# Coinflip Security Review

## Pashov Audit Group

Conducted by: 0xunforgiven, Hals, merlinboii

February 19th 2025 - February 22nd 2025

# Contents

# 1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

# 2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

# 3. Introduction

A time-boxed security review of the **arcade-contracts/coinflip-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

# 4. About Coinflip

Coinflip is a decentralized coin flipping game combined with a staking pool, allowing users to participate in the game and earn rewards through staking. The project uses VRF (Verifiable Random Function) for randomness and plans to deploy on multiple chains.

# 5. Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

## 5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

## 5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

## 5.3. 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

# 6. Security Assessment Summary

*review commit hash* - 9849fa45d21a9b331164051805fa0d4e93dd85c6

*fixes review commit hash* - 6e7bb0621911f07939ec0aeacacd6ac9b76190b4

## Scope

The following smart contracts were in scope of the audit:

- `Flip`
- `Staking`

# 7. Executive Summary

Over the course of the security review, 0xunforgiven, Hals, merlinboii engaged with Coinflip to review Coinflip. In this period of time a total of **18** issues were uncovered.

## Protocol Summary

| | |
|---|---|
| **Protocol Name** | Coinflip |
| **Repository** | https://github.com/arcade-contracts/coinflip-contracts |
| **Date** | February 19th 2025 - February 22nd 2025 |
| **Protocol Type** | Game |

## Findings Count

| **Severity** | **Amount** |
|---|---|
| Critical | 2 |
| Medium | 6 |
| Low | 10 |
| **Total Findings** | **18** |

# Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [C-01] | Pending payouts excluded from total balance cause incorrect share calculations | Critical | Resolved |
| [C-02] | Pending stake not accounted for in liquidity calculations | Critical | Resolved |
| [M-01] | Issues with game refunds and completion when Staking contract is changed | Medium | Resolved |
| [M-02] | Unstakers receive unfair amounts due to dynamic balance changes | Medium | Resolved |
| [M-03] | Locked tokens when token is removed during depositDelay period | Medium | Resolved |
| [M-04] | Locked liquidity when a game contract is removed from Staking.authorizedGames | Medium | Resolved |
| [M-05] | User can frontrun completeGame to finalize stake or unstake request | Medium | Resolved |
| [M-06] | Zero share minting via token balance inflation | Medium | Resolved |
| [L-01] | Staking contract does not consider token decimals | Low | Resolved |
| [L-02] | Missing reqId Deletion in cancelGame() | Low | Resolved |
| [L-03] | Incorrect owed amounts caclulation in Staking.totalOwed() function | Low | Resolved |
| [L-04] | Unstakers unable to receive staked amount if blacklisted by staking token | Low | Acknowledged |

| | | | |
|---|---|---|---|
| [L-05] | Claim failure for blacklisted addresses in claimPendingPayout and claimPendingWithdrawal | Low | Resolved |
| [L-06] | Strict equality to zero share validation blocks token removal | Low | Resolved |
| [L-07] | Games can be completed even after gameTimeout | Low | Resolved |
| [L-08] | Locked tokens and payout failure when removing token without checking lockedLiquidity | Low | Acknowledged |
| [L-09] | Inconsistent payout handling due to mixing staker rewards with lockedLiquidity | Low | Acknowledged |
| [L-10] | Player loss due to netPayout less than betAmount for some combinations | Low | Acknowledged |

# 8. Findings

## 8.1. Critical Findings

## [C-01] Pending payouts excluded from total balance cause incorrect share calculations

### Severity

**Impact:** High

**Likelihood:** High

### Description

After each game completes, the protocol attempts to transfer the winner's payout through `Staking.transferPayout()`. If this transfer fails, the amount is stored in `pendingPayouts` to be claimed later. However, the contract continues to include these pending payouts in its balance calculations, which causes overestimation of the staking balance using in the calculation.

```
function transferPayout
  (address token, address recipient, uint256 amount) external {
    --- SNIPPED ---
    if (!callSucceeded) {
@<>     pendingPayouts[token][recipient] += amount;
        emit TransferFailed(token, recipient, amount);
        return false;
    }
    --- SNIPPED ---
}
```

The problem is from how the contract calculates total balance. It simply uses `IERC20(token).balanceOf(address(this))` without subtracting `pendingPayouts`. This inflated balance affects several critical functions: `totalOwed()`, `lockLiquidity()`, `finalizeStake()`, and `finalizeUnstake()`.

Consider the following scenario:

1. Initial setup:

    - Alice has staked 1000 tokens (1000 shares)
    - Someone starts a game:
        - `betAmount`: 100 tokens
        - `netPayout`: 200 tokens (2:1 odds)
        - Protocol locks 200 tokens

2. The game ends, but winner's payout of 200 tokens fails to transfer:

    - Staking contract adds 200 tokens to `pendingPayouts`
    - Pool now holds 1100 tokens (original 1000 + 100 from `betAmount`)
    - 200 tokens remain locked

Now, let's analyze the apparent token owned to Alice:

- Alice's shares represent 1100 tokens (1000 shares × 1100 / 1000), but this does not account for the loss that has occurred (Alice's shares still appear to be worth 1:1)
- Winner is owed 200 tokens
- Total owed tokens: 1300 tokens, but the protocol only has 1100 tokens

For comparison, if the payout had gone through normally:

- Pool would have 900 tokens (1000 - 200(`netPayout`)+ 100(`betAmount`))
- Alice's shares would correctly represent 900 tokens for maximum owed (1000 shares × 900 / 1000)

3. This has a cascading impact on the pool:
    - Alice withdraws 900 tokens, due to liquidity locks, but the incorrect calculation still propagates the effect.
    - Alice burns 900 shares, leaving 100 shares, which resulting in token balance of 200 (1100 - 900)
    - The winner claims their 200 tokens, depleting the token balance
    - As a result, the remaining 100 shares are no longer backed by any token balance

This leaves the pool in an loss state, as shares become worthless, and any new deposits would be immediately devalued. Furthermore, Alice could request to unstake her remaining shares and wait for new deposits, essentially stealing some of their stake.

## Recommendation

Introduce a variable to track the total pending payout for each user and properly account for pending payouts in all balance calculations and validations.

Moreover, the `lockedLiquidity` is not properly used in calculations as it also represents the on-going game reserve, which has not been finalized as a loss or win. This can create an underestimate for the shares, even though the game is expected to be finalized quickly, so the `lockedLiquidity` might be treated as being equal to the sum of the pending payouts. There is also a case where the unstake is calculated during on-going games.

# [C-02] Pending stake not accounted for in liquidity calculations

## Severity

**Impact:** High

**Likelihood:** High

## Description

The `Staking` contract uses `IERC20(token).balanceOf(address(this))` to determine total balance for share calculations and liquidity validation. However, in `Staking.requestStake()`, tokens are transferred to the contract immediately when requesting a stake without tracking the pending stake. This is included in the liquidity in many crucial processes, such as share calculation, liquidity locked validation, and amount owed calculation.

```
function requestStake(address token, uint256 amount) external nonReentrant {
    --- SNIPPED ---

    // Transfer the tokens from the user to this contract
@>  IERC20(token).safeTransferFrom(msg.sender, address(this), amount);

    --- SNIPPED ---
}
```

This is prone to incorrect token balance usage for calculation and validation, leading to pending stakers' loss of funds.

Consider the example following possible scenario below:

0. Assume there is one existing staker in the pool with 100 shares and pool balance is 100 tokens (this user owns 100% of the liquidity).

1. The previous staker decided to request unstake and they already pass the cooldown periods.

2. Alice requests to stake 100 tokens, the token transfer to the staking contract and also the share calculation uses the token contract balance in calculation without considering the requested amount that should allocate for Alice when finalizing the stake is available for her.

3. The previous staker finalize the unstake request and takes 200 tokens back (100 + 100(Alice)).

```
function finalizeUnstake(address token) external nonReentrant {
    --- SNIPEPD ---

@>  uint256 totalBalance = IERC20(token).balanceOf(address(this));
    // The user's pro-rata portion of the underlying tokens
@>  uint256 amountOwed = (sharesToRedeem * totalBalance) / totalShares;

    --- SNIPEPD ---
@>  IERC20(token).safeTransfer(msg.sender, amountOwed);

    emit Unstaked(msg.sender, token, amountOwed);
}
```

4. Alice will finalize with 100 shares but 0 tokens back.

There are also further cases from the functions that use the overestimation of the token balance in the staking pool: `finalizeStake()`, `finalizeUnstake()`, `totalOwed()`, and `lockLiquidity()`

# Recommendation

Track pending stakes and exclude them from being treated as available liquidity. Furthermore, the pending stakes should be updated whenever the stake is finalized, to include them in the liquidity pool and remove them from being pending.

# 8.2. Medium Findings

## [M-01] Issues with game refunds and completion when `Staking` contract is changed

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

In the `Flip` contract, the `setStakingContract()` function allows the contract owner to change the staking contract address. However, changing the staking contract while there are pending games will result in those pending games being left incomplete. Furthermore, the owner may not be able to refund the player by calling `cancelGame()`, as there may not be enough liquidity in the new staking contract to unlock the incomplete `game.netPayout`.

### Recommendations

Consider recording the staking contract address at the time of game creation in the `PendingGame` struct for each player, where this recorded address is used when completing or canceling the game.

## [M-02] Unstakers receive unfair amounts due to dynamic balance changes

### Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

In the `Staking` contract, users can finalize their unstake via the `finalizeUnstake()` function, where they will receive their `amountOwed` based on the available balance of the contract. However, the issue arises because the available balance dynamically changes when the game contracts interact with the contract to pay winners or transfer game losers' bits. This results in an unexpected amount of assets being transferred to the user. Additionally, this could be exploited by users who have passed their `unstakeDelay` and can frontrun the the `finalizeUnstake()` transaction to finalize their unstakes first, reducing the `amountOwed` for other unstakers.

```
function finalizeUnstake(address token) external nonReentrant {
    //...
}
```

Similar issue with the `finalizeStake()` function:

```
function finalizeStake(address token) external nonReentrant {
    //...
}
```

## Recommendation:

Add a `minAmountOut` parameter to the `finalizeUnstake()` function ( `minShare` for the `finalizeStake()` function) to ensure that the unstaker receives an acceptable amount of assets and avoids unexpected reductions in the amount they are owed due to the dynamic changes in the available contract's balance.

# [M-03] Locked tokens when token is removed during `depositDelay` period

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description:

The `Staking.finalizeStake()` function allows staking to be completed even if the intended token is removed from the `acceptedTokens` list **during the `depositDelay`** (if between a `requestStake()` and `finalizeStake()` calls, the intended token is removed). This results in locking the user's tokens for at least the duration of `depositDelay` + `unstakeDelay` (currently locking for at least a total of **49 hours**) until the user can withdraw their non-accepted tokens. Furthermore, if the contract's liquidity is insufficient to complete `finalizeUnstake()`, the user's stake could remain locked for an even longer period. This scenario is a possible due to the lack of a check on whether the token has been removed from the `acceptedTokens` when the user calls `Staking.finalizeStake()` function.

## Recommendation:

Update the `finalizeStake()` function to handle the case where the token has been removed from the `acceptedTokens` list, where the non-accepted tokens is refunded to the user , ensuring that users are not locked out of their tokens for an extended period if the token is no longer accepted.

# [M-04] Locked liquidity when a game contract is removed from `Staking.authorizedGames`

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The `Staking.removeAuthorizedGame()` function is meant to remove a game contract's authorization so it can no longer interact with the `Staking` contract. However, if there is `lockedLiquidity` locked by a game contract while the contract is being removed from the `authorizedGames` list, the locked liquidity will be permanently locked in the `Staking` contract and will not be allocated to stakers.

```
function transferPayout(address token, address recipient, uint256 amount)
        external
        nonReentrant
        returns (bool)
    {
        require(authorizedGames[msg.sender] || msg.sender == owner
          (), "Caller not authorized");
        //...
            if (callSucceeded) {
             //...
            lockedLiquidity[token] -= amount;
        }
        //...
    }
```

## Recommendation:

To mitigate this issue, the `Staking` contract should track `lockedLiquidity` per game contract, ensuring that a game contract cannot be removed from the `authorizedGames` list if it still has any locked liquidity.

# [M-05] User can frontrun `completeGame` to finalize stake or unstake request

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

Staking user can frontrun `Flip.completeGame()` and finalize his stake/unstake request (based on game result loss or win):

```
function finalizeUnstake(address token) external nonReentrant {
        UnstakeRequest memory request = unstakeRequests[token][msg.sender];
        uint256 sharesToRedeem = request.shares;
        require(sharesToRedeem > 0, "No unstake request found");
        require(
          block.timestamp>=request.requestTime+unstakeDelay,
          "Unstakedelaynotpassed"
        );
        --snipp--
    }
```

```
function finalizeStake(address token) external nonReentrant {
        StakeRequest memory request = stakeRequests[token][msg.sender];
        require(request.amount > 0, "No stake request found");
        require(
          block.timestamp>=request.requestTime+depositDelay,
          "Stakedelaynotpassed"
        );
        --snipp--
    }
```

As a result, the staking user can scape from big losses in the game or participate in big wins.

# Recommendations

To decrease the probability of this attack happening, consider the following:

1. **Let Others Finalize**: Allow other users to finalize a user's stake or unstake request as they have incentives to prevent unfair gains by attackers.

2. **Window Time to Finalize**: Define a deadline for finalizing the transaction. After this deadline, any pending stake or unstake request should can be canceled.

3. **Minimum Share**: Consider using the minimum of share price between the time of request and the time of finalization.

# [M-06] Zero share minting via token balance inflation

## Severity

**Impact:** High

**Likelihood:** Low

## Description

In `Staking.finalizeStake()`, when calculating shares for non-initial deposits (`@1>`), the function uses integer division, which can round down to zero if there are large token balances.

```
function finalizeStake(address token) external nonReentrant {
    --- SNIPPED ---
    // Retrieve the pool's current balance and total shares
    uint256 totalBalance = IERC20(token).balanceOf(address(this));
    uint256 currentShares = tokenInfo[token].totalShares;

    // Use standard share-mint formula: deposit * totalShares / totalBalance
    uint256 mintedShares;
    if (currentShares == 0 || totalBalance == 0) {
        // If no one has staked yet (or balance was 0), do a 1:1 mint
        mintedShares = depositAmount;
    } else {
        // Standard: deposit's fraction of the pool => minted shares
@1>      mintedShares = (depositAmount * currentShares) / totalBalance;
    }

    --- SNIPPED ---
    emit Staked(msg.sender, token, depositAmount);
}
```

This enables a inflation attack for every accepted token where an attacker can:

1. Backrun the `addAcceptedToken()` to be the first one to request a stake for that token with a small amount, i.e. `1 wei`.
2. As the first request staker, the upcoming stakers will consequencely be able to finalize after them and after passing the cooldown periods, the attacker can wait for the right timing to finalize for the attack.
3. Alice requests the stake with `N` tokens and waits for the timing to finalize.
4. Alice sends the tx to finalize the stake, and the attacker frontruns to finalize their 1 wei stake and donates at least `N+1` amount.
5. When Alice's tx is executed, Alice receives 0 shares as the calculation rounds down from the inflated token balance, and the attacker gets that portion of Alice's stake.

This attack is repeatable for newly accepted tokens or when the staking pool of a specific token has low liquidity.

# Recommendation

The possible approaches to mitigate this issue is the following:

- Initial deposit for each staking pool (so that the first depositor attack is not possible), leaving it as dead shares. This should consider the case for the current logic of removal the accepted token restriction.
- Apply virtual shares in the share calculation.

Moreover, applying a minimum staking amount requirement can also make it more difficult to launch the attack.

# 8.3. Low Findings

## [L-01] Staking contract does not consider token decimals

The `Staking` contract does not account for token and share decimals when calculating minted shares, leading to potential rounding errors. Users may receive fewer shares than expected.

```
function finalizeStake(address token) external nonReentrant {
    --snip--
    if (currentShares == 0 || totalBalance == 0) {
        // If no one has staked yet (or balance was 0), do a 1:1 mint
        mintedShares = depositAmount;
    } else {
        // Standard: deposit's fraction of the pool => minted shares
        mintedShares = (depositAmount * currentShares) / totalBalance;
    }
    --snip--
}
```

## [L-02] Missing reqId Deletion in `cancelGame()`

The `cancelGame()` function does not delete `reqId` in the randomness provider (RP). This may lead to leftover request IDs, potentially causing inconsistencies.

```
function cancelGame(string calldata gameId) external onlyPlayerOrOwner
      (gameId) {
        require(isGamePending[gameId], "Game is not pending");
        PendingGame memory game = pendingGames[gameId];
        require(
          block.timestamp>=game.gameStartTime+gameTimeout,
          "Gametimeoutperiodnotreached"
        );

        delete pendingGames[gameId];
        delete isGamePending[gameId];

        _tryTokenTransfer
        //(game.token, game.player, game.betAmount); // Refund bet amount

        stakingContract.unlockLiquidity(game.token, game.netPayout);

        emit GameCancelled(game.player, game.betAmount);
    }
```

# [L-03] Incorrect owed amounts caclulation in `Staking.totalOwed()` function

In the `Staking` contract, the `totalOwed()` function is designed to estimate how many tokens the user's shares represent at the current time. However, the function includes pending deposits balance in the `totalBalance`, which inflates the amount owed for shares and leads to inaccurate calculations, as the pending deposits have not been finalized yet (the shares of these pending deposits haven't been minted yet).

```
function totalOwed(address user, address token) external view returns
      (uint256) {
       //...
        uint256 totalBalance = IERC20(token).balanceOf(address(this));
        return (userShares * totalBalance) / totalShares_;
    }
```

Recommendation: consider tracking the pending deposits balance and subtracting it from the `totalBalance` when calculating the amounts owed to users.

# [L-04] Unstakers unable to receive staked amount if blacklisted by staking token

In the `Staking` contract, the `finalizeUnstake()` function attempts to send the unstaked amount to the staker's address (`msg.sender`). However, if the unstaker's address is blacklisted by the staking token, they will be unable to receive their staked amount, as the transfer will fail due to the blacklisting:

```
function finalizeUnstake(address token) external nonReentrant {
    //...
    IERC20(token).safeTransfer(msg.sender, amountOwed);
    //...
  }
```

Recommendation: consider updating the `finalizeUnstake()` function to include a `receiver` argument, which would allow the unstaked amount to be sent to a different address.

# [L-05] Claim failure for blacklisted addresses in `claimPendingPayout` and `claimPendingWithdrawal`

The `Staking.claimPendingPayout()` function is designed to allow winners to claim their rewards if the transfer failed during `transferPayout()`, a reason for the failure could be that the winner's address is blacklisted, causing the transfer to fail. However, this function will not work for blacklisted addresses because the transfer will fail again, preventing the winner from claiming their rewards as it tries to transfer the tokens to the same address that has been denied in the first trial:

```
function claimPendingPayout(address token) external nonReentrant {
        uint256 amount = pendingPayouts[token][msg.sender];
        //...
        (bool success, bytes memory data) =
            token.call(abi.encodeWithSelector
              (IERC20.transfer.selector, msg.sender, amount));
        //...
    }
```

The same issue exists in the `Flip.claimPendingWithdrawal()` function, where blacklisted addresses are unable to claim their pending withdrawals due to the failure of the transfer.

Recommendation: update both functions to accept a `receiver` address, this would allow the payout or withdrawal to be redirected to a non-blacklisted

address, ensuring that the reward or withdrawal can still be claimed, even if the original address is blacklisted.

```
-function claimPendingPayout(address token) external nonReentrant {
+function claimPendingPayout
+ (address token,address receiver) external nonReentrant {
        uint256 amount = pendingPayouts[token][msg.sender];
        //...
        (bool success, bytes memory data) =
-           token.call(abi.encodeWithSelector
- (IERC20.transfer.selector, msg.sender, amount));
+           token.call(abi.encodeWithSelector
+ (IERC20.transfer.selector, receiver, amount));
        //...
    }
```

# [L-06] Strict equality to zero share validation blocks token removal

The `Staking.removeAcceptedToken()` function checks if `totalShares` is exactly zero before allowing token removal. This strict equality check enables a malicious user to permanently prevent token removal by maintaining a minimal share balance at negligible cost.

```
function removeAcceptedToken(address token) external onlyOwner {
@>  require
  (tokenInfo[token].totalShares == 0, "Token still has staked shares");
    acceptedTokens[token] = false;
    emit TokenRemoved(token);
}
```

Consider implementing a minimum threshold approach that allows removal if shares are below a dust threshold instead of the strict equalities.

If there are exceptional cases, the rightful owner should be allowed to remove the accepted token without shares restrictions. This is because the accepted flag can block new staking on that token, but still allows for unstaking. However, re-accepting the token again should be more taking care as it can create risks if there are remaining shares.

# [L-07] Games can be completed even after `gameTimeout`

In `Flip` contract, a game can be completed even after `gameTimeout` :

```
function completeGame
      (string calldata gameId, uint256 randomNumber) external nonReentrant {
        require(isGamePending[gameId], "No pending game for this seed");
        require(useVRF, "VRF is not enabled");
        require(msg.sender == address
          (randomnessProvider), "Caller is not the randomness provider");

        _completeGame(gameId, randomNumber);
    }
```

```
function cancelGame(string calldata gameId) external onlyPlayerOrOwner
      (gameId) {
        require(isGamePending[gameId], "Game is not pending");
        PendingGame memory game = pendingGames[gameId];
        require(
          block.timestamp>=game.gameStartTime+gameTimeout,
          "Gametimeoutperiodnotreached"
        );
      --snip--
```

After the game timeout period, a losing player can frontrun `completeGame()` and call `cancelGame()` to escape losing their bet amount.

Recommendations:

Don't allow users to `completeGame()` after `gameTimeout`.

# [L-08] Locked tokens and payout failure when removing token without checking `lockedLiquidity`

In the `Staking` contract, the `removeAcceptedToken()` function allows the owner to remove a token from the `acceptedTokens` list without checking if the token has any `lockedLiquidity`. This causes the `transferPayout()` function to revert when called by a game contract to transfer the token payout to a winner. As a result, the winner loses their winnings, and the tokens become locked in the `Staking` contract, preventing the winner from receiving their payout.

```
function removeAcceptedToken(address token) external onlyOwner {
        require
          (tokenInfo[token].totalShares == 0, "Token still has staked shares");
        acceptedTokens[token] = false;
        emit TokenRemoved(token);
    }
```

```
function transferPayout(address token, address recipient, uint256 amount)
        external
        nonReentrant
        returns (bool)
    {
        //...
        require(acceptedTokens[token], "Token not supported");
        //...
    }
```

Recommendations:

Consider updating the `removeAcceptedToken()` function to check if the token has any `lockedLiquidity` before it is removed from the `acceptedTokens` list, or alternatively, remove the `acceptedTokens` check from the `transferPayout()` function to prevent payout failures.

# [L-09] Inconsistent payout handling due to mixing staker rewards with `lockedLiquidity`

The `Staking.transferPayout()` function allows the **owner** to distribute rewards to **stakers**. However, this function deducts the rewards from the `lockedLiquidity` **that has been locked by game contracts**. This results in inconsistencies when the `lockedLiquidity` becomes insufficient for game contracts, as the owner's distribution of staker rewards affects the liquidity that should only be modified by game contracts.

```
function transferPayout(address token, address recipient, uint256 amount)
        external
        nonReentrant
        returns (bool)
    {
        require(authorizedGames[msg.sender] || msg.sender == owner
          (), "Caller not authorized");
        //...
            if (callSucceeded) {
             //...
            lockedLiquidity[token] -= amount;
        }
        //...
    }
```

Recommendations:

Consider introducing a mechanism to track the rewards distributed to **stakers** separately from the `lockedLiquidity`, to ensure that the `lockedLiquidity` remains exclusively controlled by the game contracts.

## [L-10] Player loss due to `netPayout` less than `betAmount` for some combinations

In the `Flip` contract, during the `netPayout` calculation in the `flip()` function, it was noticed during testing that the user's `netPayout` is sometimes less than the `betAmount` they paid to participate in the game. For certain coin/heads combinations, the user ends up with a payout lower than the amount they bet, resulting in a loss for the player rather than a win, which makes it non-rewarding for players even if they win.

A `CoinFlipTest` test file is setup with a 10% `feePercentage` (`feePercentage` = 1000) and with a `betAmount` of 100 (where the bet token is assumed to have 18 decimals), to observe at which `numberOfCoins` to `headsRequired` combinations the calculated `netPayout` (reward) starts to be less than the game `betAmount`:

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";

contract CoinFlipTest is Test {
    uint256 public feePercentage = 1000;

    function calculatePayout(
        uint256 betAmount,
        uint256 numberOfCoins,
        uint256 headsRequired
    )
        public
        view
        returns (uint256 grossPayout, uint256 netPayout, uint256 fee)
    {
        require(numberOfCoins > 0, "Number of coins must be greater than 0");
        require(
            headsRequired > 0 && headsRequired <= numberOfCoins,
            "Invalid number of heads required"
        );

        // Calculate total number of possible outcomes (2^numberOfCoins)
        uint256 totalOutcomes = 2 ** numberOfCoins;

        // Calculate favorable outcomes
        //(where headsRequired or more heads occur)
        uint256 favorableOutcomes = 0;

        for (uint256 i = headsRequired; i <= numberOfCoins; i++) {
            favorableOutcomes += binomialCoefficient(numberOfCoins, i);
        }

        uint256 odds = (totalOutcomes * 1e18) / favorableOutcomes;

        // Corrected gross payout calculation
        grossPayout = (betAmount * odds) / 1e18; // Removed betAmount addition

        // Fee based on original bet amount
        fee = (betAmount * feePercentage) / 10000;
        netPayout = grossPayout - fee;

        return (grossPayout, netPayout, fee);
    }

    function binomialCoefficient(
        uint256 n,
        uint256 r
    ) internal pure returns (uint256) {
        if (r > n) {
            return 0;
        }
        if (r == 0 || r == n) {
            return 1;
        }

        uint256 numerator = 1;
        uint256 denominator = 1;

        // Calculate nCr = n! / (r! * (n - r)!)
        for (uint256 i = 0; i < r; i++) {
            numerator = numerator * (n - i);
            denominator = denominator * (i + 1);
        }
        return numerator / denominator;
    }
```

```
    function test_PayoutCalculation(
        uint256 numberOfCoins,
        uint256 headsRequired
    ) public {
        // same checks implemented in `Flip.flip()`
        vm.assume(numberOfCoins >= 1 && numberOfCoins <= 20);
        vm.assume(headsRequired >= 1 && headsRequired <= numberOfCoins);
        uint256 betAmount = 100e18;

        (, uint256 netPayout, ) = calculatePayout(
            betAmount,
            numberOfCoins,
            headsRequired
        );
        console.log(
            "numberOfCoins:headsRequired :",
            numberOfCoins,
            headsRequired
        );
        console.log("betAmount:netPayout :", betAmount, netPayout);
        console.log("--------------------");

        assert(netPayout >= betAmount);
    }
}
```

The result shown below that the test starts to fail (`netPayout < betAmount`) with `numberOfCoins = 9 and headsRequired = 3` :

```
$ forge test --mt test_PayoutCalculation -vvvvv

Ran 1 test for test/CoinFlipTest.t.sol:CoinFlipTest
[FAIL: panic: assertion failed (
  0x01
); counterexample: calldata=0xe8ea7ea900000000000000000000000000000000000000000000000000000000
μ: 26196, ~: 26196)
Logs:
  numberOfCoins:headsRequired : 9 3
  betAmount:netPayout : 10000000000000000000 9987124463519313300 0
  --------------------

Traces:
  [49533] CoinFlipTest::test_PayoutCalculation(9, 3)
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← [Return]
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← [Return]
    ├─ [0] console::log("numberOfCoins:headsRequired :", 9, 3) [staticcall]
    │   └─ ← [Stop]
    ├─ [0] console::log(
      "betAmount:netPayout:",
      10000000000000000000[1e20],
      9987124463519313300[9.987e19]
    ) [staticcall]
    │   └─ ← [Stop]
    ├─ [0] console::log("--------------------") [staticcall]
    │   └─ ← [Stop]
    └─ ← [Revert] panic: assertion failed (0x01)

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 32.16ms
  (31.60ms CPU time)

Ran 1 test suite in 55.82ms
  (32.16ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/CoinFlipTest.t.sol:CoinFlipTest
[FAIL: panic: assertion failed (
  0x01
); counterexample: calldata=0xe8ea7ea900000000000000000000000000000000000000000000000000000000
μ: 26196, ~: 26196)

Encountered a total of 1 failing tests, 0 tests succeeded
```

When tested separately for `numberOfCoins = 10 and headsRequired = 3`

```
function test_testPayoutForASpecificValues() public {
        uint256 betAmount = 100e18;
        uint256 numberOfCoins = 10;
        uint256 headsRequired = 3;
        (, uint256 netPayout, ) = calculatePayout(
            betAmount,
            numberOfCoins,
            headsRequired
        );
        console.log("betAmount:netPayout :", betAmount/1e18, netPayout/1e18);
    }
```

the result is (`netPayout = 95e18` while `betAmount = 100e18` ):

```
$ forge test --mt test_testPayoutForASpecificValues -vvvvv

Ran 1 test for test/CoinFlipTest.t.sol:CoinFlipTest
[PASS] test_testPayoutForASpecificValues() (gas: 53257)
Logs:
  betAmount:netPayout : 100 95

Traces:
  [53257] CoinFlipTest::test_testPayoutForASpecificValues()
    ├─ [0] console::log("betAmount:netPayout :", 100, 95) [staticcall]
    │   └─ ← [Stop]
    └─ ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 12.12ms
  (11.56ms CPU time)

Ran 1 test suite in 34.93ms
  (12.12ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendations:

Consider updating the `flip()` function to revert the transaction if the calculated `netPayout` is less than the `betAmount`.