SMART CONTRACT AUDIT REPORT

for

Feeder Finance

Prepared By: Yiqun Chen

PeckShield

July 9, 2021

Document Properties

Client	Feeder Finance
Title	Smart Contract Audit Report
Target	FeedVaults
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 9, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 3, 2021	Xuxian Jiang	Release Candidate #1
0.3	June 30, 2021	Xuxian Jiang	Additional Findings #2
0.2	June 28, 2021	Xuxian Jiang	Additional Findings #1
0.1	June 25, 2021	Xuxian Jiang	Initial Draft

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 Feeder Finance, 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 Feeder Finance

Feeder Finance is a DeFi aggregator for diversified yield generation on Binance Smart Chain (BSC). The protocol aims to allow investors to feed capital into lending protocols, liquidity pools, vaults, and other DeFi products in an automated and diversified way. Through a single deposit, investors are able to spread investments across multiple platforms, ensuring capital are more secure from a single incident, yields are optimized, and investment process simplified. As a result, the protocol will help to lower the entry barrier for normal users to benefit from the protocol gains.

The basic information of the Feeder protocol is as follows:

Table 1.1: Basic Information of The Feeder Protocol

Item	Description
Issuer	Feeder Finance
Website	https://feeder.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 9, 2021

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

- https://github.com/FeederFinance/vaults-contracts.git (9afacf3)

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

- https://github.com/FeederFinance/vaults-contracts.git (2c1cf56)

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 FeedVaults 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	2	■■
Medium	3	■■■
Low	4	■■■■
Informational	0	
Undetermined	1	■
Total	10	

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, 3 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and and 1 undetermined issue.

Table 2.1: Key FeedVaults Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Revisited Share Calculation in FeedVault::deposit()	Business Logic	Fixed
PVE-002	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Fixed
PVE-003	Low	Accommodation Of Possible Non-Compliant ERC20 Tokens	Coding Practices	Fixed
PVE-004	Medium	Proper Asset Rebalance For Disabled Target Vaults	Business Logic	Fixed
PVE-005	Low	Improper withdraw() in TargetVaultDopple	Coding Practices	Fixed
PVE-006	Medium	Incorrect Fee Collection in TargetVaultPancake::_collectFees()	Business Logic	Fixed
PVE-007	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-008	Low	Improved Emergency Withdrawal in TargetVaultBunny	Business Logic	Fixed
PVE-009	High	Force Investment Risk in TargetVaultDopple	Business Logic	Fixed
PVE-010	High	Improper balanceOf() in TargetVaultDopple/TargetVaultACrypto	Business Logic	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 Revisited Share Calculation in FeedVault::deposit()

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: FeedVault
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

At the core of the Feeder protocol, there is a FeedVault contract that allows the investor deposit funds that will be redirected into various target vaults for investment. Upon the deposit, the investor will get return the pool share tokens representing the yields from the overall investment. When examining the pool share calculation, we notice the current approach can be better improved.

To elaborate, we show below the related deposit() routine. It implements a rather straightforward logic in transferring the funds into the FeedVault. collecting necessary entry fees, and computing the pool token shares. However, the current pool token share is calculated based on the difference from before and after the rebalance. While we agree with the methodology, the actual implementation in using depositedBalance() (lines 361 and 367) may not properly reflect the deserved share. The actual balance difference can be better computed using depositedTokenBalance(false).

```
328     function deposit(uint256 _amount) public virtual isEnabled hasVaults nonReentrant {
329         // Transfer Token from Depositor to Vault
330         token.safeTransferFrom(address(msg.sender), address(this), _amount);

332         // Collect Entry Fees
333         uint256 _entryFees = _amount.mul(entryFeesBP).div(10000);
334         if (_entryFees > 0) {
335             if (autoSwapEntryFees) {
336                 uint256 buyBackBefore = IERC20(entrySwapToken).balanceOf(address(this));
337                 IERC20(token).safeIncreaseAllowance(swapRouterAddress, _entryFees);
338                 IPancakeRouter02(swapRouterAddress).
                        swapExactTokensForTokensSupportingFeeOnTransferTokens(
```

```
                    _entryFees,
                    0,
                    entryFeesToTokenPath,
                    address(this),
                    block.timestamp + 120
                );
                uint256 buyBackAfter = IERC20(entrySwapToken).balanceOf(address(this));
                uint256 buyBackAmount = buyBackAfter.sub(buyBackBefore);
                IERC20(entrySwapToken).safeTransfer(entryFeesCollector, buyBackAmount);

                emit FeesCollected(true, address(entryFeesCollector), address(
                    entrySwapToken), _entryFees);
            } else {
                IERC20(token).safeTransfer(entryFeesCollector, _entryFees);
            }
        }

        // Collect Target Vaults Profit Shares
        uint256 length = vaultInfo.length;
        for (uint256 i = 0; i < length; i++) {
            if (ITargetVault(vaultInfo[i].targetVault).balanceOfToken() > 0)
                ITargetVault(vaultInfo[i].targetVault).collectFees();
        }

        uint256 _beforeDepositBalance = depositedBalance();

        // Allocate to vaults in order to get new vaults balance
        _allocate();

        // Get Pool Balance after deposited
        uint256 _afterDepositBalance = depositedBalance();

        // Additional check for deflationary tokens
        uint256 _balance = _afterDepositBalance.sub(_beforeDepositBalance);

        if (_balance > 0) {
            uint256 shares = 0;
            uint256 totalSupply = totalSupply();
            if (totalSupply == 0) {
                shares = _balance;
            } else {
                shares = (_balance.mul(totalSupply)).div(_beforeDepositBalance);
            }
            _mint(address(msg.sender), shares);
        }

        emit Deposited(address(msg.sender), _balance);
    }
```

Listing 3.1: `FeedVault::deposit()`

Recommendation Properly compute the pool share for each deposit to reflect the fair contribution to the pool.

Status The issue has been fixed by this commit: `5db7d3`.

3.2 Possible Sandwich/MEV Attacks For Reduced Returns

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

Description

As mentioned in Section 3.1, the `Feeder` protocol has designed a generic approach to invest VC funds, harvest growing yields, and collect any gains, if any, to the share holders. In the meantime, we notice the `Feeder` protocol takes a different approach by directly rewarding the yields back to investors.

To elaborate, we show below the `harvest()` function in `TargetVaultPancake`. This routine essentially collects any pending rewards and then re-invests the collected rewards for additional gains, which essentially redistributes the rewards evenly to current share holders.

```
    /**
     * @dev Harvest and compound token
     */
    function _harvest() internal virtual {
        if (!isCakeStaking) {
            pancakeStaking.deposit(pid, 0);
        } else {
            pancakeStaking.leaveStaking(0);
        }

        uint256 swapAmt = IERC20(rewardToken).balanceOf(address(this));
        if (swapAmt > 0) {
            if (!isCakeStaking) {
                IERC20(rewardToken).safeIncreaseAllowance(swapRouterAddress, swapAmt);
                IPancakeRouter02(swapRouterAddress).
                    swapExactTokensForTokensSupportingFeeOnTransferTokens(
                    swapAmt,
                    0,
                    rewardTokenToTokenPath,
                    address(this),
                    block.timestamp + 120
                );
            }

            _collectFees();
            _deposit();
```

```

            emit Harvested(swapAmt);
        }
    }
```

Listing 3.2: `TargetVaultPancake::_harvest()`

We notice the collected rewards are evenly distributed to share holders. With that, it is possible for a malicious actor to launch a flashloan-assisted deposit to claim the majority of rewards, resulting in significantly less rewards to legitimate share holders. This is possible even though the `harvest()` routine can only be invoked by the permitted harvester. Note a flashbot-assisted sandwich attack can greatly facilitate this type of attacks.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of investors. It is suggested to apply necessary slippage control to ensure the shareholders can still expected gains.

Status The issue has been fixed by the following commits: `676f105` and `647b33c`.

3.3 Accommodation Of Possible Non-Compliant ERC20 Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
```

```
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.3: USDT Token **Contract**

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

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()`/`transferFrom()` as well, i.e., `safeApprove()`/`safeTransferFrom()`. We highlight that this issue is present in a number of contracts, including `TargetVault`, `TargetVaultBunny`, `TargetVaultDopple.sol`, etc.

In the following, we use the `TargetVault::retireTargetVault()` routine as an example. This routine is designed to retire a target vault and send all balances back to the feed vault. To accommodate the specific idiosyncrasy, there is a need to replace `transfer()` (line 142) with `safeTransfer()`.

```
137    /**
138     * @dev Retire target vault and send all balance back to vault
139     */
140    function retireTargetVault() public virtual onlyOwner {
141        uint256 balance = IERC20(token).balanceOf(address(this));
142        IERC20(token).transfer(feedVault, balance);

144        emit TargetVaultRetired();
145    }
```

Listing 3.4: `TargetVault::retireTargetVault()`

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

Status The issue has been fixed by this commit: cd46570.

3.4 Proper Asset Rebalance For Disabled Target Vaults

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: FeedVault
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In Feeder, the FeedVault contract is an essential one with a number key risk parameters that can be dynamically configured by privileged account, i.e., admin. While examining a specific privileged function toggleTargetVault(), we realize the current handling logic needs to be improved.

To elaborate, we show below the related toggleTargetVault() routine. It implements a basic logic in toggling the enable status of the given target vault. However, when a target vault is disabled, the funds allocated to the target vault for investment needs to retrieved back for reallocation. Such reallocation operation is not performed yet.

```
    /**
     * @dev Toggle enable status of target vault
     */
    function toggleTargetVault(uint256 _vid, bool _status) public onlyRole(ADMIN_ROLE)
        nonReentrant {
        vaultInfo[_vid].enabled = _status;
    }
```

Listing 3.5: FeedVault::toggleTargetVault()

Recommendation Revise the above toggleTargetVault() routine so that the funds are properly re-balanced for all active target vaults.

Status The issue has been fixed by this commit: cd46570.

3.5 Improper withdraw() in TargetVaultDopple

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TargetVaultDopple`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `Feeder` protocol, it supports a number of target vaults and each vault essentially acts as the corresponding investment strategy. Each target vault inherits from the base contract `TargetVault`, which defines the standard APIs, including `deposit()`, `withdraw()`, `emergencyWithdrawAll()`, `harvest()`, and `collectFees()`.

If we examine the `withdraw()` routine in the `TargetVaultDopple` contract, this routine allows for withdrawing from the target vault to the central feed vault (line 150). It needs to be clarified that the `withdraw()` expects the token amount argument, while the internal `doppleStaking.withdraw()` expects a share amount. The interface mismatch may result in an inappropriate amount of tokens being withdrawn.

```
131     /**
132      * @dev Withdraw from target vault to target vault
133      */
134     function withdraw(uint256 _amount) external virtual onlyVault {
135         if (_amount > 0) {
136             depositedBalance -= _amount;
137             doppleStaking.withdraw(address(this), pid, _amount);
138 
139             uint256 _balance = dopLP.balanceOf(address(this));
140             // Redeem dopLP back to Token
141             uint256 _tokenBefore = token.balanceOf(address(this));
142             if (_balance > 0) {
143                 dopLP.safeIncreaseAllowance(address(dopPool), _balance);
144                 dopPool.removeLiquidityOneToken(_balance, getDoppleTokenIndex(), 0,
                        block.timestamp + 600);
145             }
146             uint256 _tokenAfter = token.balanceOf(address(this));
147             uint256 _tokenAmount = _tokenAfter.sub(_tokenBefore);
148 
149             // Send token back to FeedVault
150             token.safeTransfer(feedVault, _tokenAmount);
151 
152             if (depositedBalance == 0) {
153                 cachedPricePerShare = 1e18;
154             } else {
155                 cachedPricePerShare = targetPricePerShare();
```

```
            }

            emit Withdrawed(_amount);
        }
    }
```

Listing 3.6: `TargetVaultDopple::withdraw()`

Recommendation Be consistent in using the actual token amounts for withdrawal. The above `withdraw()` routine needs to properly transform the token amount into the corresponding share amount.

Status The issue has been fixed by this commit: `abb3f9e`.

3.6 Incorrect Fee Collection in TargetVaultPancake::_collectFees()

- ID: PVE-006
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `TargetVaultPancake`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.5, the `Feeder` protocol supports a number of target vaults and each vault essentially acts as the corresponding investment strategy. Each target vault needs to implements the standard APIs, including `deposit()`, `withdraw()`, `emergencyWithdrawAll()`, `harvest()`, and `collectFees()`.

In the following, we examine the specific `collectFees()` API from the `TargetVaultPancake` contract. This routine is designed to collect fees and invest the remaining funds back for additional gains. However, it comes to our attention that the collected fee is denominated at the token for investment, not the reward token. The current implementation incorrectly uses the reward token for fee collection (line 148), which needs to be changed back to the target token.

```
    function _collectFees() internal virtual {
        uint256 _balance = IERC20(rewardToken).balanceOf(address(this));
        uint256 _fees = _balance.mul(feesBP).div(10000);

        if (_fees > 0) {
            if (autoBuyBack) {
                uint256 buyBackBefore = IERC20(buyBackToken).balanceOf(address(this));
                token.safeIncreaseAllowance(swapRouterAddress, _fees);
```

```
                IPancakeRouter02(swapRouterAddress).
                    swapExactTokensForTokensSupportingFeeOnTransferTokens(
                    _fees,
                    0,
                    tokenToBuyBackPath,
                    address(this),
                    block.timestamp + 120
                );
                uint256 buyBackAfter = IERC20(buyBackToken).balanceOf(address(this));
                uint256 buyBackAmount = buyBackAfter.sub(buyBackBefore);
                IERC20(buyBackToken).safeTransfer(feesCollector, buyBackAmount);

                emit FeesCollected(address(feesCollector), address(buyBackToken), _fees)
                    ;
            } else {
                token.safeTransfer(feesCollector, _fees);

                emit FeesCollected(address(feesCollector), address(token), _fees);
            }
        }

        _deposit();
    }
```

Listing 3.7: `TargetVaultPancake::_collectFees()` firstnumber

Recommendation Revive the above `_collectFees()` routine to use the intended investment token, instead of the reward token, for fee collection.

Status The issue has been fixed by this commit: `abb3f9e`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Feeder` protocol, the privileged admin account plays a critical role in governing and regulating the system-wide operations (e.g., `vault/strategy` addition and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the three components, i.e., `FeedVault`, and `TargetVaults`. Our analysis shows that the admin-related accounts are indeed privileged. In the following, we show representative privileged operations in the `Feeder` protocol.

```
    /**
     * @dev Update target vault
     */
    function setTargetVault(uint256 _vid, uint256 _allocPoint) public onlyRole(MANAGER_ROLE) {
        totalAllocPoint = totalAllocPoint.sub(vaultInfo[_vid].allocPoint).add(_allocPoint);
        vaultInfo[_vid].allocPoint = _allocPoint;
    }

    /**
     * @dev Toggle enable status of target vault
     */
    function toggleTargetVault(uint256 _vid, bool _status) public onlyRole(ADMIN_ROLE) nonReentrant {
        vaultInfo[_vid].enabled = _status;
    }

    /**
     * @dev Update multiple target vaults alloc point
     */
    function setAllocPoints(uint256[] memory _allocPoints) public onlyRole(MANAGER_ROLE) nonReentrant {
        require(_allocPoints.length == vaultInfo.length, "FeedVault(setAllocPoints): number of vaults is incorrect");
        uint256 length = vaultInfo.length;
        for (uint256 i = 0; i < length; i++) {
            setTargetVault(i, _allocPoints[i]);
        }
    }
```

Listing 3.8: Various `Setters` in `FeedVault`

We emphasize that the privilege assignment with various contracts is necessary and required for proper protocol operations. However, it is worrisome if the `admin` account is not governed by a `DAO`-like structure.

We point out that a compromised `admin` account would allow the attacker to add a malicious `vault` or change other settings to steal funds in current protocol, which directly undermines the assumption of the `Feeder` protocol.

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 confirmed and partially mitigated with a multi-sig account to regulate the admin privileges.

3.8 Improved Emergency Withdrawal in TargetVaultBunny

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

Description

As mentioned in Section 3.5, the `Feeder` protocol supports a number of target vaults and each target vault needs to implements the standard APIs, including `deposit()`, `withdraw()`, `emergencyWithdrawAll()`, `harvest()`, and `collectFees()`. In the following, we examine the `emergencyWithdrawAll()` routine from the `TargetVaultBunny` contract.

As the name indicates, this routine is designed to withdraw all deposited balances back to the feed vault. However, the current implementation in a number of target vaults does not properly return back the funds back to the feed vault. Mover, this routine also needs to properly reset `depositedBalance` back to 0. The affected target vaults include `TargetVaultAutoFarm`, `TargetVaultACrypto`, `TargetVaultBunny`, and `TargetVaultPancake`.

```
    /**
     * @dev Withdraw all deposited balance back to target vault
     */
    function emergencyWithdrawAll() external virtual onlyOwner {
        bunnyVault.withdrawAll();

        emit EmergencyWithdrawed();
    }
```

Listing 3.9: `TargetVaultBunny::emergencyWithdrawAll()`

Recommendation Revised the affected target vaults to properly return funds back to the feed vault and reset `depositedBalance` back to 0.

Status The issue has been fixed by this commit: `38095f1`.

3.9 Force Investment Risk in TargetVaultDopple

- ID: PVE-009
- Severity: High
- Likelihood: High
- Impact: High
- Target: TargetVaultDopple
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Feeder protocol is a decentralized DeFi aggregator for diversified yield generation on Binance Smart Chain (BSC). The investment subsystem is inspired from the yearn.finance framework and thus shares similar architecture with vaults, and strategies.

While examining the TargetVaultDopple implementation, we notice a potential force investment risk that has been exploited in earlier hacks, e.g., yDAI [13] and BT.Finance [1]. To elaborate, we show blow the related TargetVaultDopple vault.

Specifically, new target vault contracts have been designed and implemented to invest VC assets, harvest growing yields, and return any gains, if any, to the investors. In order to have a smooth investment experience, the target vault contract has a dedicated function, i.e., deposit(), that can be invoked to kick off the investment.

```
    /**
     * @dev Deposit from target vault to target vault
     */
    function deposit() external virtual onlyVault {
        _deposit();
        cachedPricePerShare = targetPricePerShare();
    }

    function _deposit() internal virtual {
        uint256 _balance = token.balanceOf(address(this));

        if (_balance > 0) {
            uint256 _dopBefore = dopLP.balanceOf(address(this));

            // Deposit to Belt to get Dopple
            token.safeIncreaseAllowance(address(dopPool), _balance);
            uint256[] memory amounts = new uint256[](dopPoolLength);
            amounts[getDoppleTokenIndex()] = _balance;
            dopPool.addLiquidity(amounts, 0, block.timestamp + 600);
            uint256 _dopAfter = dopLP.balanceOf(address(this));
            _balance = _dopAfter.sub(_dopBefore);

            depositedBalance += _balance;
            dopLP.safeIncreaseAllowance(address(doppleStaking), _balance);
            doppleStaking.deposit(address(this), pid, _balance);
```

```
        emit Deposited(_balance);
    }
}
```

Listing 3.10: `TargetVaultDopple::deposit()`

It comes to our attention that the `deposit()` function is not guarded or can be invoked by any one to initiate the investment. If the configured strategy blindly invests the deposited funds into an imbalanced `Dopple` pool, the strategy will not result in a profitable investment. In fact, earlier incidents (yDAI and BT hacks [13, 1]) have prompted the need of a guarded call to the investment function. For the very same reason, we argue for the guarded call to block potential flashloan-assisted attacks. One mitigation will enforce certain lockup period for investment.

Recommendation Develop the lockup time period to block unwanted flashloan attacks. And take extra care in ensuring the vault assets will not be blindly deposited into a faulty target vault (that is currently not making any profit).

Status The issue has been fixed by this commit: `e9b7396`.

3.10 Improper balanceOf() in TargetVaultDopple/TargetVaultACrypto

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In previous sections, we have examined a number of standard APIs, i.e., `deposit()`, `withdraw()`, `emergencyWithdrawAll()`, `harvest()`, and `collectFees()`. Next, we examine another commonly-defined function `balanceOf()`. Note that this function is used to return back the token balance that is being invested in the target vault.

To elaborate, we show below the related `balanceOf()` routine from the `TargetVaultDopple` contract. It implements a rather straightforward logic in computing the balance of target vault plus deposited balance (line 232). It comes to our attention that `availableBalance()` returns token balance denominated at the target token for investment, while `vaultBalance()` returns the share amount held in the staking contract, i.e., `doppleStaking`. In other words, the share amount needs to properly transformed back to the amount of investment tokens. The same issue is also applicable to another target vault, i.e., `TargetVaultACrypto`.

```
    /**
     * @dev Balance of target vault plus deposited balance
     */
    function balanceOf() public view virtual returns (uint256) {
        return availableBalance().add(vaultBalance());
    }
```

Listing 3.11: TargetVaultDopple::balanceOf()

Recommendation Revised the above `balanceOf()` of affected target vaults to return the right balance.

Status The issue has been fixed by this commit: `abb3f9e`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Feeder` protocol. The audited system presents a unique addition to current DeFi offerings by offering a decentralized DeFi aggregator for diversified yield generation on `Binance Smart Chain (BSC)`. 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] BT Finance. BT.Finance Exploit Analysis Report. https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.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.