SMART CONTRACT AUDIT REPORT

for

Phuture

Prepared By: Xiaomi Huang

PeckShield

July 20, 2022

Document Properties

Client	Phuture
Title	Smart Contract Audit Report
Target	Phuture
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	July 20, 2022	Xuxian Jiang	Final Release
1.0-rc	July 18, 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 **4**

1.1 About Phuture 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Revisited creator Field in OrderDetails 11

3.2 Improved Logic in Orderer::externalSwap() 12

3.3 Trust Issue of Admin Keys 14

3.4 Suggested Constant Use in IndexRouter 15

3.5 Potential DoS in IndexRouter::mintSwap() 16

4 Conclusion **18**

References **19**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Phuture` 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 Phuture

`Phuture` is a decentralised crypto index platform that simplifies investments through automated, themed index funds. In particular, the index funds provide themed exposure to crypto assets, making them ideal for investors looking to upgrade their crypto investment strategy. Once set, the index strategy is managed by code and remains unchanged, in perpetuity. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Stader` Protocol

Item	Description
Issuer	Phuture
Website	https://www.phuture.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 20, 2022

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

- https://github.com/Phuture-Finance/phuture-contracts.git (9ab855b)

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

- https://github.com/Phuture-Finance/phuture-contracts.git (0680408)

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

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

Table 1.3: The Full List of Check Items

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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.

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 design and implementation of the `Phuture` 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 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	2	■■
Low	2	■■
Informational	1	■
Total	5	

We have so far identified a list of potential issues.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 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Phuture Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited creator Field in OrderDetails	Coding Practices	Resolved
PVE-002	Low	Improved Logic in Orderer::externalSwap()	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Suggested Constant Use in IndexRouter	Coding Practices	Resolved
PVE-005	Medium	Potential DoS in IndexRouter::mintSwap()	Business Logic	Resolved

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

3 | Detailed Results

3.1 Revisited creator Field in OrderDetails

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Orderer`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `Phuture` protocol has a core `Orderer` contract that contains logic for order creation and execution, as well as reweigh execution. Each order contains an order details structure with the associated assets list, the creator address, as well as the creation timestamp and asset details. Our analysis shows that the creator address is not properly recorded when the order is created.

To elaborate, we show below the related `placeOrder()` routine in the `Orderer` contract. As the name indicates, this routine is used to place an order, which naturally creates a new order structure. We notice the creation timestamp is properly initialized, but not the creator.

```
    function placeOrder() external override onlyRole(INDEX_ROLE) returns (uint _orderId)
         {
        delete orderDetailsOf[lastOrderIdOf[msg.sender]];
        unchecked {
            ++_lastOrderId;
        }
        _orderId = _lastOrderId;
        OrderDetails storage order = orderDetailsOf[_orderId];
        order.creationTimestamp = block.timestamp;
        lastOrderIdOf[msg.sender] = _orderId;
        emit PlaceOrder(msg.sender, _orderId);
    }
```

Listing 3.1: `Orderer::placeOrder()`

Recommendation Properly initialize the creator address when an order is placed.

Status This issue has been fixed in the following commit: `acfad11`.

3.2 Improved Logic in Orderer::externalSwap()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Orderer`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
```

```
    }
```

Listing 3.2: ZRX.sol

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()`.

In the following, we show the `externalSwap()` routine in the `Orderer` contract. If the `USDT` token is supported as `sellAsset`, the unsafe version of `IERC20(_details.sellAsset).transfer(address(_details.sellVToken), change)` (line 338) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
    function externalSwap(ExternalSwapV2 calldata _info) external override onlyRole(
        KEEPER_JOB_ROLE) {
        require(_info.swapTarget != address(0) && _info.swapData.length > 0, "Orderer:
            INVALID");
        require(IAccessControl(registry).hasRole(INDEX_ROLE, _info.account), "Orderer:
            INVALID");

        SwapDetails memory _details = _swapDetails(
            IIndex(_info.account).vTokenFactory(),
            address(0),
            _info.sellAsset,
            _info.buyAsset
        );
        ...
            uint change = IERC20(_details.sellAsset).balanceOf(address(this));
            if (change > 0) {
                IERC20(_details.sellAsset).transfer(address(_details.sellVToken), change
                    );
            }
        ...
    }
```

Listing 3.3: `Orderer::externalSwap()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. For the safe-version of `approve()`, there is a need to `safeApprove()` twice: the first one reduces the allowance to 0 and the second one sets the new allowance.

Status This issue has been fixed in the following commit: `acfad11`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `Phuture` redemption contracts, there is a privileged manager account (with the `ASSET_MANAGER_ROLE`) that plays a critical role in governing and regulating the system-wide operations (e.g., authorize other roles as well as configure various protocol risk parameters, etc.). Our analysis shows that the privileged account needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the privileged account.

Specifically, the privileged functions in the `IndexRegistry` contract allow for the configuration of a variety of risk parameters as well as the addition/removal of supported assets.

```
    function registerIndex(address _index, IIndexFactory.NameDetails calldata
        _nameDetails) external override {
        require(!hasRole(INDEX_ROLE, _index), "IndexRegistry: EXISTS");

        grantRole(INDEX_ROLE, _index);
        _setIndexName(_index, _nameDetails.name);
        _setIndexSymbol(_index, _nameDetails.symbol);
    }

    /// @inheritdoc IIndexRegistry
    function setMaxComponents(uint _maxComponents) external override onlyRole(
        INDEX_MANAGER_ROLE) {
        require(_maxComponents >= 2, "IndexRegistry: INVALID");

        maxComponents = _maxComponents;
        emit SetMaxComponents(msg.sender, _maxComponents);
    }

    /// @inheritdoc IIndexRegistry
    function setIndexLogic(address _indexLogic) external override onlyRole(
        INDEX_MANAGER_ROLE) {
        require(_indexLogic != address(0), "IndexRegistry: ZERO");

        indexLogic = _indexLogic;
        emit SetIndexLogic(msg.sender, _indexLogic);
    }
```

Listing 3.4: Example Privileged Operations in `IndexRegistry`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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 The issue has been confirmed by the team. The team clarifies that the above admin key will be managed by a multisig account.

3.4 Suggested Constant Use in IndexRouter

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `IndexRouter`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [2]

Description

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 `Phuture` protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency. Also, if the state variables are fixed constants, we can simply define them as `constant`. Examples include the `IndexRouter` contract, which defines a number of roles and these roles can be simply defined as `constant`, avoiding the need of dynamically initializing them.

```
    /// @notice Index role
    bytes32 internal INDEX_ROLE;
    /// @notice Asset role
    bytes32 internal ASSET_ROLE;
    /// @notice Exchange admin role
    bytes32 internal EXCHANGE_ADMIN_ROLE;
    /// @notice Skipped asset role
    bytes32 internal SKIPPED_ASSET_ROLE;
    /// @notice Exchange target role
    bytes32 internal EXCHANGE_TARGET_ROLE;
    }
```

Listing 3.5: `IndexRouter`

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

Status This issue has been fixed in the following commit: `acfad11`.

3.5 Potential DoS in IndexRouter::mintSwap()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `IndexRouter`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Phuture` protocol provides an `IndexRouter` contract, which is designed to be the main entry for interaction with protocol users. In particular, one entry routine, i.e., `mintSwap()`, accepts asset transfer-in, swaps and sends assets in certain proportions to `vTokens`, and mints the corresponding index tokens to represent the depositor's share. Our analysis shows that the current implementation has a potential denial-of-service issue.

To elaborate, we show below the implementation of the related `_mint()` routine, which is invoked inside `mintSwap()`. We notice the requirement statement at the end of `_mint()`, i.e., `require(IERC20(_inputToken).balanceOf(address(this))== 0)` (line 393). This enforcement is error-prone as a malicious actor may intentionally donate a tiny amount of `inputToken`, which renders the `mintSwap()` routine non-functional!

```
365     function _mint(
366         address _index,
367         address _inputToken,
368         uint _amountInInputToken,
```

```
        MintQuoteParams[] calldata _quotes
    ) internal {
        uint quotesCount = _quotes.length;
        IvTokenFactory vTokenFactory = IvTokenFactory(IIndex(_index).vTokenFactory());
        for (uint i; i < quotesCount; i++) {
            address asset = _quotes[i].asset;

            // if one of the assets is inputToken we transfer it directly to the vault
            if (asset == _inputToken) {
                IERC20(_inputToken).safeTransfer(
                    vTokenFactory.createdVTokenOf(_inputToken),
                    _quotes[i].buyAssetMinAmount
                );
                continue;
            }
            address swapTarget = _quotes[i].swapTarget;
            require(IAccessControl(registry).hasRole(EXCHANGE_TARGET_ROLE, swapTarget),
                "IndexRouter: INVALID_TARGET");
            _safeApprove(_inputToken, swapTarget, _amountInInputToken);
            // execute the swap with the quote for the asset
            _fillQuote(swapTarget, _quotes[i].assetQuote);
            uint assetBalanceAfter = IERC20(asset).balanceOf(address(this));
            require(assetBalanceAfter >= _quotes[i].buyAssetMinAmount, "IndexRouter:
                UNDERBOUGHT_ASSET");
            IERC20(asset).safeTransfer(vTokenFactory.createdVTokenOf(asset),
                assetBalanceAfter);
        }
        require(IERC20(_inputToken).balanceOf(address(this)) == 0, "IndexRouter:
            INVALID_INPUT_AMOUNT");
    }
```

Listing 3.6: `IndexRouter::_mint()`

Recommendation Revise the above `_mint()` function to avoid the above denial-of-service situation.

Status This issue has been fixed in the following commit: `acfad11`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Phuture` protocol, which is a decentralised crypto index platform that simplifies investments through automated, themed index funds. In particular, the index funds provide themed exposure to crypto assets, making them ideal for investors looking to upgrade their crypto investment strategy. Once set, the index strategy is managed by code and remains unchanged, in perpetuity. The current code base is clearly organized and 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-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.