SMART CONTRACT AUDIT REPORT

for

Feeder Lending

Prepared By: Yiqun Chen

Hangzhou, China
December 31, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	December 31, 2021	Shulin Bie	Final Release
1.0-rc	December 31, 2021	Shulin Bie	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 Feeder Lending, 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 Lending

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. Feeder Lending, as an important part of Feeder Finance, is a permission-less decentralized protocol that provides lending and borrowing services through innovatively introducing an auction mechanism. It is an important component in the Feeder Finance ecosystem.

Table 1.1: Basic Information of Feeder Lending

Item	Description
Target	Feeder Lending
Website	https://feeder.finance/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 31, 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/lending-contracts.git (06ee0c2)

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

- https://github.com/FeederFinance/lending-contracts.git (d799469)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further 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) [9], 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 1.3: 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

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 `Feeder Lending` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	4	■■■■
Low	3	■■■
Informational	0	
Total	9	

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, 4 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Feeder Lending Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Logic Of viewBidsPerOffer()	Business Logic	Fixed
PVE-002	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Mitigated
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Low	Duplicate Vault Detection and Prevention	Business Logic	Fixed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-006	High	Improper Logic Of VaultKeeperFeed::deposit()	Business Logic	Fixed
PVE-007	High	Potential Repeated acceptBid() For The Same Offer	Business Logic	Fixed
PVE-008	Medium	Improper Logic Of liquidateOnBehalf()	Business Logic	Fixed
PVE-009	Medium	Potential Sandwich/MEV Attack In liquidate()	Time and State	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improper Logic Of viewBidsPerOffer()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: DealManager/FeedLoan
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

By design, Feeder Lending protocol implements an auction mechanism to provide lending, borrowing and liquidating services. When the borrower intends to use his assets as collateral to borrow other assets, he should create an offer for his assets. Others can bid for the offer by providing the type of the loanable asset, amount, interest rate, time duration, etc. Once the borrower accepts one of the bids, he will receive the bid related assets. If the borrower cannot repay the borrowed assets on time, his collateral will be liquidated. Feeder Lending protocol also provides a series of query routines for the user. In particular, one routine, i.e., DealManager::viewBidsPerOffer(), is designed to query the bids' information of an offer. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the DealManager contract. The DealManager ::viewBidsPerOffer() routine has three input parameters: the first _offerId parameter specifies the queried offer identification, the second _cursor parameter specifies the start index of the offerBids [_offerId] array, and the third _size parameter indicates the number of the offerBids[_offerId] array element starting from _cursor. However, we notice the returned _values copies from 0 of the offerBids[_offerId] array rather than _cursor (line 510). Given this, we suggest to improve the implementation as below: _values[i] = offerBids[_offerId][_cursor + i] (line 510).

```
497     function viewBidsPerOffer(
498         uint256 _offerId,
499         uint256 _cursor,
```

```
        uint256 _size
    ) external view returns (OfferBidInfo[] memory, uint256) {
        uint256 _length = _size;
        uint256 _bidsLength = offerBids[_offerId].length;
        if (_length > _bidsLength - _cursor) {
            _length = _bidsLength - _cursor;
        }

        OfferBidInfo[] memory _values = new OfferBidInfo[](_length);
        for (uint256 i = 0; i < _length; i++) {
            _values[i] = offerBids[_offerId][i];
        }

        return (_values, _cursor + _length);
    }
```

Listing 3.1: `DealManager::viewBidsPerOffer()`

Note other routines, i.e., `DealManager::viewBidsPerBidder()`, `DealManager::viewOffers()`, `DealManager::viewOffersByCollateral()`, `FeedLoan::viewLoans()`, `FeedLoan::viewLoansPerLender()`, and `FeedLoan::viewLoansPerBorrower()`, share the same issue.

Recommendation Correct the implementation of above-mentioned routines.

Status The issue has been addressed by the following commit: `83b672f`.

3.2 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As section 3.1 mentioned, `Feeder Lending` protocol implements an auction mechanism to provide lending, borrowing and liquidating services. By design, the borrower's collateral assets and the lender's assets will be transferred between the internal contracts of the protocol. This is reasonable under the assumption that these transfers will always result in full transfer.

```
    function createOffer(
        address _collateral,
        uint256 _collateralAmount,
        bool _useVault,
        uint256 _vaultId
    ) external nonReentrant {
```

```
        ...

        // Transfer collateral to contract
        IERC20(_collateral).safeTransferFrom(address(msg.sender), address(this),
            _collateralAmount);

        // Emit OfferCreated event
        emit OfferCreated(_offerId, address(msg.sender), _collateral, _collateralAmount,
             _useVault, _vaultId);
    }
```

Listing 3.2: `DealManager::createOffer()`

```
    function startLoan(
        address _lender,
        address _asset,
        uint256 _assetAmount,
        address _borrower,
        address _collateral,
        uint256 _collateralAmount,
        uint256 _duration,
        uint256 _intRateBP,
        bool _intProRated,
        bool _useVault,
        uint256 _vaultId
    ) external onlyDealManager returns (uint256) {
        // Transfer collateral from DealManager to this contract
        IERC20(_collateral).safeTransferFrom(msg.sender, address(this),
            _collateralAmount);

        // Transfer lending asset to borrower
        IERC20(_asset).safeTransferFrom(msg.sender, _borrower, _assetAmount);

        ...
    }
```

Listing 3.3: `FeedLoan::startLoan()`

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 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 routines related to token transfer.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming 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. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Feeder Lending`. In `Feeder Lending`, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better 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 The issue has been mitigated by the following commit: `20d80a6`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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 analyze 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 3.4: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `DealManager::acceptBid()` routine as an example. In this routine, `approve()` is executed to assign approval to the `FeedLoan` contract. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function acceptBid(uint256 _offerId, uint256 _bidId) external nonReentrant {
        ...

        // Transfer asset and collateral to loan manager and open a loan and mint nft
        IERC20(_offer.collateral).approve(address(feedLoan), _offer.collateralAmount);
        IERC20(_bid.asset).approve(address(feedLoan), _bid.amount);
        uint256 _loanId = IFeedLoan(feedLoan).startLoan(
            _bid.account,
            _bid.asset,
            _bid.amount,
            _offer.maker,
            _offer.collateral,
            _offer.collateralAmount,
            _bid.duration,
            _bid.intRateBP,
            _bid.intProRated,
            _offer.useVault,
            _offer.vaultId
        );

        // Set loan's ID to offer info
        _offer.loanId = _loanId;

        // Set accepted bid's ID to offer info
        _offer.bidId = _bid.id;

        if (_bid.allowLiquidator) IFeedLoan(feedLoan).setAllowLiquidator(_loanId, _bid.
            allowLiquidator);

        // Emit OfferBidAccepted event
        emit OfferBidAccepted(_offerId, _bidId);
    }
```

Listing 3.5: `DealManager::acceptBid()`

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. Similarly, there is a safe version of approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom().

In the following, we show the FeedLoan::payback() routine. If the USDT token is supported as _loan.collateral, the unsafe version of IERC20(_loan.collateral).transfer(loanBorrower[_loanId], _withdrawnAmount) may revert as there is no return value in the USDT token contract's transfer() implementation (but the IERC20 interface expects a return value). We may intend to replace transfer() with safeTransfer().

```
    function payback(uint256 _loanId) external nonReentrant {
        ...

        // If collateral is in vault
        if (loanVault[_loanId].useVault) {
            uint256 _withdrawnAmount = _withdrawFromVault(_loanId);
            // Transfer collateral to borrower
            IERC20(_loan.collateral).transfer(loanBorrower[_loanId], _withdrawnAmount);
        } else {
            // Transfer collateral to borrower
            IERC20(_loan.collateral).transfer(loanBorrower[_loanId], _loan.
                collateralAmount);
        }

        // Emit LoanRepaid event
        emit LoanRepaid(_loanId, _repaymentAmount, _loan.earnedInterest);
    }
```

Listing 3.6: FeedLoan::payback()

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

Status The issue has been addressed by the following commit: 98586b1.

3.4 Duplicate Vault Detection and Prevention

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VaultController`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Feeder Lending` protocol, the `VaultController` contract plays a vault proxy role, which maintains the relationship of the staking token and vault address. In current implementation, there are a number of concurrent vaults and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account). To accommodate these new vaults, the design has the necessary mechanism in place that allows for dynamic additions of new vaults.

The addition of a new vault is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks to avoid duplicate vault addition. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong vault introduction from human omissions.

```
    function add(IERC20 _token, address _vault) public onlyOwner nonReentrant {
        // Store new vault info in storage
        vaultInfo.push(VaultInfo({token: _token, vault: _vault}));

        // Store vault address mapping to vid
        adddressToVid[_vault] = vaultInfo.length - 1;

        // Emit VaultAdded event
        emit VaultAdded(adddressToVid[_vault], address(_token), _vault);
    }
```

Listing 3.7: `VaultController::add()`

Recommendation Add necessary sanity checks to avoid duplicate vault addition.

Status The issue has been addressed by the following commit: `219b006`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

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

```
    function setLenderFeeBP(uint256 _lenderFeeBP) external onlyOwner nonReentrant {
        require(_lenderFeeBP >= 0, "SetLenderFeeBP: must greater than or equal to zero")
            ;

        lenderFeeBP = _lenderFeeBP;

        emit LenderFeeBPChanged(lenderFeeBP);
    }

    function setBorrowerFeeBP(uint256 _borrowerFeeBP) external onlyOwner nonReentrant {
        require(_borrowerFeeBP >= 0, "SetBorrowerFeeBP: must greater than or equal to
            zero");

        borrowerFeeBP = _borrowerFeeBP;

        emit BorrowerFeeBPChanged(borrowerFeeBP);
    }

    function setLenderFeeCollector(address _lenderFeeCollector) external onlyOwner
        nonReentrant {
        require(_lenderFeeCollector != address(0), "SetLenderFeeCollector: Cannot be
            zero address");

        lenderFeeCollector = _lenderFeeCollector;

        emit LenderFeeCollectorChanged(lenderFeeCollector);
    }

    function setBorrowerFeeCollector(address _borrowerFeeCollector) external onlyOwner
        nonReentrant {
        require(_borrowerFeeCollector != address(0), "SetBorrowerFeeCollector: Cannot be
             zero address");

        borrowerFeeCollector = _borrowerFeeCollector;

```

```
        emit BorrowerFeeCollectorChanged(borrowerFeeCollector);
    }
```

Listing 3.8: `FeedLoan`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `Feeder Lending` design.

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.

3.6 Improper Logic Of VaultKeeperFeed::deposit()

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `VaultKeeperFeed`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

By design, the `VaultKeeperFeed` contract is the main entry for interaction with the `FeedVault` contract. In particular, one entry routine, i.e., `deposit()`, accepts the deposits of the supported `token` assets and then deposits the assets to `FeedVault` (specified by the `vaultAddress`). While examining its logic, we notice the share calculation is incorrect.

To elaborate, we show below the related code snippet of the `VaultKeeperFeed` contract. In the `deposit()` function, the following statement is executed to calculate the share for the deposit: `_shares = (_amount.mul(totalShares)).div(_before)` (line 90). We notice `totalShares` represents the total shares held by all the depositors of the `VaultKeeperFeed` contract, which is corresponding to the total balance of the `token` deposited to the `VaultKeeperFeed` contract. However, `_before` stores the total balance of the `token` deposited to the `vaultAddress` rather than the `VaultKeeperFeed` contract(line 70), which directly undermines the `deposit()` design.

```
68    function deposit(uint256 _amount) external nonReentrant {
69        // Balance before deposit
70        uint256 _before = balance();
```

```
        // Transfer token from sender
        token.safeTransferFrom(msg.sender, address(this), _amount);

        // Deposit token to target vault
        token.approve(vaultAddress, _amount);
        IFeedVault(vaultAddress).deposit(_amount);

        // Balance after deposited
        uint256 _after = balance();

        // Additional check for deflationary tokens
        _amount = _after.sub(_before);

        // Calculate shares to be added
        uint256 _shares = 0;
        if (totalShares == 0) {
            _shares = _amount;
        } else {
            _shares = (_amount.mul(totalShares)).div(_before);
        }

        // Get user info from storage
        UserInfo storage user = userInfo[address(msg.sender)];

        // Add shares to total shares
        totalShares = totalShares.add(_shares);

        // Add shares to user info
        user.shares = user.shares.add(_shares);

        // Emit Deposited event
        emit Deposited(_amount);
    }
```

Listing 3.9: `VaultKeeperFeed::deposit()`

Recommendation Correct the implementation of the `deposit()` routine as above-mentioned.

Status The issue has been addressed by the following commit: `a8fba4d`.

3.7 Potential Repeated acceptBid() For The Same Offer

- ID: PVE-007
- Severity: High
- Likelihood: High
- Impact: High
- Target: `DealManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, when the borrower intends to use his assets as collateral to borrow other assets, he should create an `offer` for his assets with the call to `createOffer()`, while others can bid for the `offer` with the call to `offerBid()` by providing the type of the loanable asset, amount, interest rate, time duration, etc. After that, the `acceptBid()` is called by the borrower to accept one of the bids that he is interested in. By doing so, he can borrow the bid related assets. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `DealManager` contract. In the `acceptBid()` function, this requirement of `require(_offer.maker == address(msg.sender), "AcceptBid: account not maker")` (line 408) is executed to ensure only the owner of the `offer` (specified by the input `_offerId` parameter) can accept the bid, and the next requirement of `require(_bid.status == OfferBidStatus.Open, "AcceptBid: bid is already canceled")` (line 409) is executed to ensure the validity of the bid (specified by the input `_bidId` parameter). However, we notice it doesn't check whether the `offer` has accepted a bid before, which may be exploited by a malicious actor to accept other bids for the same `offer` again and again. Given this, we suggest to add necessary sanity check at the beginning of the `acceptBid()` function to prevent this case as follows: `require(_offer.status == OfferStatus.Pending)`.

```
400     function acceptBid(
401         uint256 _offerId,
402         uint256 _bidId,
403         uint256 _safeDuration
404     ) external nonReentrant {
405         require(_offerId < totalOffersCount, "AcceptBid: offer not found");
406         Offer storage _offer = offers[_offerId];
407         OfferBidInfo storage _bid = offerBids[_offer.id][_bidId];
408         require(_offer.maker == address(msg.sender), "AcceptBid: account not maker");
409         require(_bid.status == OfferBidStatus.Open, "AcceptBid: bid is already canceled"
            );
410         require(block.timestamp > _bid.updatedAt + _safeDuration, "AcceptBid: bid is
            recently updated");
411
412         // Set offer status to closed
413         _offer.status = OfferStatus.Closed;
```

```
        // Set offer taker to lender address
        _offer.taker = _bid.account;

        // Set bid status to Accepted
        _bid.status = OfferBidStatus.Accepted;

        // Reduce total active offers counter
        totalActiveOffers -= 1;

        // Reduce offer bids count
        offerActiveBidsCount[_offerId] -= 1;

        // Reduce bidder bids count
        bidderActiveBidsCount[_offerId][_bid.account] -= 1;

        // Transfer asset and collateral to loan manager and open a loan and mint nft
        IERC20(_offer.collateral).safeApprove(address(feedLoan), 0);
        IERC20(_offer.collateral).safeApprove(address(feedLoan), _offer.collateralAmount
            );
        IERC20(_bid.asset).safeApprove(address(feedLoan), 0);
        IERC20(_bid.asset).safeApprove(address(feedLoan), _bid.amount);
        uint256 _loanId = IFeedLoan(feedLoan).startLoan(
            _bid.account,
            _bid.asset,
            _bid.amount,
            _offer.maker,
            _offer.collateral,
            _offer.collateralAmount,
            _bid.duration,
            _bid.intRateBP,
            _bid.intProRated,
            _offer.useVault,
            _offer.vaultId
        );

        ...
    }
```

Listing 3.10: `DealManager::acceptBid()`

Recommendation Add the above-mentioned sanity check inside the `acceptBid()` routine.

Status The issue has been addressed by the following commit: `7b290fa`.

3.8 Improper Logic Of liquidateOnBehalf()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FeedLoan`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, if a borrower has not capability to repay his borrowed assets on time, his collateral assets will be liquidated by others. In particular, one entry routine, i.e., `liquidateOnBehalf()`, allows others to liquidate the borrower's collateral assets on behalf of the lender. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `FeedLoan` contract. By design, during liquidating the borrower's collateral assets, the part of the repaid assets (specified by the `_lenderFee` and `_borrowerFee` variables) will be respectively transferred to `lenderFeeCollector` (line 510) and `borrowerFeeCollector` (line 513) as transaction fee. However, we notice `_lenderFee` is incorrectly transferred to `borrowerFeeCollector`, which directly undermines the original intention of design. Given this, we suggest to correct the implementation as below: `IERC20(_loan.asset).safeTransferFrom(address(msg.sender), address(borrowerFeeCollector), borrowerFeeCollector)` (line 513).

```
465     function liquidateOnBehalf(uint256 _loanId) external nonReentrant {
466         // Fetch loan from storage
467         Loan storage _loan = loans[_loanId];
468
469         // Check whether lender allow liquidator to liquidate loan
470         require(_loan.allowLiquidator, "FeedLoan(liquidateOnBehalf): Liquidator is not
                allowed");
471
472         // Loan should not be repaid, liquidated or completed
473         require(_loan.status == LoanStatus.Active, "FeedLoan(liquidateOnBehalf): Loan is
                 not active");
474
475         // Current block time is greater than loan starting time plus duration
476         require(block.timestamp > _loan.startTime.add(_loan.duration), "FeedLoan(
                liquidateOnBehalf): Loan is not overdue");
477
478         uint256 _interestDue = _loan.maxRepayment.sub(_loan.assetAmount);
479         if (_loan.intProRated) {
480             _interestDue = _calcInterestDue(
481                 _loan.assetAmount,
482                 _loan.intRateBP,
483                 _loan.duration,
484                 block.timestamp.sub(_loan.startTime),
```

```
                _loan.intProRated
            );
        }

        uint256 _lenderFee = _interestDue.mul(lenderFeeBP).div(10000);
        uint256 _borrowerFee = _interestDue.mul(borrowerFeeBP).div(10000);

        // If fees controller is set, adjust lender and borrower fees accordingly
        if (feesController != address(0)) {
            // Calculate and set lender & borrower fee by using discount basis point from FeesController
            _lenderFee = _lenderFee.sub(_lenderFee.mul(IFeesController(feesController).getDiscountBP(loanLender[_loanId])).div(10000));
            _borrowerFee = _borrowerFee.sub(
                _borrowerFee.mul(IFeesController(feesController).getDiscountBP(address(msg.sender))).div(10000)
            );
        }

        uint256 _repaymentAmount = _loan.assetAmount.add(_interestDue).sub(_lenderFee.add(_borrowerFee));

        // Transfer principal including interest from liquidator to contract
        uint256 _assetAmount = _safeDeflationaryTransfer(address(msg.sender), address(this), _loan.asset, _repaymentAmount);

        // Update loan asset amount in case token is deflationary
        _loan.assetAmount = _assetAmount.sub(_interestDue.sub(_lenderFee.add(_borrowerFee)));

        // Transfer lender's fee
        IERC20(_loan.asset).safeTransferFrom(address(msg.sender), address(lenderFeeCollector), _lenderFee);

        // Transfer borrower's fee
        IERC20(_loan.asset).safeTransferFrom(address(msg.sender), address(borrowerFeeCollector), _lenderFee);

        ...
    }
```

Listing 3.11: `FeedLoan::liquidateOnBehalf()`

Recommendation Correct the above implementation in `liquidateOnBehalf()`.

Status The issue has been addressed by the following commit: `31bd742`.

3.9 Potential Sandwich/MEV Attack In liquidate()

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FeedLoan`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, if a borrower has not capability to repay his borrowed assets on time, his collateral assets will be liquidated by others. In particular, one entry routine, i.e., `liquidate()`, allows the lender to liquidate the borrower's collateral assets by himself. While examining its logic, we observe there is a vulnerability that can be exploited by the lender to decrease transaction fee.

To elaborate, we show below the related code snippet of the `FeedLoan` contract. By design, the `_lenderFee` can be discounted (line 580) according to the amount of the `token` (specified by the `token` variable of the `FeesController` contract) held by the lender. If the amount of the `token` is larger than `50000000000000000` (line 66), the `_lenderFee` will be discounted to zero. By borrowing a huge of the `token` through `flashloan` before the call to `liquidate()`, the lender can reduce the transaction fee along with receiving more collateral assets than normal.

```
540     function liquidate(uint256 _loanId) external nonReentrant {
541         // Fetch loan from storage
542         Loan storage _loan = loans[_loanId];
543 
544         // Loan should not be repaid, liquidated or completed
545         require(_loan.status == LoanStatus.Active, "FeedLoan(liquidate): Loan is not
                active");
546 
547         // Current block time is greater than loan starting time plus duration
548         require(block.timestamp > _loan.startTime.add(_loan.duration), "FeedLoan(
                liquidate): Loan is not overdue");
549 
550         // Get loan's lender
551         address _lender = loanLender[_loanId];
552 
553         // Only lender is allowed to liquidate the loan
554         require(_lender == msg.sender, "FeedLoan(liquidate): Sender is not lender");
555 
556         // Burn NFT
557         _burn(_loanId);
558 
559         // Set loan status
560         _loan.status = LoanStatus.Liquidated;
561 
562         // Update total number of active loans
```

```
        totalActiveLoans -= 1;

        uint256 _returnAmount = 0;
        // If collateral is in vault
        if (loanVault[_loanId].useVault) {
            // Collateral balance AFTER withdraw
            _returnAmount = _withdrawFromVault(_loanId);
        } else {
            _returnAmount = _loan.collateralAmount;
        }

        uint256 _lenderFee = _returnAmount.mul(lenderFeeBP).div(10000);
        uint256 _borrowerFee = _returnAmount.mul(borrowerFeeBP).div(10000);

        // If fees controller is set, adjust lender and borrower fees accordingly
        if (feesController != address(0)) {
            // Calculate and set lender & borrower fee by using discount basis point
                from FeesController
            _lenderFee = _lenderFee.sub(_lenderFee.mul(IFeesController(feesController).
                getDiscountBP(loanLender[_loanId])).div(10000));
            _borrowerFee = _borrowerFee.sub(
                _borrowerFee.mul(IFeesController(feesController).getDiscountBP(
                    loanBorrower[_loanId])).div(10000)
            );
        }

        // Transfer lender's fee
        IERC20(_loan.collateral).safeTransfer(lenderFeeCollector, _lenderFee);

        // Transfer borrower's fee
        IERC20(_loan.collateral).safeTransfer(borrowerFeeCollector, _borrowerFee);

        // Calculate amount of collateral to return to lender after fees
        _returnAmount = _returnAmount.sub(_lenderFee).sub(_borrowerFee);

        // Tranfer collateral to lender
        IERC20(_loan.collateral).safeTransfer(_lender, _returnAmount);

        // Emit LoanLiquidated event
        emit LoanLiquidated(_loanId, _returnAmount);
    }
```

Listing 3.12: `FeedLoan::liquidate()`

```
    function getDiscountBP(address _user) external view returns (uint256) {
        // Set default discount basis point to zero
        uint256 _discountBP = 0;

        // Get user balance of a token
        uint256 _balance = IERC20(token).balanceOf(_user);

        // Get total supply of a token
        uint256 _totalSupply = IERC20(token).totalSupply();
```

```

        // If balance or total supply is 0 return 0
        if (_balance == 0  _totalSupply == 0) return _discountBP;

        // Compute user shares based token holding balance over total supply
        uint256 _shares = _balance.mul(1e18).div(_totalSupply);

        if (_shares < 500000000000000) {
            // Shares < 0.05%
            _discountBP = 0;
        } else if (_shares >= 500000000000000 && _shares < 1000000000000000) {
            // Shares >= 0.05% and < 0.1%
            _discountBP = 1500;
        } else if (_shares >= 1000000000000000 && _shares < 10000000000000000) {
            // Shares >= 0.1% and < 1%
            _discountBP = 2500;
        } else if (_shares >= 10000000000000000 && _shares < 30000000000000000) {
            // Shares >= 0.1% and < 0.3%
            _discountBP = 5000;
        } else if (_shares >= 30000000000000000 && _shares < 50000000000000000) {
            // Shares >= 0.05% and < 0.1%
            _discountBP = 7500;
        } else if (_shares >= 50000000000000000) {
            // Shares >= 5%
            _discountBP = 10000;
        }

        // Return discount basis point
        return _discountBP;
    }
```

Listing 3.13: `FeedLoan::liquidateOnBehalf()`

Note the `liquidateOnBehalf()` routine shares the same issue.

Recommendation Develop an effective mitigation to the above MEV attack. One possible mitigation is to ensure the liquidator is a `EOA` account.

Status The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the `Feeder Lending` design and implementation. `Feeder Lending`, as an important part of `Feeder Finance`, is a permission-less decentralized protocol that provides lending and borrowing services through innovatively introducing an auction mechanism. It enriches the `Feeder Finance` ecosystem. 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] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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

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

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