SMART CONTRACT AUDIT REPORT

for

Phuture FRPVault

Prepared By: Xiaomi Huang

PeckShield

August 25, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	August 25, 2022	Luck Hu	Final Release
1.0-rc	August 15, 2022	Luck Hu	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 `Phuture FRPVault` contract, 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 FRPVault

`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. The audited `FRPVault` is introduced to make it easy for investors to get access to fixed rate yields without having to manually manage the maturities and choose the highest yielding maturities each time. Meanwhile it targets to make users transactions as gas efficient as possible. The basic information of the audited `FRPVault` is as follows:

Table 1.1: Basic Information of The `Phuture FRPVault`

Item	Description
Issuer	Phuture
Website	https://www.phuture.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 25, 2022

In the following, we show the Git repository of reviewed file and the commit hash value used in this audit. Note this audit only covers the `src/FRPVault.sol` contract.

- https://github.com/Phuture-Finance/phuture-frp-contracts.git (b6b7a7e)

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-frp-contracts.git (c5b11df)

1.2 About PeckShield

PeckShield Inc. [7] 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 [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.

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.

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

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) 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 FRPVault`. 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	3	■■■
Low	1	■
Informational	0	
Total	4	

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 contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key Phuture FRPVault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Redeem Share Distribution	Business Logic	Fixed
PVE-002	Medium	Incorrect Deposit Share Distribution	Business Logic	Fixed
PVE-003	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Incorrect Redeem Share Distribution

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FRPVault
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The FRPVault contract inherits from the ERC4626Upgradeable of OpenZeppelin and overwrites the redeem ()/withdraw() interfaces to charge a burning fee. While examining the shares distribution between the user and the fee recipient, we notice the distribution is incorrect.

To elaborate, we show below the code snippet from the FRPVault contact. As the name indicates, the redeem() function is used to redeem _shares amount of shares from the _owner and transfer the underlying assets to the _receiver. The input _shares consists of two parts. One part could be withdrawn to the user and the left is charged as the burning fee to the fee recipient. The burning fee is charged on top of the shares to be withdrawn to the user. If we assume the shares withdrawn to the user is A, the burning fee is computed as A*fee with the following equation: A *fee + A = _shares. As a result, A = _shares/(1+fee), where fee = BURNING_FEE_IN_BP/BP. We can further derive A = (_shares * BP)/ (BP + BURNING_FEE_IN_BP) and the burning fee = A*fee = (_shares * BURNING_FEE_IN_BP)/ (BP + BURNING_FEE_IN_BP).

However, the redeem() function directly uses the input _shares to calculate the burning fee (line 178). Per our calculation, it shall use (_shares * BP)/ (BP + BURNING_FEE_IN_BP) as the base to calculate the burning fee.

What is more, it shares the same issue in the previewRedeem() routine where the burning fee is calculated based on the input _shares, not expected (_shares * BP)/ (BP + BURNING_FEE_IN_BP) (line 228).

```
    /// @inheritdoc IERC4626Upgradeable
    function redeem(
        uint256 _shares,
        address _receiver,
        address _owner
    ) public override returns (uint256) {
        require(_shares <= maxRedeem(_owner), "FRPVault: redeem more than max");
        // previewReedem is fine to use here since we are dealing with exact input of shares so we calculate burning fee on that
        uint256 assetsMinusFee = previewRedeem(_shares);
        uint fee = _chargeBurningFee(_shares, _owner);
        // burns _shares - fee since fee is transferred to the feeRecipient
        _withdraw(msg.sender, _receiver, _owner, assetsMinusFee, _shares - fee);

        return assetsMinusFee;
    }
```

Listing 3.1: `FRPVault::redeem()`

```
    /// @inheritdoc IERC4626Upgradeable
    function previewRedeem(uint256 _shares) public view override returns (uint256) {
        // amount of assets received is reduced by the shares amount
        return convertToAssets(_shares - (_shares * BURNING_FEE_IN_BP) / BP);
    }
```

Listing 3.2: `FRPVault::previewRedeem()`

Recommendation Revise the above mentioned `redeem()/previewRedeem()` to correctly distribute the shares between the user and the fee recipient.

Status This issue has been fixed in the following commit: `401bd34`.

3.2 Incorrect Deposit Share Distribution

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FRPVault`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned earlier, the `FRPVault` contract inherits from the `ERC4626Upgradeable` of `OpenZeppelin` and overwrites the `mint()/deposit()` interfaces to charge a minting fee. While examining the distribution of the new minted shares between the user and the fee recipient, we notice the distribution is incorrect.

To elaborate, we show below the code snippet from the FRPVault contact. As the name indicates, the deposit() function is used to deposit _assets amount of assets to the vault and mint the new shares to the _receiver. The deposit() function converts the input _assets to the new shares to be minted. The new shares consists of two parts. One part is minted to the user and the left is minted to the fee recipient as minting fee. The minting fee is charged on top of the shares to be minted to the user. If we assume the shares to the user is A, the minting fee is A*fee with the following equation: A*fee + A = shares. As a result, A = shares/(1+fee), where fee = MINTING_FEE_IN_BP/BP. Moreover, A = (_shares * BP)/ (BP + MINTING_FEE_IN_BP) and the minting fee = A*fee = (_shares * MINTING_FEE_IN_BP)/ (BP + MINTING_FEE_IN_BP).

However, the deposit() function directly uses the total shares as the base to calculate the minting fee (line 209). Per our calculation, it shall use (_shares * BP)/ (BP + MINTING_FEE_IN_BP) as the base. What is more, it shares the same issue in the previewDeposit() routine which shall use (_shares * BP)/ (BP + MINTING_FEE_IN_BP) as the base to calculate the minting fee (line 240).

```
202     /// @inheritdoc ERC4626Upgradeable
203     function deposit(uint256 _assets, address _receiver) public override returns (
            uint256) {
204         require(_assets <= maxDeposit(_receiver), "FRPVault: deposit more than max");
205         // calculate the shares to mint
206         uint shares = convertToShares(_assets);
207         // charge the actual fees
208         _chargeAUMFee();
209         uint fee = (shares * MINTING_FEE_IN_BP) / BP;
210         if (fee != 0) {
211             _mint(feeRecipient, fee);
212         }
213         _deposit(msg.sender, _receiver, _assets, shares - fee);
214         return shares - fee;
215     }
```

Listing 3.3: FRPVault::deposit()

```
237     /// @inheritdoc ERC4626Upgradeable
238     function previewDeposit(uint256 _assets) public view override returns (uint256) {
239         uint shares = super.previewDeposit(_assets);
240         uint fee = (shares * MINTING_FEE_IN_BP) / BP;
241         // While depositing exact amount of assets user receives shares minus fee payed
                on that amount
242         return shares - fee;
243     }
```

Listing 3.4: FRPVault::previewDeposit()

Recommendation Revise the above mentioned deposit()/previewDeposit() to correctly distribute the shares minted to the user and the fee recipient.

Status This issue has been fixed in the following commit: 401bd34.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: FRPVault
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The FRPVault contract provides one entry routine, i.e., deposit(), via which users can deposit the underlying asset into the vault. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the vault. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the code snippet of the deposit() routine.

```
function deposit(uint256 _assets, address _receiver) public override returns (uint256) {
    require(_assets <= maxDeposit(_receiver), "FRPVault: deposit more than max");
    // calculate the shares to mint
    uint shares = convertToShares(_assets);
    // charge the actual fees
    _chargeAUMFee();
    uint fee = (shares * MINTING_FEE_IN_BP) / BP;
    if (fee != 0) {
        _mint(feeRecipient, fee);
    }
    _deposit(msg.sender, _receiver, _assets, shares - fee);
    return shares - fee;
}
```

Listing 3.5: FRPVault::deposit()

```
    /**
    * @dev Deposit/mint common workflow.
    */
   function _deposit(
       address caller,
       address receiver,
       uint256 assets,
       uint256 shares
   ) internal virtual {
       // If _asset is ERC777, `transferFrom` can trigger a reenterancy BEFORE the transfer happens through the
       // `tokensToSend` hook. On the other hand, the `tokenReceived` hook, that is triggered after the transfer,
       // calls the vault, which is assumed not malicious.
       //
```

```
        // Conclusion: we need to do the transfer before we mint so that any reentrancy would happen before the
        // assets are transfered and before the shares are minted, which is a valid state.
        // slither-disable-next-line reentrancy-no-eth
        SafeERC20Upgradeable.safeTransferFrom(_asset, caller, address(this), assets);
        _mint(receiver, shares);

        emit Deposit(caller, receiver, assets, shares);
    }
```

Listing 3.6: `ERC4626Upgradeable::_deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into the `FRPVault` for depositing. In fact, the `FRPVault` is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been confirmed by the team that they do not intend to support these types of tokens as an asset for the vault.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FRPVault
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the FRPVault contact, there is a VAULT_MANAGER_ROLE (granted by the VAULT_ADMIN_ROLE), that plays a critical role in governing and regulating the system-wide operations. Our analysis shows that this privileged role needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the VAULT_MANAGER_ROLE.

Specifically, the privileged functions in FRPVault allow for the VAULT_MANAGER_ROLE to set the maxLoss , upgrade the vault, and allow for the DEFAULT_ADMIN_ROLE to grant new VAULT_ADMIN_ROLE which can further grant new VAULT_MANAGER_ROLE.

```
    /// @inheritdoc IFRPVault
    function setMaxLoss(uint16 _maxLoss) external isValidMaxLoss(_maxLoss) {
        require(hasRole(VAULT_MANAGER_ROLE, msg.sender), "FRPVault: FORBIDDEN");
        maxLoss = _maxLoss;
    }

    /// @inheritdoc UUPSUpgradeable
    function _authorizeUpgrade(address _newImpl) internal view virtual override {
        require(hasRole(VAULT_MANAGER_ROLE, msg.sender), "FRPVault: FORBIDDEN");
    }
```

Listing 3.7: Example Privileged Operations in the FRPVault Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the VAULT_MANAGER_ROLE may also be a counter-party risk to the protocol users. It is worrisome if the privileged VAULT_MANAGER_ROLE 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 mitigated as the team confirms they plan to use multi-sig for all privileged roles.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Phuture FRPVault`. `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. The audited `FRPVault` is introduced to make it easy for investors to get access to fixed rate yields without having to manually manage the maturities and choose the highest yielding maturities each time. Meanwhile it targets to make users transactions as gas efficient as possible. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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