SMART CONTRACT AUDIT REPORT

for

Vovo Finance

Prepared By: Yiqun Chen

PeckShield

February 26, 2022

Document Properties

Client	Vovo Finance
Title	Smart Contract Audit Report
Target	Vovo Finance
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 26, 2022	Jing Wang	Final Release
1.0-rc	December 12, 2021	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
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 `Vovo Finance` 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 Vovo Finance

`Vovo Finance` provides passive investment returns with customizable risks. The `token` product of the protocol allows users to stake `Vovo` tokens and receive rewards. The `PrincipalProtectedVault` product of the protocol receives `vaultToken` from users and deposits received `vaultToken` into yield farming pools. Periodically, the vault collects the yield rewards and uses the rewards to open a leverage trade on a perpetual swap exchange.

The basic information of the `Vovo Finance` protocol is as follows:

Table 1.1: Basic Information of The `Vovo Finance` Protocol

Item	Description
Name	Vovo Finance
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 26, 2022

In the following, we list the reviewed files and the commit hash values used in this audit.

- https://github.com/VovoFinance/VovoProducts.git (c110397)

- https://github.com/VovoFinance/token.git (5266827)

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

- https://github.com/VovoFinance/VovoProducts.git (39d5922)
- https://github.com/VovoFinance/token.git (0613f4d)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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) [10], 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 `Vovo Finance` 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	5	
Low	3	
Informational	0	
Total	8	

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 some 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 5 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Vovo Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Costly LPs From Improper Vault Initialization	Time and State	Confirmed
PVE-002	Medium	Suggested Permission-Restricted earn()	Time and State	Mitigated
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Mitigated
PVE-005	Low	Proper Handling of Switching isLong in setIsLong()	Business Logic	Fixed
PVE-006	Low	Price Manipulation Of getUnderlying-Price()	Business Logics	Fixed
PVE-007	Medium	Improved Logic of Vesting::revoke()	Business Logics	Fixed
PVE-008	Medium	Proper Handling of expectedVaultTo-kenAmount calculation in _withdra-wOne()	Business Logics	Fixed

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 Possible Costly LPs From Improper Vault Initialization

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: PrincipalProtectedVault
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The PrincipalProtectedVault contract aims to provide incentives so that users can stake and lock their funds in a stake pool. The staking users will get their pro-rata share based on their staked amount. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the deposit() routine. This deposit() routine is used for participating users to deposit the supported asset (e.g., vaultToken) and get respective rewards in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
function deposit(uint256 amount) public {
   uint256 _pool = balance();
   require(isDepositEnabled && _pool.add(amount) < cap, "!deposit");
   uint256 _before = IERC20(vaultToken).balanceOf(address(this));
   IERC20(vaultToken).safeTransferFrom(msg.sender, address(this), amount);
   uint256 _after = IERC20(vaultToken).balanceOf(address(this));
   amount = _after.sub(_before); // Additional check for deflationary tokens
   uint256 shares = 0;
   if (totalSupply() == 0) {
     shares = amount;
   } else {
     shares = (amount.mul(totalSupply())).div(_pool);
   }
   _mint(msg.sender, shares);
   emit Minted(msg.sender, shares);
```

```
199     }
```

Listing 3.1: `PrincipalProtectedVault::deposit()`

Specifically, when the pool is being initialized, the share value directly takes the value of `shares = amount` (line 193), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = 1 WEI`. With that, the actor can further deposit a huge amount of `vaultToken` with the goal of making the share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of share calculation to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status The issue has been confirmed by the team. And the team clarifies that they will ensure guarded launch that safeguards the first deposit to avoid being manipulated.

3.2 Suggested Permission-Restricted earn()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PrincipalProtectedVault`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `PrincipalProtectedVault` protocol is designed and implemented to invest farmers' assets (in `vaultToken`), harvest growing yields, and sell any gains, if any, to the original asset. In order to have a smooth investment experience, the `PrincipalProtectedVault` protocol opens up a public function, i.e., `earn()`, that can be invoked by anyone to kick off the investment.

```
    function earn() public {
        uint256 tokenBalance = IERC20(vaultToken).balanceOf(address(this));
        if (tokenBalance > 0) {
          IERC20(vaultToken).safeApprove(lpToken, 0);
          IERC20(vaultToken).safeApprove(lpToken, tokenBalance);
          uint256 expectedLpAmount = tokenBalance.mul(1e18).div(vaultTokenBase).mul(1e18
              ).div(ICurveFi(lpToken).get_virtual_price());
          uint256 lpMinted = ICurveFi(lpToken).add_liquidity([tokenBalance, 0],
              expectedLpAmount.mul(DENOMINATOR.sub(slip)).div(DENOMINATOR));
          emit LiquidityAdded(tokenBalance, lpMinted);
        }
        uint256 lpBalance = IERC20(lpToken).balanceOf(address(this));
        if (lpBalance > 0) {
          IERC20(lpToken).safeApprove(gauge, 0);
          IERC20(lpToken).safeApprove(gauge, lpBalance);
          Gauge(gauge).deposit(lpBalance);
          emit GaugeDeposited(lpBalance);
        }
      }
```

Listing 3.2: `PrincipalProtectedVault::earn()`

Unfortunately, this public entry has been exploited in a number of recent incidents (yDAI and BT hacks [13, 1]) that prompt the need of a guarded call to the `earn()`. By doing so, it ensures the assets in `PrincipalProtectedVault` will not blindly deposited into a pool that is currently not making any profit.

Recommendation Ensure the `earn()` can only be called via a trusted entity.

Status This issue has been confirmed by the team. And the team clarifies that they will set the slippage at `0.3%` and manually adjust the slippage as time goes depending on the fund size. Also, `isKeeperOnly` is added to restrict the calling of `earn()`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Vovo Finance` protocol, there is a special administrative account, i.e., `admin/owner`. This `admin/owner` account plays a critical role in governing and regulating the system-wide operations (e.g., minting tokens, setting protocol-wide risk parameters, moving assets, etc.). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the `mint()` functions in the `Vovo Finance` token contract, which allows the `minter` to add tokens into circulation and the recipient can be directly provided when the mint operation takes place.

```
    function mint(address dst, uint rawAmount) external {
        require(msg.sender == minter, "VOVO::mint: only the minter can mint");
        require(dst != address(0), "VOVO::mint: cannot transfer to the zero address");
        // mint the amount
        uint96 amount = safe96(rawAmount, "VOVO::mint: amount exceeds 96 bits");
        totalSupply = safe96(SafeMath.add(totalSupply, amount), "VOVO::mint: totalSupply
             exceeds 96 bits");

        // transfer the amount to the recipient
        balances[dst] = add96(balances[dst], amount, "VOVO::mint: transfer amount
            overflows");
        emit Transfer(address(0), dst, amount);

        // move delegates
        _moveDelegates(address(0), delegates[dst], amount);
    }
```

Listing 3.3: `Vovo::mint()`

Also, the `admin` of the `PrincipalProtectedVault` protocol takes the important responsibility to manage `keepers` and `governor`, who are able to withdraw all funds from the contract.

```
    function withdrawAsset(address _asset) external {
        require(keepers[msg.sender]  msg.sender == governor, "!keepers");
        IERC20(_asset).safeTransfer(msg.sender, IERC20(_asset).balanceOf(address(this)))
            ;
      }
```

Listing 3.4: `PrincipalProtectedVault::withdrawAsset()`

It is worrisome if the privileged `owner`/`admin` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies that they will use the `TimeLock` contract to be the owner of the `Vovo` token contract. Also they will use a multi-sig contract to be the owner of the `TimeLock` contract. For the `PrincipalProtectedVault::withdrawAsset()` routine, the team adds the exclusion for the `vaultToken` to mitigate this issue by this commit: `6ca5090`.

3.4 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PrincipalProtectedVault`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `PrincipalProtectedVault` contract has a helper routine, i.e., `collectReward()`, that is designed to collect the yield rewards and use the rewards to open a leverage trade on a `perpetual swap exchange` contract. It has a rather straightforward logic to swap the rewards to the `underlying` tokens by calling `swapExactTokensForTokens()` to actually perform the intended token swap.

```
  function collectReward() private returns(uint256 tokenReward) {
      uint256 _before = IERC20(underlying).balanceOf(address(this));
      Gauge(gauge).claim_rewards(address(this));
      uint256 _crv = IERC20(crv).balanceOf(address(this));
      if (_crv > 0) {
        IERC20(crv).safeApprove(dex, 0);
        IERC20(crv).safeApprove(dex, _crv);
        address[] memory path;
        if (underlying == weth) {
          path = new address[](2);
          path[0] = crv;
          path[1] = weth;
        } else {
          path = new address[](3);
          path[0] = crv;
          path[1] = weth;
          path[2] = underlying;
        }
        Uni(dex).swapExactTokensForTokens(_crv, 0, path, address(this), block.
            timestamp.add(1800))[path.length - 1];
      }
      uint256 _after = IERC20(underlying).balanceOf(address(this));
      tokenReward = _after.sub(_before);
      totalFarmReward = totalFarmReward.add(tokenReward);
      emit Harvested(tokenReward, totalFarmReward);
```

```
249         }
```

Listing 3.5: `PrincipalProtectedVault::collectReward()`

To elaborate, we show above the `collectReward()` routine. We notice the token swap are routed to `dex` and the actual swap operation via `swapExactTokensForTokens()` (line 243) essentially do not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense. Note another routine `closeTrade()` shares the same issue.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the team. And the team clarifies that in the short term, the contract will be deployed on `Arbitrum` with single sequencer without any MEV concerns. Also, `isKeeperOnly` is added and when it's possible to do MEV in future on `Arbitrum`, the team could set the `isKeeperOnly` to true, and only trigger the transaction via `flashbots` if it's supported.

3.5 Proper Handling of Switching isLong in setIsLong()

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `PrincipalProtectedVault`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `PrincipalProtectedVault` contract provides a `poke()` routine to collect rewards from the `Curve Gauge` contract and use the rewards to open a new leverage trade on a `perpetual swap exchange` contract. To elaborate, we show below the related routines.

```
function poke() external {
```

```
    require(keepers[msg.sender]  msg.sender == governor, "!keepers");
    require(lastPokeTime + pokeInterval < block.timestamp, "!poke time");
    uint256 tokenReward = 0;
    if (Gauge(gauge).balanceOf(address(this)) > 0) {
      tokenReward = collectReward();
    }
    closeTrade();
    if (tokenReward > 0) {
      openTrade(tokenReward);
    }
    earn();
    lastPokeTime = block.timestamp;
  }
```

Listing 3.6: `PrincipalProtectedVault::poke()`

```
  function openTrade(uint256 amount) private {
    address[] memory _path = new address[](1);
    _path[0] = underlying;
    uint256 _price = isLong ? IVault(gmxVault).getMaxPrice(underlying) : IVault(gmxVault).getMinPrice(underlying);
    uint256 _sizeDelta = leverage.mul(amount).mul(getUnderlyingPrice()).mul(1e12).div(underlyingBase);
    IERC20(underlying).safeApprove(gmxRouter, 0);
    IERC20(underlying).safeApprove(gmxRouter, amount);
    IRouter(gmxRouter).increasePosition(_path, underlying, amount, 0, _sizeDelta, isLong, _price);
    emit OpenPosition(underlying, _sizeDelta, isLong);
  }
```

Listing 3.7: `PrincipalProtectedVault::openTrade()`

```
  function closeTrade() private {
    (uint256 size,,,,,,,) = IVault(gmxVault).getPosition(address(this), underlying, underlying, isLong);
    if (size == 0) {
      return;
    }
    ...
  }
```

Listing 3.8: `PrincipalProtectedVault::closeTrade()`

```
  function setIsLong(bool _isLong) external onlyGovernor {
    isLong = _isLong;
    emit isLongSet(isLong);
  }
```

Listing 3.9: `PrincipalProtectedVault::setIsLong()`

We notice the `closeTrade()` routine is using the combination of `account`, `collateralToken`, `indexToken` and `isLong` to query the previous opened position. If `isLong` is switched by `Governor`, the positioned opened before switching positions could not be closed as the `getPosition()` would fail to return the previous position opened.

Recommendation Close the previous opened position before `setIsLong()`.

Status The issue has been fixed by this commit: `d661b81`.

3.6 Price Manipulation Of getUnderlyingPrice()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PrincipalProtectedVault`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The contract `PrincipalProtectedVault` defines a main function, i.e., `getUnderlyingPrice()`. This function is used to obtain the price of `underlying` token based on `USDC` price on the market. During the analysis of the `PrincipalProtectedVault::getUnderlyingPrice()`, we notice the price of `underlying` token is possible to be manipulated. In the following, we show the code snippet of the `getLpTokenValue()` function.

```
394    function getUnderlyingPrice() public view returns (uint256) {
395        address pair = IUniswapV2Factory(dexFactory).getPair(usdc, underlying);
396        (uint112 reserve0, uint112 reserve1,) = IUniswapV2Pair(pair).getReserves();
397        if (usdc > underlying) {
398          (reserve0, reserve1) = (reserve1, reserve0);
399        }
400        return uint256(reserve0).mul(1e18).div(uint256(reserve1)).mul(underlyingBase).
               div(usdcBase);
401      }
```

Listing 3.10: `PrincipalProtectedVault::getUnderlyingPrice()`

Specifically, if we examine the implementation of the `getUnderlyingPrice()`, the final price of the `underlying` token is derived from `uint256(reserve0).mul(1e18).div(uint256(reserve1)).mul(underlyingBase).div(usdcBase)` (line 400), where `reserve0` or `reserve1` is the token amount in the pool thus can be manipulated by flash loans, which will cause the final values of the `underlying` token not trustworthy.

Recommendation Consult an `Oracle` to get the price of `underlying` token.

Status The issue has been fixed by this commit: `d661b81`.

3.7 Improved Logic of Vesting::revoke()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Vesting
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the Vovo Finance protocol, the Vesting contract is used to management the schedule of the user vesting. It allows the owner to add the vesting schedule for each payee and each payee could claim available vested funds based on the schedule. While reviewing the implementation of the revoking logic, we found the the revoked user could double release their funds from the vesting contract. To elaborate, we show below the revoke() routine and the _vestedAmount() routine from the Vesting contract.

```
141   function revoke(address beneficiary) external onlyOwner {
142     require(_revocable, "Vesting: cannot revoke");
143     require(!_revoked[beneficiary], "Vesting: token already revoked");
144 
145     uint256 balance = _beneficiaries[beneficiary].amount;
146 
147     uint256 unreleased = _releasableAmount(beneficiary);
148     uint256 refund = balance.sub(unreleased);
149 
150     if (_upfrontReleased[beneficiary]) {
151         refund = refund.sub(_beneficiaries[beneficiary].upfront);
152     }
153 
154     _revoked[beneficiary] = true;
155 
156     vovo.safeTransfer(owner(), refund);
157 
158     emit TokenVestingRevoked(beneficiary);
159 }
```

Listing 3.11: Vesting::revoke()

We notice the refund amount of tokens, which is the unreleased amount of tokens should be calculated from balance.sub(_released[beneficiary]) rather than balance.sub(unreleased) (line 148). Also, the unreleased amount is released to user but not counted into _released[beneficiary], thus will cause this part of funds be double released.

Recommendation Refund the balance.sub(_released[beneficiary]) amount of tokens and update _released[beneficiary] afterward.

Status The issue has been fixed by this commit: `0613f4d`.

3.8 Proper Handling of expectedVaultTokenAmount calculation in _withdrawOne()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PrincipalProtectedVault`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `PrincipalProtectedVault` contract, the `withdraw()` function allows the user to withdraw `vaultToken` from the vault. It will withdraw `LP tokens` from the `Curve Gauge` contract and remove the liquidity from `Curve Pool` if needed. While reviewing the implementation, we notice the calculation of `expectedVaultTokenAmount` in `_withdrawOne()` is incorrect. To elaborate, we show below the related routine.

```
383  function _withdrawOne(uint256 _amnt) private {
384    IERC20(lpToken).safeApprove(lpToken, 0);
385    IERC20(lpToken).safeApprove(lpToken, _amnt);
386    uint256 expectedVaultTokenAmount = _amnt.mul(vaultTokenBase).div(ICurveFi(lpToken).
         get_virtual_price());
387    ICurveFi(lpToken).remove_liquidity_one_coin(_amnt, 0, expectedVaultTokenAmount.mul(
         DENOMINATOR.sub(slip)).div(DENOMINATOR));
388  }
```

Listing 3.12: PrincipalProtectedVault :: _withdrawOne()

The `expectedVaultTokenAmount` (line 386) should be derived from `_amnt.mul(ICurveFi(lpToken).get_virtual_price())` while the current implementation is using `_amnt.div(ICurveFi(lpToken).get_virtual_price())`.

Recommendation Properly compute the `expectedVaultTokenAmount` value in `_withdrawOne()`.

Status The issue has been fixed by this commit: `d661b81`.

4 Conclusion

In this audit, we have analyzed the `Vovo Finance` protocol design and implementation. `Vovo Finance` provides two products: the `token` product and the `PrincipalProtectedVault` product. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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.

References

[1] BT Finance. BT.Finance Exploit Analysis Report. https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[13] PeckShield. The yDAI Incident Analysis: Forced Investment. https://peckshield.medium.com/the-ydai-incident-analysis-forced-investment-2b8ac6058eb5.