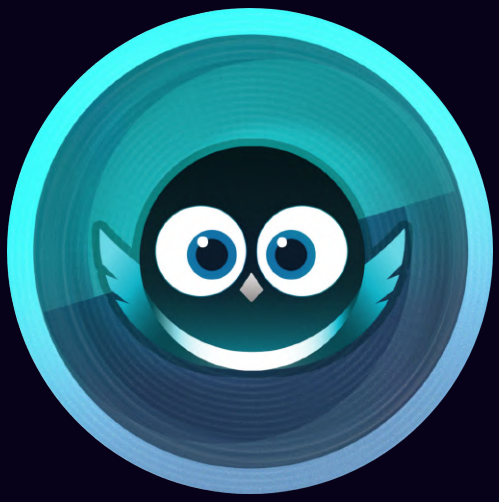


SMART CONTRACT AUDIT

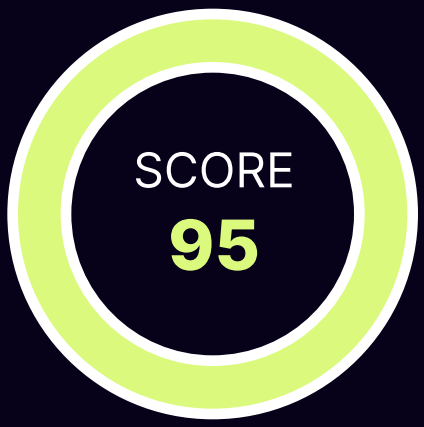
zokyo

July 13th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.



TECHNICAL SUMMARY

This document outlines the overall security of the Penpie smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Penpie smart contracts codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the audit. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Penpie team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Penpie repository:
https://github.com/magpiexyz/pendleMagpie

Last commit - 62c58f07f99900079a1535a4625babfb43976f87

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- vlPenpie.sol
- PenpieOFT.sol
- PendleVoteManagerBaseUpg.sol
- PendleVoteManagerMainChain.sol
- PendleVoteManagerSideChain.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Penpie smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contracts by industry leaders.
03	Thorough manual review of the codebase line by line.

Executive Summary

The auditing team from Zokyo has conducted a thorough examination of Penpies' smart contracts. Throughout the auditing procedure, various issues were identified, ranging from high to low severity, as well as some informational issues. A comprehensive analysis of these findings can be found in the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Penpie team and the Penpie team is aware of it, but they have chosen to not solved it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Possibility of Reentrance via vote()	High	Resolved
2	Incorrect use of modifier	Medium	Resolved
3	Centralization risks in vlPenpie	Medium	Resolved
4	Denial of Service via setCoolDownInSecs()	Medium	Resolved
5	Lock more in order to Unlock	Medium	Acknowledged
6	Centralization risk in PendleVoteManager	Medium	Resolved
7	Unsafe Uint64 Casting in PendleVoteManager	Medium	Resolved
8	Renounce Ownership	Low	Resolved
9	Transfer Ownership	Low	Acknowledged
10	Unprotected initializer could be called by anyone	Low	Resolved
11	Missing zero address checks	Low	Resolved
12	Missing Reentrance init call	Low	Resolved
13	For loop over dynamic array	Low	Acknowledged
14	Missing forwarding of rewards	Low	Resolved
15	Returned boolean value ignored	Low	Resolved
16	Possibility of out of gas in castVotes	Low	Acknowledged
17	Discarded Votes due to unsuitable datatype	Low	Acknowledged
18	Returned UserPoolData from external call is not validated	Low	Acknowledged
19	Init function call missing	Low	Resolved
20	Incorrect usage of sign	Informational	Resolved

#	Title	Risk	Status
21	Incorrect Natspec comments	Informational	Resolved
22	Unused modifier	Informational	Resolved
23	Missing events for critical functions	Informational	Acknowledged
24	Unnecessary redundant initialization of variables	Informational	Resolved
25	Lock Solidity	Informational	Resolved
26	Follow checks effects interactions pattern	Informational	Acknowledged
27	Declaration of gap variable	Informational	Resolved

HIGH | RESOLVED

Possibility of Reentrance via vote()

The **vote()** function of the **PendleVoteManagerMainChain** contract is vulnerable to a reentrance attack. This is because the **_updateVoteAndCheck()** function makes an external call via **stakeFor()** and **withdrawFor()** functions. This attack can be carried out if the bribe contract is a malicious contract.

Recommendation:
It is advised to follow the checks effects interactions pattern by making all the state variable changes after the **stakeFor()** and **withdrawFor()** function calls. Alternatively, a reentrance guard can be added for the same.

Comment: The client added a nonReentrant modifier for vote() and castVotes(). The bribePool was removed and not used anymor

MEDIUM | RESOLVED

Incorrect use of modifier

The **_claimRewardFor()** function in **PendleVoteManagerBaseUpg** contract, calls the **getReward()** function of the BribeRewardPool. But the **getReward()** is an **onlyMasterPenPie** function, which means that only **masterPenPie** contract can call it. This will result in all the **_claimRewardFor()** calls to fail in production.

Recommendation:
The current code does not allow two different contracts(masterPenPie and **PendleVoteManager)** to call the **getReward()** function of the **BribeRewardPool** contract. Consider either adding two operators in the modifier of the **getReward()** function, or reviewing the business and operational logic.

Comment: The claim functions were moved to PenpieBribeRewardDistributor with different codes and logic.

Centralization risks in vlPenpie

The **vlPenpie** can be paused anytime by an admin using **pause()**. This can be exploited by a malicious admin to deny service to users.
Also, the token can be upgraded anytime to change its behavior. This can be exploited by a malicious admin to introduce a malicious piece of code that can result in loss of funds.
The **setPendleVoteManager()** can be used by a malicious admin to change the **pendleVoteManager**. This can result in a Denial of Service, as the **startUnlock** will fail to work due to the if statement on line: 274.
Similarly, **setMasterChief()** can be used by a malicious admin to change the **masterPenPie**. This can result in Denial of Service as the users will not be able to call **unlock()** or **forceUnlock()** leading to stuck funds.

Recommendation:

It is advised to use a multisig wallet in order to decentralize the use of this function between trusted participants.

Comment: The client said that they would be utilising a multisig in order to mitigate this issue.

MEDIUM | RESOLVED

Denial of Service via setCoolDownInSecs()

The **setCoolDownInSecs()** function can be used to change the **coolDownInSecs** variable to a very large value. This can be unfair, especially for the participants who had already locked their tokens and were expecting a different cooldown value. For example, let's a user A locks his tokens at a time when the cooldown was T1. Assume that the owner changes the coolDownInSecs to T2. Now, when the user A tries to unlock his tokens, cooldown T1 should be applied for his token's unlock and not T2

Recommendation:

It is advised to do the following to mitigate this issue:
A) Add an upper limit to the coolDownInSecs to be set via **setCoolDownInSecs()**.
B) Add require checks to ensure that the above scenario is mitigated as described in the above example.
C) Use a multisig wallet to decentralize the usage of **setCoolDownInSecs()** between trusted participants.

Comment: The client said that they would be utilising a multisig in order to mitigate this issue.

MEDIUM | ACKNOWLEDGED

Lock more in order to Unloc

If the users locked their tokens at say time T1. And the **pendleVoteManager** was set after T1. Then users will need to lock more **Penpie** in the contract if they want to call **startUnlock()** on their previous tokens. Thus, it can be unfair for the users who had locked their tokens before the setting of **pendleVoteManager**.

Recommendation:

It is advised to let only users who have locked their tokens in the **vlPenpie** contract after time T1 to follow the condition on **line: 274** which leads to revert. This could require rewriting of the if statement accordingly to make the fix.

Comment: The client said that voted vlPNP can not be startUnlocked, so if users did not vote with their vlPNP after PendleVoterManager is set, they should still be able to startUnlock.

Centralization risks in PendleVoteManager

The excessive authority granted to the admin poses a vulnerability in the contracts. The central issue is that the admin has unrestricted access to critical functions such as **setRemoteDelegate()** and general setters/mutators.

Also, the function **removePool()** can be used to disable the pool at anytime. A malicious admin can exploit this to manipulate the outcome of voting results, such as by disabling the pool before the expected time when the pool was supposed to be disabled.

This concentration of power raises concerns, especially when certain functions carry higher risks than others. Depending on the project's objectives, it is advisable to implement governance mechanisms or utilize multisig wallets to distribute control and mitigate potential risks associated with a single wallet's authority.

Recommendation:

It is advised to use a multisig wallet in order to decentralize the use of this function between trusted participants. Additionally, a secure governance mechanism can be implemented for the same.

Comment: The client said that they would be utilising a multisig in order to mitigate this issue.

Unsafe Uint64 Casting in PendleVoteManager

In _getVoteInPercentage() function:

```
169  else {
170    votePerc = exactVoteCount * 100 / totalVotes();
171    pendleVote = uint64(_getExactPercentage(votePerc));
172  }
```

And

```
161     exactVoteCount = uint256(_vote) + _currentVote;
162     votePerc = exactVoteCount * 100 / totalVotes();
163     pendleVote = uint64(_getExactPercentage(votePerc));
```

_getExactPercentage() function returns **uint256** and it is being casted to **uint64** without validating that the input number is less than or equal to **type(uint64).max**. The input number needs to be as low as **1845** in order to provoke this casting.

Recommendation:

Implement a safe casting function that validates input is less than **type(uint64).max**

Fix: In commit 60e393d, the function is no longer implemented, making the issue irrelevant.

LOW | RESOLVED

Renounce Ownership

The **renounceOwnership()** function of the **Ownable** contract, which is inherited by **VLPenPie**, can be called accidentally by the admin leading to the immediate renouncement of ownership to zero address after which any **onlyOwner** functions will not be callable which can be risky.

Recommendation:
It is advised that the Owner cannot call **renounceOwnership()** without first transferring ownership to a different address. Additionally, if a multi-signature wallet is utilized, executing the **renounceOwnership()** method for two or more users should be confirmed. Alternatively, the Renounce Ownership functionality can be disabled by overriding it.

Comment: The client said that they would be utilizing a multisig in order to mitigate this issue.

LOW | ACKNOWLEDGED

Transfer Ownership

The **transferOwnership()** function of the Ownable contract, which is inherited by **VLPenPie**, allows the current admin to transfer his privileges to another address. However, inside **transferOwnership()**, the newOwner is directly stored
into the storage owner, after validating the newOwner is a non-zero address, and immediately overwrites the current owner. This can lead to cases where the admin has transferred ownership to an incorrect address and wants to revoke the transfer of ownership, or in the cases where the current admin comes to know that the new admin has lost access to his account.

Recommendation:
It is advised to make ownership transfer a two-step process.
Refer- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol
and https://github.com/razzorsec/RazzorSec-Contracts/blob/main/AccessControl/SafeOwn.sol

LOW | RESOLVED

Unprotected initializer could be called by anyone

The **Initializable** library is used by **vlPenPie**. To prevent leaving an implementation contract uninitialized, it is recommended to add the **_disableInitializers()** function in the constructor to lock the contracts automatically during the deployment.

Recommendation:
Consider locking the implementation contract by invoking **_disableInitializers()** function.

Refer- https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#initializing_the_implementation_contract

LOW | RESOLVED

Missing zero address check

The **_penPie** parameter in **__vlPenPie_init_()** of the vlPenpie contract, has a missing zero address check. Due to this a zero address can be passed accidentally to the parameter to set **_penpie** as a zero address.
Also the **lockFor()** function has a missing zero address check.This can lead to funds being locked for the "For" address which is zero address, resulting in lock of funds forever.

Recommendation:
It is advised to add a zero address check because **PenPie** address can not be changed to another address once it is set, although upgrading the contract with new parameters is entirely possible.

LOW | RESOLVED

Missing Reentrance init call

Missing call to **__ReentrancyGuard_init()** in the **__vlPenPie_init_()** function. Although it is expected that the nonReentrant modifier will work in either case, it would be a good security practice that the Reentrance Guard variables are initialized to avoid confusion and unintended issues.

Recommendation:
It is advised to add a call to **__ReentrancyGuard_init()** in the **__vlPenPie_init_()** function.

LOW | ACKNOWLEDGED

For loop over dynamic array

If the size of **poolInfos** is too large, then **getClaimReward()** might run out of gas resulting in a Denial of Service.

Recommendation:
It is advised to restrict the m**aximum** size of the pool.

LOW | RESOLVED

Missing forwarding of rewards

In the **harvestSinglePool()** function, no rewards are forwarded using **_forwardRewards()**. This is inconsistent with the **getClaimReward()** function, which is similar but forwards the rewards after harvest using **_forwardRewards()**. This could result in the rewards only being approved and not being transferred by the contract. This may result in stuck funds if the msg.sender for the **harvestSinglePool()** is a smart contract because it might be unable to call transferFrom due to its lack of usage of transferFrom.

Recommendation:
It is advised to add **_forwardRewards()** at the end of **harvestSinglePool()** function and check business and operational logic.

Comment: The function was **harvestSinglePool()** was removed in the fixed version of the contract by the client.

LOW | RESOLVED

Returned boolean value ignored

The **_claimRewardFor()** ignores the return value by **getReward()** function, which is a boolean value.

Recommendation:
It is advised to also return the boolean value or add a requirement check on the boolean value returned.

Comment: The **_claimRewardFor()** was removed in the latest fixes and code logic by the client

LOW | ACKNOWLEDGED

Possibility of out of gas in castVotes

The c**astVotes()** function has external calls inside a for loop in the form of **voter.getUserPoolVote()** and **vePendle.balanceOf()**. This can result in a Denial of Service if there are a lot of pools to iterate over.

Recommendation:
It is advised to add a limit on the maximum number of pools in order to avoid this issue.

LOW | ACKNOWLEDGED

Discarded Votes due to unsuitable datatype

In **_updateVoteAndCheck()** function of the **PendleVoteManagerBaseUpg** contract, the weights of votes add up as follows:

```
263  uint256 absVal = uint256(-weight);
264  pool.totalVoteInVlPenPie -= absVal;
265  userVotedForPoolInVlPenPie[_user][pool.market] -= absVal;
```

This is given **weight < 0** , while we have **pool.totalVoteInVlPenPie** of type uint256 and **userVotedForPoolInVlPenPie** maps to a **uint256** also. These arithmetic operations revert when **userVotedForPoolInVlPenPie** and pool.totalVoteInVlPenPie are smaller than absVal. Suppose the following scenario:

```
User A sends vote with weight 20
User B sends vote with weight -10
```

At the beginning at zero total votes, we have **User A** sending a positive vote and **User B** sends a negative vote then we end up with total **votes = 20 - 10 = 10**. In another alternative scenario **User A** notices the vote of **User B** in the transaction pool, then he tunes the gas fee so that **User B** has his call mined first. In that case, total votes are **20** because User B fails to get his vote counted.
The issue arises because of the **uint256** type of **pool.totalVoteInVlPenPie** and **userVotedForPoolInVlPenPie** which does not suit the way vote is counted.

Recommendation:
It is advised to add a limit on the maximum number of pools in order to avoid this issue.

LOW | ACKNOWLEDGED

Returned UserPoolData from external call is not validated

PendleVoteManagerMainChain.sol - In **getVoteForMarket() function**, the **userPoolData** return is not validated to be actually existent. An invalid address of market might be entered which returns a zero value by default. The result of the calculation is zero in this case, and it is not clear if the zero is a result of a valid calculation, or it is just due to an invalid market input.

```
75  function getVoteForMarket(address market) public view returns
(uint256) {
76      IPVoteController.UserPoolData memory userPoolData = voter
77          .getUserPoolVote(address(pendleStaking), market);
78      uint256 poolVote = userPoolData.weight * totalVotes() / 1e18 ;
79      return poolVote;
80  }
```

Recommendation:
Short term, add a require statement to validate the attributes of userPoolData to be non-zero value.

LOW | RESOLVED

Init function call missing

__Pausable_init() is not called in the new version of PendleVoteManagerBaseUpg contract.

Recommendation:
It is advised to call the appropriate init function in the contract.

INFORMATIONAL | RESOLVED

Incorrect usage of sign

The **_maxSlots** in **__vlPenPie_init_()** of the vlPenPie contract can never be negative as it is an unsigned integer.
Similarly, the **_coolDownSecs** can never be negative as it is also an unsigned integer..

Recommendation:
It is advised to remove the < sign as it is not required.

INFORMATIONAL | RESOLVED

Incorrect Natspec comments

The Natspec comments above the **startUnlock()** function describe **3 parameters** that do not exist in this function.

Recommendation:
It is advised to write Natspec correct comments with describing the correct parameters existing in the function.

INFORMATIONAL | RESOLVED

Unused modifier

The **PendleVoteManagerBaseUpg** contract has modifier **refundUnusedEth()** which is not being used anywhere.

Recommendation:
It is advised to remove the unused modifier.

Comment: The client has said that they would be removing it in the fixes.

INFORMATIONAL | ACKNOWLEDGED

Missing events for critical functions

Critical functions like **setMainChainId()**, **setRemoteDelegate()**, and **setMinRemoteCastGas()** in PendleVoteManager contracts do not emit an event. The same goes for **setPendleVoteManager()** in **VLPenpie** contract.
Emitting events is a best practice for offchain monitoring.

Recommendation:
It is advised to emit appropriate events for the above mentioned events.

Fix: In commit `60e393d`, the issue is partially resolved, as of this last commit, setPendleVoteManager() does not emit the relevant events.

INFORMATIONAL | RESOLVED

Unnecessary redundant initialization of variables

PendleVoteManagerMainChain.sol/PendleVoteManagerSideChain.sol - In **__PendleVoteManagerMainChain_init()/__PendleVoteManagerSideChain_init()** functions which are invoked in order to initialize the state of the contracts. There exist a repeated initialization of two state variables:
pendleStaking = _pendleStaking;
vlPenPie = _vlPenPie;

These variables are initialized already by the base contract on calling **__PendleVoteManagerBaseUpg_init()** function, which is the base of each of the two contracts.

Recommendation:
No need for **pendleStaking** and **vlPenPie** value assignment, therefore better to be removed from the body of the initializer functions.

Fix: Issue addressed at commit `60e393d`

Lock Solidity

Lock the pragma to a specific version, since not all the EVM compiler versions support all the features, especially the latest one's which are kind of beta versions, So the intended behavior written in code might not be executed as expected. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs.

Recommendation:
Fix the pragma version of each contract.
Fix: Fixed at version 0.8.19

Follow checks-effects-interactions pattern

As a best practice it is recommended to follow checks-effects-interactions pattern whenever it is possible. Worth noting to point out snippets of **vlPenpie** contract in which this pattern is broken.
In **_unlock()** function we have interaction with **masterPenPie** preceding the effects of updating the values of **totalAmountInCoolDown** and **totalAmount.**

```
IMasterPenpie(masterPenPie).withdrawVlPenPieFor(_unlockedAmount,
msg.sender);
totalAmountInCoolDown -= _unlockedAmount;
totalAmount -= _unlockedAmount;
```

In **_lock()** function we have interaction with **penPie** preceding the effects of updating the value of **totalAmount**.

```
penPie.safeTransferFrom(spender, address(this), _amount);
IMasterPenpie(masterPenPie).depositVlPenPieFor(_amount, _for);
totalAmount += _amount;
```

In **transferPenalty()** function, interaction with **penPie** precedes the update of the value of **totalPenalty**.

```
IERC20(penPie).safeTransfer(penaltyDestination, totalPenalty);
emit PenaltySentTo(penaltyDestination, totalPenalty);
totalPenalty = 0;
```

Recommendation:
Move the interaction line of codes after the effects.

Fix: Not possible to change as it will affect function's intended behavior demanded by the client.

Declaration of gap variable

In **PendleVoteManagerBaseUpg**, the gap variable is declared before the state variables which is contrary to the Openzeppelin convention of declaring gap variables after the state variables.

Recommendation:
It is advised to follow Openzeppelin standard of declaring gap variables at the end of all the state variables as a good security practice.
Refer- https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#storage-gaps

	vlPenpie.sol PenpieOFT.sol PendleVoteManagerBaseUpg.sol PendleVoteManagerMainChain.sol PendleVoteManagerSideChain.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Penpie team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Penpie team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo