i);
            addMarketInternal(
                i,
                LPoolInterface(market.pool0),
                LPoolInterface(market.pool1),
                market.token0,
                market.token1,
                OPBorrowingLib.uint32ToBytes(openLev.getMarketSupportDexs(i)[0])
            );
        }
    }

    function setTwaLiquidity(uint16[] calldata marketIds, OPBorrowingStorage.Liquidity[]
         calldata liquidity) external override onlyAdminOrDeveloper {
        require(marketIds.length == liquidity.length, "IIL");
        for (uint i = 0; i < marketIds.length; i++) {
            uint16 marketId = marketIds[i];
            setTwaLiquidityInternal(marketId, liquidity[i].token0Liq, liquidity[i].
                token1Liq);
        }
    }

    function setMarketConf(uint16 marketId, OPBorrowingStorage.MarketConf calldata
        _marketConf) external override onlyAdmin {
        marketsConf[marketId] = _marketConf;
        emit NewMarketConf(
            marketId,
            _marketConf.collateralRatio,
            _marketConf.maxLiquidityRatio,
            _marketConf.borrowFeesRatio,
```

```
            _marketConf.insuranceRatio,
            _marketConf.poolReturnsRatio,
            _marketConf.liquidateFeesRatio,
            _marketConf.liquidatorReturnsRatio,
            _marketConf.liquidateInsuranceRatio,
            _marketConf.liquidatePoolReturnsRatio,
            _marketConf.liquidateMaxLiquidityRatio,
            _marketConf.twapDuration
        );
    }
```

Listing 3.3: Example `Setters` in the `OPBorrowing` Contract

In addition, we notice the `admin` account that is able to add new markets and configure various liquidity. Apparently, if the privileged `admin` 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 that current contracts have the support of being deployed behind a proxy. And 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 with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.4 Improved Sanity Checks on Protocol Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OPBorrowing`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `audited` protocol is no exception. Specifically, if we examine the `OPBorrowing` contract, it has defined a number of protocol-wide risk parameters, such as `liquidatorReturnsRatio` and `liquidateInsuranceRatio`. In the following, we show the corresponding routines that allow for their changes.

```
    function setMarketConf(uint16 marketId, OPBorrowingStorage.MarketConf calldata
        _marketConf) external override onlyAdmin {
        marketsConf[marketId] = _marketConf;
        emit NewMarketConf(
            marketId,
            _marketConf.collateralRatio,
            _marketConf.maxLiquidityRatio,
            _marketConf.borrowFeesRatio,
            _marketConf.insuranceRatio,
            _marketConf.poolReturnsRatio,
            _marketConf.liquidateFeesRatio,
            _marketConf.liquidatorReturnsRatio,
            _marketConf.liquidateInsuranceRatio,
            _marketConf.liquidatePoolReturnsRatio,
            _marketConf.liquidateMaxLiquidityRatio,
            _marketConf.twapDuration
        );
    }
```

Listing 3.4: OPBorrowing::setMarketConf()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of various fee parameters `_fee` may charge unreasonably high fee in the `borrow/repayment` operations, hence incurring cost to borrowers or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `f7da719`.

3.5 Possible Insurance Reduction in Liquidation

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OPBorrowing`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the over-collateralized borrowing protocol may liquidate under-collateralized borrow positions. By design, the protocol reserves a portion of accrued liquidation fee as the insurance to cover potential loss in the liquidation process. While examining the liquidation feature, we identify a potential flashloan-assisted scenario to consume the insurance funds.

```
        {
            uint collateralLiquidity = dexAgg.getToken0Liquidity(borrowVars.collateralToken, borrowVars.borrowToken, liquidateVars.dexData);
            uint maxLiquidity = (collateralLiquidity * marketConf.liquidateMaxLiquidityRatio) / RATIO_DENOMINATOR;
            if (liquidateVars.liquidationAmount >= maxLiquidity) {
                liquidateVars.liquidationShare = liquidateVars.liquidationShare / 2;
                liquidateVars.liquidationAmount = OPBorrowingLib.shareToAmount(
                    liquidateVars.liquidationShare,
                    borrowVars.collateralTotalShare,
                    borrowVars.collateralTotalReserve
                );
                liquidateVars.isPartialLiquidate = true;
            }
        }
        (liquidateVars.price0, ) = dexAgg.getPrice(markets[marketId].token0, markets[marketId].token1, liquidateVars.dexData);
        // compute sell collateral amount, borrowings + liquidationFees + tax
        {
            uint24 borrowTokenTransTax = openLev.taxes(marketId, borrowVars.borrowToken, 0);
            uint24 borrowTokenBuyTax = openLev.taxes(marketId, borrowVars.borrowToken, 2);
            uint24 collateralSellTax = openLev.taxes(marketId, borrowVars.collateralToken, 1);

            liquidateVars.repayAmount = Utils.toAmountBeforeTax(liquidateVars.borrowing, borrowTokenTransTax);
            liquidateVars.liquidationFees = (liquidateVars.borrowing * marketConf.liquidateFeesRatio) / RATIO_DENOMINATOR;
```

```
            OPBorrowingLib.safeApprove(IERC20(borrowVars.collateralToken), address(
                dexAgg), liquidateVars.liquidationAmount);
            (liquidateVars.buySuccess, ) = address(dexAgg).call(
                abi.encodeWithSelector(
                    dexAgg.buy.selector,
                    borrowVars.borrowToken,
                    borrowVars.collateralToken,
                    borrowTokenBuyTax,
                    collateralSellTax,
                    liquidateVars.repayAmount + liquidateVars.liquidationFees,
                    liquidateVars.liquidationAmount,
                    liquidateVars.dexData
                )
            );
        }
        /*
         * if buySuccess==true, repay all debts and returns collateral
         */
        if (liquidateVars.buySuccess) {
            uint sellAmount = borrowVars.collateralTotalReserve - OPBorrowingLib.
                balanceOf(IERC20(borrowVars.collateralToken));
            liquidateVars.collateralToBorrower = liquidateVars.collateralAmount -
                sellAmount;
            liquidateVars.buyAmount = OPBorrowingLib.balanceOf(IERC20(borrowVars.
                borrowToken)) - borrowVars.borrowTotalReserve;
            require(liquidateVars.buyAmount >= liquidateVars.repayAmount, "BLR");
            OPBorrowingLib.repay(borrowVars.borrowPool, borrower, liquidateVars.
                repayAmount);
            // check borrowing is 0
            require(OPBorrowingLib.borrowStored(borrowVars.borrowPool, borrower) == 0, "
                BGO");
            unchecked {
                liquidateVars.liquidationFees = liquidateVars.buyAmount - liquidateVars.
                    repayAmount;
            }
            liquidateVars.liquidationShare = collateral;
        }
        /*
         * if buySuccess==false and isPartialLiquidate==true, sell liquidation amount
            and repay with buyAmount
         * if buySuccess==false and isPartialLiquidate==false, sell liquidation amount
            and repay with buyAmount + insurance
         */
        else {
            liquidateVars.buyAmount = dexAgg.sell(
                borrowVars.borrowToken,
                borrowVars.collateralToken,
                liquidateVars.liquidationAmount,
                0,
                liquidateVars.dexData
            );
```

```
304             liquidateVars.liquidationFees = (liquidateVars.buyAmount * marketConf.liquidateFeesRatio) / RATIO_DENOMINATOR;
305             if (liquidateVars.isPartialLiquidate) {
306                 liquidateVars.repayAmount = liquidateVars.buyAmount - liquidateVars.liquidationFees;
307                 OPBorrowingLib.repay(borrowVars.borrowPool, borrower, liquidateVars.repayAmount);
308                 require(OPBorrowingLib.borrowStored(borrowVars.borrowPool, borrower) != 0, "BEO");
309             } else {
310                 uint insuranceShare = collateralIndex ? insurances[marketId].insurance0 : insurances[marketId].insurance1;
311                 uint insuranceAmount = OPBorrowingLib.shareToAmount(insuranceShare, borrowVars.borrowTotalShare, borrowVars.borrowTotalReserve);
312                 uint diffRepayAmount = liquidateVars.repayAmount + liquidateVars.liquidationFees - liquidateVars.buyAmount;
313                 uint insuranceDecrease;
314                 if (insuranceAmount >= diffRepayAmount) {
315                     OPBorrowingLib.repay(borrowVars.borrowPool, borrower, liquidateVars.repayAmount);
316                     insuranceDecrease = OPBorrowingLib.amountToShare(diffRepayAmount, borrowVars.borrowTotalShare, borrowVars.borrowTotalReserve);
317                 } else {
318                     liquidateVars.repayAmount = liquidateVars.buyAmount + insuranceAmount - liquidateVars.liquidationFees;
319                     borrowVars.borrowPool.repayBorrowEndByOpenLev(borrower, liquidateVars.repayAmount);
320                     liquidateVars.outstandingAmount = diffRepayAmount - insuranceAmount;
321                     insuranceDecrease = insuranceShare;
322                 }
323                 decreaseInsuranceShare(insurances[marketId], !collateralIndex, borrowVars.borrowToken, insuranceDecrease);
324             }
325         }
```

Listing 3.5: `OPBorrowing::liquidate()`

To elaborate, we show above the code snippet from the `liquidate()` routine. It implements the intended logic by taking real-time pricing from the on-chain AMM model as a reference and utilizing it in risk calculation and liquidation. Unfortunately, a flashloan-assisted manipulation may make the on-chain pricing information highly skewed. As a result, the liquidation logic can be "guided" into stealing the insurance fund. In particular, the skewed DEX pricing influences the execution flow such that all held collaterals need to be sold (line 297), and the returned borrow token amount is still insufficient to pay `repayAmount` and `liquidationFees`, hence taking the execution path (line 317). After that, the insurance funds will be used for payment.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above flashloan attack to better protect the interests of protocol users.

Status This issue has been resolved with the built-in price inflation detection as well as the borrow cap enforcement.

4 Conclusion

In this audit, we have analyzed the design document and related smart contract source code of the `Over-collateralized Borrowing` protocol. It is a permissionless margin trading protocol that allows traders or other applications to be long or short on any trading pair on DEXs efficiently and securely. This audit covers a new over-collateralized borrowing protocol that is permissionless and enables borrowers to collateralize and borrow any pair on DEXs efficiently and securely. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[4] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

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

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

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

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