Penpie Lucky Spin Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

Penpie is a next-generation DeFi platform designed to provide Pendle Finance users with yield and veTokenomics boosting services. Integrated with Pendle Finance, Penpie focuses on locking PENDLE tokens to obtain governance rights and enhanced yield benefits within Pendle Finance. Penpie revolutionizes the way users can maximize returns on their investments and monetize their governance power.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	Penpie Lucky Spin
Repository	penpie-contracts
Commit	353307553c77...
Date	October 2023

Issues Found

Severity	**Count**
Critical Risk	2
High Risk	0
Medium Risk	1
Low Risk	4
Informational	3
Gas Optimizations	0
Total Issues	**10**

Summary of Findings

Title	**Status**
[C-1] Reward expiration prevents future interactions	Resolved
[C-2] Pausing the contract can lead to loss of rewards	Resolved
[M-1] Removing a "spinReward" doesnt account for unclaimed user rewards	Resolved
[L-1] Unsafe use of transfer()/transferFrom() with IERC20	Resolved
[L-2] safeApprove() is deprecated	Resolved
[L-3] safeApprove() may revert if the current approval is not zero	Resolved
[L-4] Set a lower/upper bound on reward rate/weights	Resolved
[I-1] Consider using uint48 for time-related variables	Resolved
[I-2] Missing Event for critical parameters change	Resolved
[I-3] Remove unused code	Resolved

5 Findings

5.1 Critical Risk

[C-1] Reward expiration prevents future interactions

Context: PenpieLuckySpin

Impact: Users who let their rewards expire can no longer use the contract, as they will be permanently prevented from interacting further due to the state of the waitingClaim flag.

Description: The lottary uses the waitingClaim flag to manage user interactions. When users spin the wheel, their waitingClaim is set to true:

```
userInfo.waitingClaim = true;
```

This prevents users from spinning the wheel while they have a reward pending:

```
function _spin(uint256 _inputAmount) internal returns (uint256 requestID) {
    UserSpinInfo storage userInfo = userSpinInfos[msg.sender];
    if (userInfo.waitingClaim) revert WaitToClaim();
}
```

But also ensures they have pending rewards:

```
function getRewards() external whenNotPaused nonReentrant {
    UserSpinInfo storage userInfo = userSpinInfos[msg.sender];
    if(!userInfo.waitingClaim) revert NothingToClaim();
}
```

However, there's a design flaw. Rewards are expirable:

```
if(
    expiredSecond > 0 &&
    userInfo.spinTime + expiredSecond < block.timestamp
) revert RewardExpired();
userInfo.waitingClaim = false;
```

If a reward expires, the waitingClaim flag **cannot be** reset, preventing the user from further using the contract.

Recommendation: Instead of reverting, just reset the user spin info of the user.

[C-2] Pausing the contract can lead to loss of rewards

Context: PenpieLuckySpin

Impact: Users of the PenpieLuckySpin contract may lose their rewards due to an inability to claim them during contract pause periods. This can result in a loss of trust in the system and potential monetary loss for the users.

Description: The getRewards function is designed to allow users to claim their rewards. However, there's a conditional check that ensures rewards are not claimed after their expiration:

```
function getRewards() external whenNotPaused nonReentrant {
        UserSpinInfo storage userInfo = userSpinInfos[msg.sender];
        if(!userInfo.waitingClaim) revert NothingToClaim();
        if(userInfo.waitingVRF) revert UnderSpinning();

        if(
            expiredSecond > 0 &&
            userInfo.spinTime + expiredSecond < block.timestamp
        ) revert RewardExpired();
        userInfo.waitingClaim = false;
```

A potential vulnerability arises due to the contract's ability to be paused:

```
    function pause() external nonReentrant onlyOwner {
        _pause();
    }

    function unpause() external nonReentrant onlyOwner {
        _unpause();
    }
```

When the contract is paused, functionalities, including the getRewards function (due to the whenNotPaused modifier), are halted. The issue is that while the contract is paused, the expiration time for the rewards continues to progress, thereby preventing users from claiming their rewards.

If the contract remains paused beyond a reward's expiration time, users will be left unable to claim their rewards.

Recommendation: To ensure users don't lose out on their rewards during contract pauses, consider the following solutions:

1. Remove the whenNotPaused modifier from the getRewards function.
2. Alternatively, implement a mechanism to pause the progression of reward expiration during contract pauses. This can be achieved by adjusting the spinTime or introducing a variable that keeps track of total paused time, offsetting the expiration check accordingly.

5.2 Medium Risk

[M-1] Removing a "spinReward" doesnt account for unclaimed user rewards

Context: PenpieLuckySpin

Impact: Removing a reward from the PenpieLuckySpin contract can inadvertently lock user's rewards and prevent subsequent spins. This can result in potential losses for users and cause service disruption.

Description: The contract allows the owner to add rewards using the addReward function:

```
function addReward(address _token, uint256 _baseRate, uint256 _probWeight) public onlyOwner whenPaused {
    spinRewards[rewardsCount] = SpinReward(_token, _baseRate, _probWeight);
    rewardsCount++;
}
```

Rewards are then calculated and if a user wins, his reward is associated with a specific number through the fulfillRandomWords function:

```
function fulfillRandomWords(
    uint256 _requestId ,
    uint256[] memory randomWords
) internal override _onlyCoordinator {
    address userAddress = userRequests[_requestId];
    UserSpinInfo storage userInfo = userSpinInfos[userAddress];
    userInfo.waitingVRF = false;
>>      userInfo.requestResult = _pickRandomReward(randomWords[0]);
```

However, there's a critical flaw in the removal process. The removeReward function does not take into account any pending-to-be-claimed rewards:

```
function removeReward(uint256 _index) external onlyOwner whenPaused {
    if(_index >= rewardsCount) revert OutOfRange();

    if (_index < rewardsCount - 1) {
        spinRewards[_index] = spinRewards[rewardsCount - 1];
    }
    delete spinRewards[rewardsCount - 1];
    rewardsCount--;
}
```

This creates an issue because once the reward is removed, users with pending rewards can neither claim them nor request another spin as the waitingClaim variable would remain "true".

This design oversight effectively renders the removal of rewards problematic for the protocol. If executed, it guarantees disruption for users by either causing losses or making their rewards inaccessible.

Recommendation: Implement a mechanism that checks for any pending-to-be-claimed rewards tied to the specific index before removal.

Additionally, you could reorganize the data structure to accommodate changes without affecting existing rewards or implement versioning to transition users smoothly.

5.3 Low Risk

[L-1] Unsafe use of transfer()/transferFrom() with IERC20

Context: PenpieLuckySpin.sol

Description: Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. When these sorts of tokens are cast to `IERC20`, their function signatures do not match and therefore the calls made, revert (see this link for a test case).

Use OpenZeppelin's `SafeERC20`'s `safeTransfer()/safeTransferFrom()` instead

```
        IERC20(inputToken).transferFrom(msg.sender, address(this), _inputAmount);
```

[L-2] safeApprove() is deprecated

Context: PenpieLuckySpin.sol

Description: Deprecated in favor of safeIncreaseAllowance() and safeDecreaseAllowance(). If only setting the initial allowance to the value that means infinite, safeIncreaseAllowance() can be used instead. The function may currently work, but if a bug is found in this version of OpenZeppelin, and the version that you're forced to upgrade to no longer has this function, you'll encounter unnecessary delays in porting and testing replacement contracts.

```
IERC20(inputToken).safeApprove(address(converter), _inputAmount);
```

[L-3] safeApprove() may revert if the current approval is not zero

Context: PenpieLuckySpin.sol

Description: While Tether is known to do this, it applies to other tokens as well, which are trying to protect against this attack vector. safeApprove() itself also implements this protection. Always reset the approval to zero before changing it to a new value (forceApprove() does this for you), or use safeIncreaseAllowance()/safeDecreaseAllowance()

```
      IERC20(inputToken).safeApprove(address(converter), _inputAmount);
```

[L-4] Set a lower/upper bound on reward rate/weights

Context: PenpieLuckySpin.sol

Description: Consider adding minimum/maximum value checks to ensure that the reward base rate/weight below can never be used to excessively harm users

```
    function updateReward(uint256 _index, uint256 newBaseRate, uint256 newProbWeight) external onlyOwner
    ↪   whenPaused {
        if(_index >= rewardsCount) revert OutOfRange();

        SpinReward storage reward = spinRewards[_index];
        reward.baseRate = newBaseRate;
        reward.probWeight = newProbWeight;

        // emit RewardUpdated(_index, reward.token, newBaseRate, newProbWeight);
    }
```

5.4 Informational

[I-1] Consider using uint48 for time-related variables

Context: PenpieLuckySpin.sol

Description

While `uint32` ends in 2106 and could cause some issues in this distant future, higher types than `uint48` (like `uint256`) aren't necessary for time-related variables.

```
    uint256 spinTime;
```

[I-2] Missing Event for critical parameters change

Context: PenpieLuckySpin.sol

Description: Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

```
function addReward(address _token, uint256 _baseRate, uint256 _probWeight) public onlyOwner whenPaused {
  ...
  }

  function updateReward(uint256 _index, uint256 newBaseRate, uint256 newProbWeight) external onlyOwner
  ↪   whenPaused {
  }

  function removeReward(uint256 _index) external onlyOwner whenPaused {
  ...
  }

  function setThresholds(Threshold[] memory _thresholds) public onlyOwner whenPaused {
  ...
  }
```

[I-3] Remove unused code

Context: PenpieLuckySpin.sol

Description

```
  uint256 public constant WAD = 1e18;
```