SMART CONTRACT AUDIT REPORT

for

OpenLeverage Limit Order

Prepared By: Xiaomi Huang

PeckShield

September 10, 2022

Document Properties

Client	OpenLeverage
Title	Smart Contract Audit Report
Target	OpenLeverage Limit Order
Version	1.0
Author	Xuxian Jiang
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 10, 2022	Xuxian Jiang	Final Release
1.0-rc	September 5, 2022	Xuxian Jiang	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 `Limit Order` support in `OpenLeverage`, 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 OpenLeverage

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 the much-needed limit order support. The basic information of the audited smart contracts is as follows:

Table 1.1: Basic Information of OpenLeverage Limit Order

Item	Description
Issuer	OpenLeverage
Website	https://openleverage.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 10, 2022

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

this audit.

- https://github.com/OpenLeverageDev/open-order-contracts.git (e23b839)

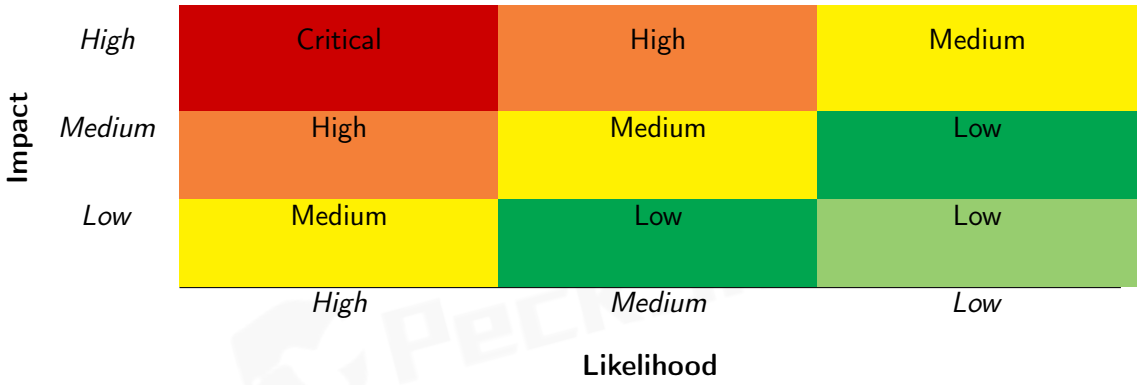
And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/OpenLeverageDev/open-order-contracts.git (d3178edb)

1.2 About PeckShield

PeckShield Inc. [9] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- 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.

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
	Algem 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

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) [7], 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 1.4: 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 `Limit Order` support in `OpenLeverage`. 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	1	■
Medium	1	■
Low	1	■
Total	3	

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 high-severity vulnerability, 1 medium-severity vulnerability, and 1 low-severity vulnerability.

Table 2.1: Key OpenLeverage Limit Order Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Possible Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-002	High	Possible Signature Malleability in OpenZeppelin ECDSA	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Resolved

Beside 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 Accommodation of Possible Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OPLimitOrder`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 `transferFrom()` routine and 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. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
171     function transferFrom(address _from, address _to, uint _value) public
            onlyPayloadSize(3 * 32) {
172         var _allowance = allowed[_from][msg.sender];

174         // Check is not needed because sub(_allowance, _value) will already throw if
                this condition is not met
175         // if (_value > _allowance) throw;

177         uint fee = (_value.mul(basisPointsRate)).div(10000);
178         if (fee > maximumFee) {
179             fee = maximumFee;
180         }
181         if (_allowance < MAX_UINT) {
182             allowed[_from][msg.sender] = _allowance.sub(_value);
183         }
```

```
184         uint sendAmount = _value.sub(fee);
185         balances[_from] = balances[_from].sub(_value);
186         balances[_to] = balances[_to].add(sendAmount);
187         if (fee > 0) {
188             balances[owner] = balances[owner].add(fee);
189             Transfer(_from, owner, fee);
190         }
191         Transfer(_from, _to, sendAmount);
192     }
```

Listing 3.1: USDT Token **Contract**

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()`/`transfer()` as well, i.e., `safeApprove()`/ `safeTransfer()`.

In current implementation, if we examine the `OPLimitOrder::fillOpenOrder()` routine that is designed to fill a given open order, to accommodate the specific idiosyncrasy, there is a need to use `safeTransferFrom()`, instead of `transferFrom()` (lines 69 and 78).

```
47     function fillOpenOrder(OpenOrder memory order, bytes calldata signature, uint256
           fillingDeposit, bytes memory dexData) external override nonReentrant {
48         require(block.timestamp <= order.deadline, 'EXR');
49         bytes32 orderId = _openOrderId(order);
50         uint256 remainingDeposit = _remaining[orderId];
51         require(remainingDeposit != _ORDER_FILLED, "RDO");
52         if (remainingDeposit == _ORDER_DOES_NOT_EXIST) {
53             remainingDeposit = order.deposit;
54         } else {
55             remainingDeposit -= 1;
56         }
57         require(fillingDeposit <= remainingDeposit, 'FTB');
58         require(SignatureChecker.isValidSignatureNow(order.owner, _hashOpenOrder(order),
               signature), "SNE");

60         uint256 fillingRatio = fillingDeposit * MILLION / order.deposit;
61         require(fillingRatio > 0, 'FRO');

63         OpenLevInterface.Market memory market = openLev.markets(order.marketId);
64         // long token0 price lower than price0 or long token1 price higher than price0
65         uint256 price = _getPrice(market.token0, market.token1, dexData);
66         require((!order.longToken && price <= order.price0)  (order.longToken && price
               >= order.price0), 'PRE');

68         address depositToken = order.depositToken ? market.token1 : market.token0;
69         IERC20(depositToken).transferFrom(order.owner, address(this), fillingDeposit);
70         IERC20(depositToken).safeApprove(address(openLev), fillingDeposit);
```

```
72          uint increaseHeld = _marginTrade(order, fillingRatio, dexData);
73          // check that increased position held is greater than expect held
74          require(increaseHeld * MILLION >= order.expectHeld * fillingRatio, 'NEG');

76          uint commission = order.commission * fillingRatio / MILLION;
77          if (commission > 0) {
78              IERC20(order.commissionToken).transferFrom(order.owner, msg.sender,
                    commission);
79          }
80          remainingDeposit = remainingDeposit - fillingDeposit;
81          emit OrderFilled(msg.sender, orderId, commission, fillingDeposit,
                remainingDeposit);
82          _remaining[orderId] = remainingDeposit + 1;
83      }
```

Listing 3.2: `OPLimitOrder::fillOpenOrder()`

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

Status This issue has been fixed in the commit: 173a1ac.

3.2 Possible Signature Malleability in OpenZeppelin ECDSA

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `OPLimitOrder`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `OPLimitOrder` contract relies on a number of library contracts to facilitate its functionality and organization. One specific one is the popular `OpenZeppelin`, which is an open-source framework to build secure smart contracts. It comes to our attention the imported `@openzeppelin/contracts` has the `4.7.0` version, which has a so-called `ECDSA signature malleability` issue on its signature verfication logic.

To elaborate, we show below the affected function `fillOpenOrder()`, which is designed to fill an order order and makes use of `SignatureChecker.isValidSignatureNow()` to validate the given signature. Specifically, it checks whether a signature is valid for a given signer and data hash. If the signer is a smart contract, the signature is validated against that smart contract using ERC1271, otherwise it's validated using `ECDSA.tryRecover()`.

```
47      function fillOpenOrder(OpenOrder memory order, bytes calldata signature, uint256
            fillingDeposit, bytes memory dexData) external override nonReentrant {
48          require(block.timestamp <= order.deadline, 'EXR');
49          bytes32 orderId = _openOrderId(order);
50          uint256 remainingDeposit = _remaining[orderId];
51          require(remainingDeposit != _ORDER_FILLED, "RDO");
52          if (remainingDeposit == _ORDER_DOES_NOT_EXIST) {
53              remainingDeposit = order.deposit;
54          } else {
55              remainingDeposit -= 1;
56          }
57          require(fillingDeposit <= remainingDeposit, 'FTB');
58          require(SignatureChecker.isValidSignatureNow(order.owner, _hashOpenOrder(order),
                signature), "SNE");
59
60          ...
61      }
62
63      function isValidSignatureNow(
64          address signer,
65          bytes32 hash,
66          bytes memory signature
67      ) internal view returns (bool) {
68          (address recovered, ECDSA.RecoverError error) = ECDSA.tryRecover(hash, signature
                );
69          if (error == ECDSA.RecoverError.NoError && recovered == signer) {
70              return true;
71          }
72
73          (bool success, bytes memory result) = signer.staticcall(
74              abi.encodeWithSelector(IERC1271.isValidSignature.selector, hash, signature)
75          );
76          return (success && result.length == 32 && abi.decode(result, (bytes4)) ==
                IERC1271.isValidSignature.selector);
77      }
```

Listing 3.3: `OPLimitOrder::fillOpenOrder()`

However, the current version of `tryRecover()` is vulnerable to a kind of signature malleability "*due to accepting EIP-2098 compact signatures in addition to the traditional 65 byte signature format.*" Note that this is only an issue for the functions that take a single bytes argument, and not the functions that take r, v, s or r, vs as separate arguments.

Recommendation Upgrade imported library of `@openzeppelin/contracts` to the latest version (>=4.7.3) to resolve the possible signature malleability issue.

Status This issue has been fixed in the commit: 173a1ac.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: OPLimitOrder
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the limit order support of OpenLeverage, there is a special administrative account, i.e., admin. This admin account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration). 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 admin account and its related privileged accesses in current contract.

To elaborate, we show the initialize() routine from the OPLimitOrder contract. This function allows the admin account to repeatedly update the critical parameters on openLev and dexAgg.

```
32    function initialize(OpenLevInterface _openLev, DexAggregatorInterface _dexAgg)
          external {
33        require(msg.sender == admin, "NAD");
34        openLev = _openLev;
35        dexAgg = _dexAgg;
36    }
```

Listing 3.4: OPLimitOrder::initialize()

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged admin account is a plain EOA account. 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.

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. The team clarifies the admin key is a timelock contract, not an EOA address.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Limit Order` support in `OpenLeverage`, which 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. This audit covers the much-needed limit order support. Our analysis shows the current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Meanwhile, 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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

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

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