SMART CONTRACT AUDIT REPORT

for

PERPETUAL-LIMIT-ORDERS

Prepared By: Shuxiao Wang

PeckShield

May 18, 2021

Document Properties

Client	APEX
Title	Smart Contract Audit Report
Target	Perpetual-Limit-Orders
Version	1.0
Author	Xuxian Jiang
Auditors	Jian Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 18, 2021	Xuxian Jiang	Final Release
1.0-rc	May 18, 2021	Xuxian Jiang	Release Candidate
0.2	May 11, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 3, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Perpetual-Limit-Orders Protocol** design document and related smart contract source code, 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 is well-designed. This document outlines our audit results.

1.1 About APEX Protocol

`Advanced Perpetual Exchange (APEX)` is built upon the `Perpetual Protocol (PP)`. It leverages the decentralized perpetual futures trading protocol built by the `PP` team and adds additional functionality, such as the ability to create limit orders and other advanced order types. The aim of `APEX` is to create a decentralized and trustless system that is fully non-custodial. While other AMM based derivative DEXes rely on a trusted set up to execute limit orders, the `APEX` smart contracts ensure the timely execution of orders through a permissionless network of `keeper-bots`. This means that at no point does `APEX-PP` have access to your funds, nor does it depend on trusted third parties.

The basic information of the Perpetual-Limit-Orders protocol is as follows:

Table 1.1: Basic Information of Perpetual-Limit-Orders

Item	Description
Name	APEX
Website	https://www.apex.win/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 18, 2021

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

- https://github.com/perpfutui/perpetual-limit-orders.git (b258574)

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

- https://github.com/perpfutui/perpetual-limit-orders.git (ebaa724)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 [6]:

- 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 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

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 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) [5], 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), 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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Perpetual-Limit-Orders protocol design and 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	0	
High	0	
Medium	0	
Low	4	🟦🟦🟦🟦
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 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Perpetual-Limit-Orders Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Cleanup Upon Order Deletion	Code Practices	Fixed
PVE-002	Low	Improved Validation Checks in onlyValidOrder()	Code Practices	Fixed
PVE-003	Low	Improved Decimal Operation on modD()	Code Practices	Fixed
PVE-004	Informational	Improved Logic Of SmartWallet::executeCall()	Code Practices	Fixed
PVE-005	Low	Accommodation of approve() Idiosyncrasies	Time and State	Fixed

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 Proper Cleanup Upon Order Deletion

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LimitOrderBook
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate the limit order and other advanced ones, the Perpetual-Limit-Orders protocol organizes orders with two different data structures LimitOrder and TrailingOrderData. In addition, the protocol defines a number of entry functions (e.g., addLimitOrder(), addStopOrder(), addStopLimitOrder(), addTrailingStopMarketOrderAbs(), addTrailingStopMarketOrderPct(), and deleteOrder()) for users to interact with. In the following, we examine the deleteOrder() logic.

To elaborate, we show below the implementation of the deleteOrder() routine. As the name indicates, this routine is designed to delete the intended order. However, the current logic only deletes the related entry in orders, but not trailingOrders. An improvement can be made to check whether there is a related entry in trailingOrders and remove it if present.

```
/*
 * @notice Delete an order
 */
function deleteOrder(
  uint order_id
) external onlyMyOrder(order_id) onlyValidOrder(order_id){
  LimitOrder memory order = orders[order_id];
  if((order.orderType == OrderType.TRAILINGSTOPMARKET ||
      order.orderType == OrderType.TRAILINGSTOPLIMIT)) {
        emit TrailingOrderCancelled(order_id);
        delete trailingOrders[order_id];
    }
  emit OrderCancelled(order.trader, order_id);
```

```
    delete orders[order_id];
  }
```

Listing 3.1: LimitOrderBook::deleteOrder()

Recommendation Revise the above `deleteOrder()` logic to properly remove the related entry in `trailingOrders` if present.

Status This issue has been fixed in this commit: `0a8a7ce`.

3.2 Improved Validation Checks in onlyValidOrder()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LimitOrderBook`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.1, the Perpetual-Limit-Orders protocol defines a number of entry functions for users to interact with. In the following, we examine a specific `onlyValidOrder()` modifier that is being used by a number of entry functions to validate a given order.

To elaborate, we show below the implementation of the `onlyValidOrder()` modifier. Our analysis shows that `onlyValidOrder()` can be strengthened by further ensuring `require(order.stillValid, 'No longer valid')` so that the executed order can be safely filtered out.

```
  /*
   * MODIFIERS
   */

  modifier onlyValidOrder(uint order_id) {
    require(order_id < orders.length, 'Invalid ID');
    LimitOrder memory order = orders[order_id];
    require(order.stillValid, 'No longer valid');
    require(((order.expiry == 0 ) (block.timestamp<order.expiry)), 'Order expired');
    _;
  }
```

Listing 3.2: LimitOrderBook::onlyValidOrder()

Recommendation Revised the `onlyValidOrder()` modifier to remove executed orders.

Status This issue has been fixed in this commit: `620d6f2`.

3.3 Improved Decimal Operation on modD()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Decimal
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine the current library routines that support a number of arithmetic operations, e.g., addD(), subD(), addD(), mulD() divD(), and modD().

In particular, we use the Decimal::modD() as an example. This routine is used to calculate the remainder from the given two numbers. However, its current implementation improperly scales up the remainder by 2^{18}. Fortunately, this library function is not used in current codebase.

```
    function modD(decimal memory x, decimal memory y) internal pure returns (decimal
        memory) {
        return decimal(x.d.mul(DecimalMath.unit(18)) % y.d);
    }
```

Listing 3.3: Decimal::modD()

Recommendation Properly revise the aforementioned modD() implementation. An example revision is shown below:

```
    function modD(decimal memory x, decimal memory y) internal pure returns (decimal
        memory) {
        return decimal(x.d % y.d);
    }
```

Listing 3.4: Revised Decimal::modD()

Status This issue has been fixed in this commit: ebaa724.

3.4 Improved Logic Of SmartWallet::executeCall()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SmartWallet`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The Perpetual-Limit-Orders protocol comes with a convenient smart wallet contract, which is a hybrid between a smart contract and a personal wallet (hence the name). The owner of the smart wallet is able to relay functions as though it was the smart wallet executing those functions. This smart wallet will allow a keeper to execute an advanced order if and only if its conditions are met. While examining the smart wallet implementation, we notice an improvement that can be made to enhance current interactions with external entities.

Specifically, we show below the `executeCall()` routine. As the name indicates, this routine is designed to allow the owner of the smart wallet to execute any transaction on an external smart contract. The purpose is to impose no restrictions on the contracts that the user can interact with. It comes to our attention that current logic does not support the `payable` modifier, which somehow limits the functionality in not supporting the direct transfer of `ether`. Also, the internal call with `target.functionCall(callData)` (line 156) can be accordingly enhanced to carry with the given `ether`, i.e., `msg.value`.

```
38  /*
39   * @notice allows the owner of the smart wallet to execute any transaction
40   *  on an external smart contract. There are no restrictions on the contracts
41   *  that the user can interact with so needs to make sure that they do not
42   *  interact with any malicious contracts.
43   *  This utilises functions from OpenZeppelin's Address.sol
44   * @param target the address of the smart contract to interact with (will revert
45   *    if this is not a valid smart contract)
46   * @param callData the data bytes of the function and parameters to execute
47   *    Can use encodeFunctionData() from ethers.js
48   */
49
50  function executeCall(
51    address target,
52    bytes calldata callData
53  ) external override onlyOwner() returns (bytes memory) {
54    require(target.isContract(), 'call to non-contract');
55    require(factory.isWhitelisted(target), 'Invalid target contract');
56    return target.functionCall(callData);
}
```

Listing 3.5: SmartWallet:: executeCall ()

Recommendation Extend the current execution logic of `executeCall()` to support direct `ether` transfer.

Status This issue has been fixed in this commit: `005a56e`.

3.5 Accommodation of approve() Idiosyncrasies

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SmartWallet`
- Category: Time and State [3]
- CWE subcategory: CWE-362 [2]

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;
```

```
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.6: 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 `SmartWallet::_handleOpenPositionWithApproval()` routine as an example. This routine is designed to open an intended order position with the given allowance. To accommodate the specific idiosyncrasy, for each `safeIncreaseAllowance()` (line 143), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
131   function _handleOpenPositionWithApproval(
132     IAmm _asset,
133     SignedDecimal.signedDecimal memory _orderSize,
134     Decimal.decimal memory _collateral,
135     Decimal.decimal memory _leverage,
136     Decimal.decimal memory _slippage
137   ) internal {
138     //Get cost of placing order (fees)
139     (Decimal.decimal memory toll, Decimal.decimal memory spread) = _asset
140       .calcFee(_collateral.mulD(_leverage));
141     Decimal.decimal memory totalCost = _collateral.addD(toll).addD(spread);

143     IERC20(USDC).safeIncreaseAllowance(CLEARINGHOUSE, _toUint(IERC20(USDC), totalCost));

145     //Establish how much leverage will be needed for that order based on the
146     //amount of collateral and the maximum leverage the user was happy with.
147     bool _isLong = _orderSize.isNegative() ? false : true;

149     Decimal.decimal memory _size = _orderSize.abs();
150     Decimal.decimal memory _quote = (IAmm(_asset)
151       .getOutputPrice(_isLong ? IAmm.Dir.REMOVE_FROM_AMM : IAmm.Dir.ADD_TO_AMM, _size));
152     Decimal.decimal memory _offset = Decimal.decimal(1); //Need to add one wei for
          rounding
153     _leverage = minD(_quote.divD(_collateral).addD(_offset),_leverage);

155     IClearingHouse(CLEARINGHOUSE).openPosition(
156       _asset,
157       _isLong ? IClearingHouse.Side.BUY : IClearingHouse.Side.SELL,
158       _collateral,
159       _leverage,
160       _slippage
161       );
162   }
```

Listing 3.7: SmartWallet::_handleOpenPositionWithApproval()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

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

Status This issue has been fixed in this commit: `4672be8`.

4 | Conclusion

In this audit, we have analyzed the Perpetual-Limit-Orders design and implementation. The system presents a unique, robust offering as a decentralized non-custodial perpetual futures trading platform with limit orders and other advanced order types.. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Furthermore, 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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