SMART CONTRACT AUDIT REPORT

for

New Order

PeckShield

Prepared By: Xiaomi Huang

PeckShield

June 18, 2022

Document Properties

Client	New Order
Title	Smart Contract Audit Report
Target	New Order
Version	1.0-rc
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	June 18, 2022	Shulin Bie	Release Candidate

PeckShield

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

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the New Order 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 New Order

New Order is a decentralized platform that provides a series of services. It allows users to deposit their WETH as collateral to predict the token price. By doing so, the user can profit from the rise or fall of the token. Additionally, it allows users to stake or lock up the supported assets to earn yield from different farming strategies. The basic information of the audited protocol is as follows:

PeckShield

Table 1.1: Basic Information of New Order

Item	Description
Target	New Order
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 18, 2022

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

- https://github.com/new-order-network/Y2K-smartcontracts.git (96fdcf0)
- https://github.com/new-order-network/RewardsVault.git (4e2df32)

- https://github.com/new-order-network/merkle-distributor.git (9a2b109)

1.2 About PeckShield

PeckShield Inc. [10] 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

PeckShield

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- 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 contract is considered safe regarding the check item. For any discovered issue, we might further

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

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) [8], 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.

PeckShield

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 `New Order` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	🟦 🟦
Medium	1	🟦
Low	0	
Informational	2	🟦 🟦
Total	5	

PeckShield

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 2 high-severity vulnerabilities, 1 medium-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key New Order Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Logic Of Vault::withdraw()/redeem()	Business Logic	
PVE-002	High	Improper Logic Of SemiFungibleVault::setApprovalForAll()	Business Logic	
PVE-003	Informational	Immutable States If Only Set at Constructor()	Coding Practices	
PVE-004	Informational	Suggested Event Generation For Key Operations	Coding Practices	
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	

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 Improper Logic Of Vault::withdraw()/redeem()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High

- Target: Vault
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

By design, the Vault contract is one of the main entries for interaction with users, which accepts the deposits of the supported assets. In particular, one entry routine, i.e., withdraw(), is used by the user to withdraw the assets by himself or on behalf of others. While examining its logic, we observe there is an improper implementation that needs to be improved.

To elaborate, we show below the code snippet of the withdraw() routine. It accepts four input parameters: the first id parameter represents the ERC1155 token id, the second assets parameter specifies the withdrawal amount, the third receiver parameter specifies the recipient of the withdrawal assets, and the last owner parameter indicates the indeed owner of the withdrawal assets. In short, it allows the msg.sender to withdraw the assets on behalf of the specified owner. However, in the withdraw() routine, we observe there is no necessary sanity check to ensure that the owner assigns approval to the msg.sender. Given this, the malicious actor can steal other's assets.

```
    function withdraw(
        uint256 id,
        uint256 assets,
        address receiver,
        address owner
    )
        public
        override
        EpochHasEnded(id)
        marketExists(id)
        returns (uint256 shares)
    {
        shares = previewWithdraw(id, assets); // No need to check for rounding error, previewWithdraw rounds up.
        if (msg.sender != owner) {
            if (isApprovedForAll(owner, address(this))) {
                _setApprovalForAll(owner, address(this), false);
            }
        }

        uint256 entitledShares = beforeWithdraw(id, shares);
        _burn(owner, id, shares);

        emit Withdraw(msg.sender, receiver, owner, id, assets, entitledShares);
        asset.safeTransfer(receiver, entitledShares);

        return entitledShares;
    }
```

Listing 3.1: `Vault::withdraw()`

Note that the `redeem()` routine shares the same issue.

Recommendation Add necessary approval checks in above-mentioned routines.

Status

3.2 Improper Logic Of SemiFungibleVault::setApprovalForAll()

PeckShield

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High

- Target: `SemiFungibleVault`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `SemiFungibleVault` contract inherits from the `ERC1155Supply` contract, which follows the standard `ERC1155` specification. In particular, we observe it overwrites the `setApprovalForAll()` routine (designed to assign the user's own approval to the spender). While examining its logic, it comes to our attention that there is an improper implementation that needs to be improved.

To elaborate, we show below the code snippet of the `setApprovalForAll()` routine. By design, it should be used to assign the `msg.sender`'s own approval to the spender. However, we notice both the `owner` and `spender` are specified by the caller via the input `_owner` and `_spender` parameters. That is to say, the malicious actor has capability to assign anyone's approval to himself. By doing so, the malicious actor can steal anyone's assets.

```
    function setApprovalForAll(
        address _owner,
        address _spender,
        bool _approved
    ) external {
        _setApprovalForAll(_owner, _spender, _approved);
    }
```

Listing 3.2: `SemiFungibleVault::setApprovalForAll()`

Recommendation Revisit the implementation of the above-mentioned routine.

Status

3.3 Immutable States If Only Set at Constructor()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [2]

Description

PeckShield

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `New Order` protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency.

```
    contract Vault is SemiFungibleVault {

        /*//////////////////////////////////////////////////////////////////
                                    IMMUTABLES AND STORAGE
        //////////////////////////////////////////////////////////////////*/
    
        address public tokenInsured;
        ...
        int256 public strikePrice;
        address private Admin;
        ...
    }
```

Listing 3.3: `Vault`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status

3.4 Suggested Event Generation For Key Operations

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

PeckShield

While examining the events that reflect the protocol dynamics, we notice there are several key operations that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    /**
    @param _fee  uint256 of the fee value, multiply your % value by 10, Example: if you want fee of 0.5% , insert 5;
    **/
    function changeFee(uint256 _fee) public onlyAdmin {
        feeTaken = _fee;
    }

    function changeTreasury(address _treasury) public onlyAdmin {
        treasury = _treasury;
    }

    function changeTimewindow(uint256 _timewindow) public onlyAdmin {
        timewindow = _timewindow;
    }
```

Listing 3.4: Vault

With that, we suggest to emit meaningful events for these key operations. Also, the key event information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being indexed.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

PeckShield

Description

In the New Order protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setController(address _controller) public onlyAdmin {
        controller = _controller;
    }

    function changeVaultFee(uint256 _marketIndex, uint256 _fee) public onlyAdmin {
        address[] memory vaults = indexVaults[_marketIndex];
        Vault insr = Vault(vaults[0]);
        Vault risk = Vault(vaults[1]);
        insr.changeFee(_fee);
        risk.changeFee(_fee);
    }
```

Listing 3.5: VaultFactory

```
    function recoverERC20(address tokenAddress,uint256 tokenAmount) external onlyOwner {
        if (whitelistRecoverERC20[tokenAddress] == false) revert NotWhitelisted();

        uint balance = IERC20(tokenAddress).balanceOf(address(this));
        if (balance < tokenAmount) revert InsufficientBalance();

        IERC20(tokenAddress).safeTransfer(owner(), tokenAmount);
        emit RecoveredERC20(tokenAddress, tokenAmount);
    }
```

Listing 3.6: `LockRewards::recoverERC20()`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest a multi-sig account plays the privileged account to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status

PeckShield

4 Conclusion

In this audit, we have analyzed the New Order design and implementation. New Order is a decentralized platform that provides a series of services. It allows users to deposit their WETH as collateral to predict the token price. By doing so, the user can profit from the rise or fall of the token. Additionally, it allows users to stake or lock up the supported assets to earn yield from different farming strategies. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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.

PeckShield

References

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

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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

PeckShield