SMART CONTRACT AUDIT REPORT

for

Penpie Protocol

Prepared By: Xiaomi Huang

PeckShield

October 3, 2024

Document Properties

Client	Penpie
Title	Smart Contract Audit Report
Target	Penpie
Version	1.0.1
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0.1	October 3, 2024	Xuxian Jiang	Post-Final Release #1
1.0	September 18, 2024	Xuxian Jiang	Final Release
1.0-rc	September 15, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Penpie` 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 could potentially be improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Penpie

`Penpie` is a multi-chain `DeFi` product providing yield and `veTokenomics` boosting services for `Pendle Finance`. The protocol enables `PENDLE` holders to earn rewards for their active user engagement, provides `Pendle Finance` voters with cost-effective voting power, and allows liquidity providers to gain increased `APR%` without needing to lock `PENDLE` tokens as `vePENDLE`. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Penpie

Item	Description
Name	Penpie
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 3, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the repository has a number of contracts and this audit covers the list of contracts outlined in the given audit scope file (with the calculated `md5` hash sum of `ccbaff3e6814bc933cc2838c79816c95`).

- https://github.com/magpiexyz/penpie-contracts.git (664a2e22)

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

- https://github.com/magpiexyz/penpie-contracts.git (602ee33, fc860a8)

1.2 About PeckShield

PeckShield Inc. [14] 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 [13]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	4	■■■■
Informational	1	■
Total	6	

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 may involve 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 need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Emergency Withdrawal in PenpieBribeRewardDistributor	Business Logic	Resolved
PVE-002	Low	Improper Bribe Token Validation in PenpieBribeManager	Business Logic	Resolved
PVE-003	Medium	Suggested Adherence of Checks-Effects-Interactions	Time & States	Resolved
PVE-004	Low	Revised _onlyMasterChef() Modifier Logic in ARBRewarder	Coding Practices	Resolved
PVE-005	Informational	Lack of Slippage Control in BuyBackBurnProvider	Business Logic	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Improved totalPendleFee Enforcement in PendleStakingBaseUpg	Coding Practices	Resolved

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 Revisited Emergency Withdrawal in PenpieBribeRewardDistributor

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PenpieBribeRewardDistributor
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The Penpie protocol has a core PenpieBribeRewardDistributor contract, which is used for distributing rewards from voting. While examining the emergency withdrawal logic, we notice current implementation can be improved.

```
    function emergencyWithdraw(address _token, address _receiver) external onlyOwner {
        if (_token == bribeManager) {
            address payable recipient = payable(_receiver);
            recipient.transfer(address(this).balance);
        } else {
            IERC20(_token).safeTransfer(
                _receiver,
                IERC20(_token).balanceOf(address(this))
            );
        }
    }
```

Listing 3.1: PenpieBribeRewardDistributor::emergencyWithdraw()

To elaborate, we show above the implementation for the emergency withdrawal. It has a rather straightforward logic in withdrawing funds from the PenpieBribeRewardDistributor contract. It comes to our attention that it uses bribeManager to indicate the native token withdrawal, which is suggested to make use of the defined NATIVE constant instead.

Similarly, the `PenpieBribeManager` contract can be improved in its `manualClaimFees()` routine by only claiming the `unCollectedFee` amount, not the full amount.

Recommendation Improve the above logic to ensure the native token withdrawal is properly intended.

Status The issue has been fixed by the following commits: `a84ef74` and `88e5e4c`.

3.2 Improper Bribe Token Validation in PenpieBribeManager

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `PenpieBribeManager`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [3]

Description

As part of the `veTokenomics` boosting services, `Penpie` manages possible bribes in `PenpieBribeManager`. In the process of examining the bribe management, we notice current implementation to validate the input bribe token can be improved.

In the following, we show the implementation of an example routine, i.e., `addBribeERC20ToEpoch()`. As the name indicates, this routine is used to add certain bribe for the voters in the intended epoch. However, the bribe token is validated as `if (!allowedToken[_token] && _token != NATIVE) revert InvalidBribeToken()` (line 277). This validation is incorrect as `_token != NATIVE` should be removed. The reason is that it is an `ERC20`-based bribe token, not the native gas token. Similarly, the `_addBribeERC20()` routine from the same contract can be similarly improved.

```
274     function addBribeERC20ToEpoch(uint256 _epoch, uint256 _pid, address _token, uint256
            _amount, bool forVePendle) external nonReentrant whenNotPaused onlyOperator {
275         if (_epoch < exactCurrentEpoch() - 1) revert InvalidEpoch();
276         if (_pid >= pools.length  !pools[_pid]._active) revert InvalidPool();
277         if (!allowedToken[_token] && _token != NATIVE) revert InvalidBribeToken();
278         ...
279     }
```

Listing 3.2: `PenpieBribeManager::addBribeERC20ToEpoch()`

Recommendation Improve the above-mentioned routines to properly validate the bribe token.

Status The issue has been fixed by the following commit: `9da9c86`.

3.3 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [16] exploit, and the `Uniswap/Lendf.Me` hack [15].

We notice occasions where the `checks-effects-interactions` principle is violated. Using the `ARBRewarder` as an example, the `_calculateAndSendARB()` function (see the code snippet below) is provided to externally call a rewarder contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 129) starts before effecting the update on internal state (line 132), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same function. Note that there is no harm that may be caused to current protocol. However, it is still suggested to follow the known `checks-effects-interactions` best practice.

```
111    function _calculateAndSendARB(address _stakingToken, address _rewarder) internal {
112        if (_rewarder == address(0)) {
113            return;
114        }
115        PoolInfo storage pool = tokenToPoolInfo[_stakingToken];
116        if (!pool.isActive   pool.ARBPerSec == 0) {
117            return;
118        }
119
120        uint256 multiplier = block.timestamp - pool.lastRewardTimestamp;
121        if (block.timestamp >= pool.endTimestamp) {
122            pool.isActive = false;
123            multiplier = pool.endTimestamp - pool.lastRewardTimestamp;
124        }
125        uint256 rewardAmount = (multiplier * pool.ARBPerSec);
126        rewardAmount = Math.min(rewardAmount, ARB.balanceOf(address(this)));
```

```

        ARB.approve(_rewarder, rewardAmount);
        IBaseRewardPool(_rewarder).queueNewRewards(rewardAmount, address(ARB));

        emit ARBRewadsSent(_stakingToken, _rewarder, rewardAmount, pool.
            lastRewardTimestamp, pool.ARBPerSec);
        pool.lastRewardTimestamp = block.timestamp;
    }
```

Listing 3.3: ARBRewarder::_calculateAndSendARB()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. Note other routines `MasterPenpie::_withdraw()` and `VLPenpie::transferPenalty()` can be similarly improved.

Status The issue has been fixed by the following commits: `eaaadce`, `2f7adcc`, and `3aee898`.

3.4 Revised _onlyMasterChef() Modifier Logic in ARBRewarder

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High

- Target: `ARBRewarder`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Penpie` protocol has a key `ARBRewarder` contract, which is used for distributing `ARB` rewards for participating users. While examining the pool management for reward distribution, we notice the pool-specific `MasterChef` validation can be improved.

In the following, we show the implementation of the affected `_onlyMasterChef()` modifier. The purpose is to validate the calling user to be the pool-specific `MasterChef`. However, when the given `_stakingToken` is validated, the resulting `masterChef` could be `address(0)`, which successfully passes this modifier validation! To fix, there is a need to remove `masterChef != address(0)` from the modifier.

```
    modifier _onlyMasterChef(address _stakingToken) {
        address masterChef = tokenToPoolInfo[_stakingToken].masterChef;
        if (masterChef != msg.sender && masterChef != address(0)) {
            revert onlymasterChef();
        }
        _;
    }
```

Listing 3.4: `ARBRewarder::_onlyMasterChef()`

Recommendation Revise the above-mentioned modifier to ensure the intended pool-specific `MasterChef` is properly validated. Similarly, the contract `MasterPenpie` has a related modifier `_onlyPoolManager()` that can be improved to replace the `msg.sender != address(this)` with `msg.sender != owner()`.

Status The issue has been fixed by the following commit: `a264f63`.

3.5 Potential Sandwich-Based MEV With DexBalancerModuleA

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BuyBackBurnProvider`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

The `Penpie` protocol has a built-in contract named `BuyBackBurnProvider`, which aims to buyback-and-burn a given token to increase the peg. With that, it interacts with the supported `odosRouter`. While examining the token buyback-and-burn logic, we notice it may expose certain `MEV` opportunity.

```
    function _swap(bytes calldata _transactionData, address _tokenIn, uint256 _amountIn,
         address _tokenOut, address _receiver) internal returns (uint256) {
        bytes32 selector;
        assembly {
            selector := calldataload(_transactionData.offset)
        }

        require(bytes4(selector) != IERC20.transferFrom.selector, "Invalid function
            selector");

        IERC20(_tokenIn).safeApprove(address(odosRouter), _amountIn);
        uint256 initialTokenOutBalance = IERC20(_tokenOut).balanceOf(_receiver);

        (bool success,) = odosRouter.call(_transactionData);
        IERC20(_tokenIn).safeApprove(address(odosRouter), 0);

        if (!success) revert SwapFailed();

        uint256 finalTokenOutBalance = IERC20(_tokenOut).balanceOf(_receiver);
        if (finalTokenOutBalance <= initialTokenOutBalance) revert SwapFailed();

        return finalTokenOutBalance - initialTokenOutBalance;
```

```
128     }
```

Listing 3.5: BuyBackBurnProvider::_swap()

To elaborate, we show above the `_swap()` routine. We notice the token swap is routed to `odosRouter`. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of operation.

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.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above sandwich attacks to better protect the interests of protocol users.

Status This issue has been resolved as the `_transactionData` in the `odosRouter.call` (line 119) already has all the slippage control data for enforcement.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

The `Penpie` protocol has a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the protocol-wide operations (e.g., assign roles, configure risk parameters, manage rewards and pools, and upgrade proxies). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function setPoolManagerStatus(
        address _account,
        bool _allowedManager
    ) external onlyOwner {
        PoolManagers[_account] = _allowedManager;

        emit PoolManagerStatus(_account, PoolManagers[_account]);
    }

    function setPenpie(address _penpieOFT) external onlyOwner {
        if (address(penpieOFT) != address(0)) revert PenpieOFTSetAlready();

        if (!Address.isContract(_penpieOFT)) revert MustBeContract();

        penpieOFT = IERC20(_penpieOFT);
        emit PenpieOFTSet(_penpieOFT);
    }

    function updateAllowedPauser(address _pauser, bool _allowed) external onlyOwner {
        allowedPauser[_pauser] = _allowed;

        emit UpdatePauserStatus(_pauser, _allowed);
    }

    function setCompounder(address _compounder)
        external
        onlyOwner
    {
        address oldCompounder = compounder;
        compounder = _compounder;
        emit CompounderUpdated(compounder, oldCompounder);
    }

    function setVlPenpie(address _vlPenpie) external onlyOwner {
        address oldvlPenpie = address(vlPenpie);
        vlPenpie = IVLPenpie(_vlPenpie);
        emit VlPenpieUpdated(address(vlPenpie), oldvlPenpie);
    }

    function setMPendleSV(address _mPendleSV)
        external
        onlyOwner
    {
        address oldMPendleSV = mPendleSV;
        mPendleSV = _mPendleSV;
        emit mPendleSVUpdated(_mPendleSV, oldMPendleSV);
    }

    /**
     * @dev pause pool, restricting certain operations
     */
    function pause() external onlyPauser {
        _pause();
    }
    /**
     * @dev unpause pool, enabling certain operations
     */
    function unpause() external onlyOwner {
        _unpause();
    }
```

Listing 3.6: Example Privileged Operations in `MasterPenpie`

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

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated with the use of a multisig as the admin.

3.7 Improved totalPendleFee Enforcement in PendleStakingBaseUpg

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PendleStakingBaseUpg`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Penpie` protocol is no exception. Specifically, if we examine the `PendleStakingBaseUpg` contract, it has defined a number of protocol-wide risk parameters, such as `totalPendleFee` and `autoBribeFee`. In the following, we show the corresponding routines that allow for their changes.

```
    function setPendleFee(
        uint256 _index,
        uint256 _value,
        address _to,
        bool _isMPENDLE,
        bool _isAddress,
        bool _isActive
    ) external onlyOwner {
        if (_value >= DENOMINATOR) revert InvalidFee();

        Fees storage fee = pendleFeeInfos[_index];
        fee.to = _to;
        fee.isMPENDLE = _isMPENDLE;
        fee.isAddress = _isAddress;
        fee.isActive = _isActive;

        totalPendleFee = totalPendleFee - fee.value + _value;
        fee.value = _value;

        emit SetPendleFee(fee.to, _value);
    }

    function setAutoBribeFee(uint256 _autoBribeFee) external onlyOwner {
        if (_autoBribeFee > DENOMINATOR) revert InvalidFee();
        autoBribeFee = _autoBribeFee;
    }
```

Listing 3.7: PendleStakingBaseUpg::setPendleFee() and PendleStakingBaseUpg::setAutoBribeFee()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the above `setPendleFee()` routine can be improved to ensure the resulting `totalPendleFee` is not larger than `DENOMINATOR`. Note the `addPnedleFee()` routine shares the same issue.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by the following commit: `a5eef3c`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Penpie protocol, which is a multi-chain DeFi product providing yield and veTokenomics boosting services for Pendle Finance. The protocol enables PENDLE holders to earn rewards for their active user engagement, provides Pendle Finance voters with cost-effective voting power, and allows liquidity providers to gain increased APR% without needing to lock PENDLE tokens as vePENDLE. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

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

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

[15] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[16] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.