Coinflip Security Review

Pashov Audit Group

Conducted by: Shaka, Udsen, peanuts

February 5th 2025 - February 8th 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 **play-turbo/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 - e8bc317e5cb0a7b27c91d23fc47c83b4524bfa8e

fixes review commit hash - a3b2f1e75eed8c76aa3c4e03211540eb9b2aca99

Scope

The following smart contracts were in scope of the audit:

- `Flip`
- `Staking`

7. Executive Summary

Over the course of the security review, Shaka, Udsen, peanuts engaged with Coinflip to review Coinflip. In this period of time a total of **20** issues were uncovered.

Protocol Summary

Protocol Name	Coinflip
Repository	https://github.com/play-turbo/coinflip-contracts
Date	February 5th 2025 - February 8th 2025
Protocol Type	Game

Findings Count

Severity	**Amount**
Critical	2
High	3
Medium	5
Low	10
Total Findings	**20**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Check for Staking liquidity in flip() does not account for other game contracts	Critical	Resolved
[C-02]	Bet amount remains locked in the Flip contract when player wins	Critical	Resolved
[H-01]	Staking.earned calculates the rewards wrongly	High	Resolved
[H-02]	giveReward() does not update the stake balances	High	Resolved
[H-03]	Users can potentially steal other user's staking rewards	High	Resolved
[M-01]	Potential for increased fees due to delayed transaction execution	Medium	Resolved
[M-02]	Changing fee percentage after game start can block payouts	Medium	Resolved
[M-03]	PythRandomnessProvider uses always the default provider	Medium	Resolved
[M-04]	Players may not get paid if liquidity is less than winnings	Medium	Resolved
[M-05]	Game completion can be front-run by stakers	Medium	Acknowledged
[L-01]	Excessive msg.value is not transferred to the player after the requestRandomness call	Low	Resolved
[L-02]	Coinflip game does not work with fee-on-transfer tokens	Low	Acknowledged

[L-03]	Accepted tokens in Staking cannot be removed	Low	Resolved
[L-04]	Loss of precision in the calculation of user's share in earned()	Low	Resolved
[L-05]	Native tokens sent to Flip will be stuck in the contract	Low	Resolved
[L-06]	Stake.timeUntilUnstake() does not check if the user has staked any tokens	Low	Resolved
[L-07]	Same userRandomNumber can be used for different requests to pyth entropy	Low	Resolved
[L-08]	Use of transfer and transferFrom could revert transaction	Low	Resolved
[L-09]	Shares burned in Staking.unstake() can be underestimated	Low	Resolved
[L-10]	The entropyCallback may fail in blacklist transfer edge case	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Check for `Staking` liquidity in `flip()` does not account for other game contracts

Severity

Impact: High

Likelihood: High

Description

`Flip.flip()` checks that the `Staking` contract has enough funds to cover the player's reward in the case of a win. However, it is not taken into account that other game contracts might have also funds locked in the `Staking` contract.

As a result, the total liquidity locked among all game contracts might exceed the total balance of the `Staking` contract, which would cause the `Staking` contract not to be able to cover all the rewards in the case of a win.

Recommendations

In the `flip()` function check that the `Staking` contract balance is enough to cover the locked liquidity of all game contracts and not just the locked liquidity of the current game contract.

[C-02] Bet amount remains locked in the `Flip` contract when player wins

Severity

Impact: High

Likelihood: High

Description

When a game is completed and the player wins, a percentage of the bet amount (fee) is sent to the manager and/or the staking contract. But the rest of the bet amount remains locked forever.

This means that for every game won, the protocol loses `netPayout` tokens, while for every game lost, the protocol gains `betAmount` tokens.

Considering a scenario where the parameters are set to provide the maximum profitability for the protocol we would have the following:

- feePercentage = 10%
- managerWinFeePercentage = 0%
- liquidityEdge = 0%

A player bets 100 tokens for 1 head out of 1 flip, so the chance of winning is 50%.

- If the player wins, Staking contract sends 190 tokens to the player, 10 tokens (fee) are transferred from Flip to Staking contract, and 90 tokens get locked in the Flip contract. The result for the protocol is a net loss of 180 tokens.
- If the player loses, the protocol gains 100 tokens from the player bet.

As we can see, the protocol is not sustainable in the long term.

Recommendations

```
uint256 managerFee = (fee * managerWinFeePercentage) / 10000;
-        uint256 stakingFee = fee - managerFee;
+        uint256 stakingAmount = game.betAmount - managerFee;

         if (managerFee > 0) {
             game.token.safeTransfer(manager, managerFee);
         }
-        if (stakingFee > 0) {
-            game.token.safeTransfer(address(stakingContract), stakingFee);
-        }
+        game.token.safeTransfer(address(stakingContract), stakingAmount);
```

8.2. High Findings

[H-01] Staking.earned calculates the rewards wrongly

Severity

Impact: High

Likelihood: Medium

Description

`Staking.giveReward()` is called by the owner or manager to distribute rewards to users. The function calls `earned()`, which calculates rewards for the user even if the user is in deficit. The calculation is as such:

```
function earned(address user, address token) public view returns (uint256) {
        require(acceptedTokens[token], "Token not supported");
        TokenInfo storage info = tokenInfo[token];
        if (info.totalStaked == 0) {
            return 0;
        }
        uint256 totalTokenBalance = IERC20(token).balanceOf(address(this));

        // Multiply by 1e18 first to maintain precision
>       uint256 userShare =
  (stakedBalances[token][user] * 1e18 / info.totalStaked) * totalTokenBalance / 1e18;

        // If contract has lost balance, return user share of remaining balance
        if (totalTokenBalance < info.totalStaked) {
            return userShare;
        }

        // Otherwise return the difference between user's share and their staked
        // balance

                 return userShare > stakedBalances[token][user] ? userShare - stakedBa
    }
```

`earned()` takes the totalToken balance, staked balance of the user, and totalStaked to calculate the rewards.

Let's say there are 100 tokens in the contract, and 100 staked. Alice stakes 10 tokens. Assume 6 decimal tokens.

There are two scenarios from here. First, the staking contract earns token (by users losing the game and the bet amount funneling into the staking contract)

There is now 120 tokens in the contract, with 100 totalStaked.

Alice userShare is now 12e6.

```
userShare =
  (stakedBalances[token][user] * 1e18 / info.totalStaked) * totalTokenBalance / 1e18;
userShare = 10e6 * 1e18 / 100e18 * 120e18 / 1e18 = 12e6
```

Since `totalTokenBalance (120) > totalStaked (100)`, `earned()` returns 2e6. Alice reward is 2e6 in this scenario

```
return
   userShare > stakedBalances[token][user] ? userShare - stakedBalances[token][user] :
return 12e6 > 10e6 ? 12 - 10 = 2
```

Now, if the scenario is flipped and there is now 80 tokens in the contract with 100 totalStaked:

Alice userShare is now 8e6

```
userShare =
  (stakedBalances[token][user] * 1e18 / info.totalStaked) * totalTokenBalance / 1e18;
userShare = 10e6 * 1e18 / 100e18 * 80e18 / 1e18 = 8e6
```

Since `totalTokenBalance (80) < totalStaked (100)`, return `userShare` which is 8e6

Even though the staking contract lost tokens, Alice reward is now 8e6.

Recommendations

Not sure what `giveReward()` does exactly or when it would be called, since it is access controlled, but if `giveReward()` is called, it will still reward users when the staking contract loses money.

Consider returning 0 if `totalTokenBalance < info.totalStaked`

```
// If contract has lost balance, return user share of remaining balance
        if (totalTokenBalance < info.totalStaked) {
            return 0;
        }
```

[H-02] `giveReward()` does not update the stake balances

Severity

Impact: High

Likelihood: Medium

Description

In the `Staking` contract, the `giveReward()` function can be called by the owner or the manager to distribute the profit of a staker. However, this function does not update the stake balance of the staker and the total staked amount. As a result, a portion of other stakers' rewards will be distributed to the staker whose reward is being given.

Recommendations

Update the stake balance of the staker and the total staked amount in the `giveReward()` function.

[H-03] Users can potentially steal other user's staking rewards

Severity

Impact: High

Likelihood: Medium

Description

When a user calls `Staking.stake()`, `stakedBalance` of the user is equal to the amount deposited.

```
function stake(address token, uint256 amount) external nonReentrant {
        require(acceptedTokens[token], "Token not supported");
        require(amount > 0, "Cannot stake 0");

        TokenInfo storage info = tokenInfo[token];
        info.totalStaked += amount;
>       stakedBalances[token][msg.sender] += amount;
        lastStakeTime[token][msg.sender] = block.timestamp;
>       require(IERC20(token).transferFrom(msg.sender, address
  (this), amount), "Stake transfer failed");
        emit Staked(msg.sender, token, amount);
    }
```

When the user `unstakes()`, the `stakedBalance` is subtracted by calculating `userShares`:

```
uint256 totalTokenBalance = IERC20(token).balanceOf(address(this));

>       uint256 userShare = (userStaked * totalTokenBalance) / info.totalStaked;
        require(amount <= userShare, "Cannot unstake more than share");
        uint256 fraction = (amount * 1e18) / userShare;

        // Decrease user's staked balance proportionally
>       uint256 sharesToBurn = (userStaked * fraction) / 1e18;
>       stakedBalances[token][msg.sender] -= sharesToBurn;
>       info.totalStaked -= sharesToBurn;

        // Transfer amount to user
        require(IERC20(token).transfer
          (msg.sender, amount), "Unstake transfer failed");
```

For example, if Alice deposits 100 USDC, her `stakedBalance` will be 100 and the total contract balance will be 100. Let's say the total contract balance of USDC increases to 200, when Alice calls `unstake()`, she will get back 200 USDC since her `userShare` will become 200e6:

```
Alice calculation
userShare = 100e6 * 200e6 / 100e6 = 200e6
amount = 200e6
fraction = 200e6 * 1e18 / 200e6 = 1e18
sharesToBurn = 100e6 * 1e18 / 1e18 = 100e6
stakedBalance = 100e6 - 100e6 = 0
totalStaked = 100e6 - 100e6 = 0

amount transferred = 200e6
```

However, if alice decides not to unstake and someone calls `stake()` when the contract balance increases to 200, they can steal Alice rewards.

Let's say totalStaked is now 100e6 and total USDC balance is now 200e6. Bob decides to stake 400 USDC. Assuming Alice doesn't call `unstake()` or the cooldown time to unstake is low, when Bob calls unstake, he can get Alice rewards. When bob stakes 400 USDC, bob stakedBalance = 400e6, totalStaked = 500e6 and total USDC balance is 600e6

```
Bob's calculation
userShare = 400e6 * 600e6 / 500e6 = 480e6
amount = 480e6
fraction = 1e18
sharesToBurn = 400e6 * 1e18 / 1e18 = 400e6
stakedBalance = 400e6 - 400e6 = 0
totalStaked = 500e6 - 400e6 = 100e6

amount transferred = 480e6
```

Bob gets 480e6 from his 400e6 stake, and Alice stake is now worth 120e6 instead of 200e6.

Recommendations

When staking, consider calculating the `userShare` as well to ensure that the later stakers does not gain the rewards of the earlier ones.

One example would be when Bob stakes his 400e6 USDC, the ratio is at 2:1 (totalTokenBalance / totalStakedBalance), so Bob should only get 200 stakedBalance.

When Bob unstakes, his `userShare` will be `200e6 * 600e6 / 300e6 = 400e6`

8.3. Medium Findings

[M-01] Potential for increased fees due to delayed transaction execution

Severity

Impact: Medium

Likelihood: Medium

Description

In the `Flip.sol` contract, the `PendingGame` struct is initialized with `gameStartTime` set to `block.timestamp` at the time of the flip function call as shwon below:

```
pendingGames[gameId] = PendingGame({
            player: msg.sender,
            betAmount: betAmount,
            numberOfCoins: numberOfCoins,
            headsRequired: headsRequired,
            netPayout: netPayout,
            token: token,
            gameStartTime: block.timestamp // Record game start time
        });
```

This creates a vulnerability where the `flip transaction` could be stuck in the mempool and executed at a later time, potentially after the `feePercentage has been increased` by the owner. As a result, the player could incur `higher fees` on his winning amount than he initially anticipated, affecting his net payout. This situation is beyond the player's control and could lead to scenarios where the player would not have participated in the game had they known the fees would be higher in the first place.

The actual calculation of the winning payout and fee occurs in the `calculatePayout` function. The fee is calculated using the `feePercentage` and it is deducted from the `grossPayout` to calculate the `netPayout` as shown below:

```
fee = (betAmount * feePercentage) / 10000;
        netPayout = grossPayout - fee;
```

Recommendations

Introduce a `deadline parameter` to the `flip function`. This parameter should specify the latest acceptable `block timestamp` for the transaction to be executed. If the `transaction is executed after this deadline`, it should revert. This ensures that the player has control over the game play and accepts the risk and result based on the contract state at the time they initiate the transaction, preventing unexpected fee increases and other state changes.

[M-02] Changing fee percentage after game start can block payouts

Severity

Impact: High

Likelihood: Low

Description

In `Flip.sol`, the `flip()` function calculates the `netPayout` and locks the amount to ensure that the contract has enough funds to pay the winner.

However, when the game is completed, in case the player wins, the `netPayout` is recalculated.

```
            if (won) {
                (uint256 grossPayout, uint256 netPayout, uint256 fee) =
                    calculatePayout
  (game.betAmount, game.numberOfCoins, game.headsRequired);
```

The `feePercentage` is used in the calculation of `netPayout`, so if the owner changes the fee percentage after the game is started, the `netPayout` will be calculated with the new fee percentage, which can lead to the new `netPayout` being different from the amount calculated in the `flip()` function.

If `feePercentage` is increased, the user would receive a smaller payout than expected. If `feePercentage` is decreased, the new `netPayout` could exceed the locked liquidity, causing the transaction to revert.

Recommendations

Add the `fee` to the `PendingGame` struct in the `flip()` function and use the stored values of `netPayout` and `fee` instead of recalculating them in the `_completeGame()` function.

[M-03] `PythRandomnessProvider` uses always the default provider

Severity

Impact: Low

Likelihood: High

Description

The `PythRandomnessProvider` contract allows the owner to set a specific provider for the entropy.

```
function setProvider(address _provider) external onlyOwner {
        require(_provider != address(0), "Zero address not allowed");
        provider = _provider;
        emit ProviderUpdated(_provider);
    }
```

However, the `provider` variable is ignored and the default provider is always used in the interactions with the `entropy` contract. As a result, the owner does not have the ability to change the provider used to generate the entropy.

Recommendations

Use the `provider` variable when its value is different from address(0) instead of `entropy.getDefaultProvider()`. Additionally, replace the `sequenceNumberToInfo` mapping for a new `providerToSequenceNumberToInfo` mapping.

[M-04] Players may not get paid if liquidity is less than winnings

Severity

Impact: High

Likelihood: Low

Description

When `flip()` is called, the function checks whether there is available liquidity to pay the potential winnings of a game:

```
// require staked amount > maximum payout using calculatePayout
        (uint256 grossPayout, uint256 netPayout,) = calculatePayout
        //(betAmount, numberOfCoins, headsRequired); // TODO check use grossPayout?

        uint256 stakingBalance = IERC20(token).balanceOf(address
          (stakingContract));
        uint256 availableLiquidity =
          (stakingBalance * maxLiquidityPercentage) / 10000;
        require(
          lockedLiquidity+netPayout<=availableLiquidity,
          "Notenoughliquidityfornewbet"
        );

        lockedLiquidity += netPayout;
```

For example, when tested in remix, a 20 coin 20 heads game for 1 USD will yield a payout of 1048575.98 (1 million) USDC. The function checks that there is `availableLiquidity` and then checks the total `lockedLiquidity` and ensures that the staking contract can pay out that amount.

Assume a whale deposits 1.1 million USDC in the staking contract. 2 days later (the unstake cooldown duration), a player calls `flip(1e6,20,20,USDC)`. The `flip()` call is executed, and the player awaits the pyth entropy callback.

Unfortunately, the whale decides to withdraw at this exact moment, and the entropy callbacks after the withdraw. The player actually won the game yielding him 1 million+ USDC, but he cannot withdraw it since there is not enough liquidity in the staking contract anymore.

Recommendations

The staking contract should account for the `lockedLiquidity`. When a user intends to unstake. If the total balance of the staking contract falls below `lockedLiquidity`, then the user should not be able to unstake as well.

[M-05] Game completion can be front-run by stakers

Severity

Impact: High

Likelihood: Low

Description

Stakers in the protocol profit from players losing their bets and incur losses when players win.

Users can front-run the completion of a lost game by staking tokens, achieving an unrealized profit. While the staker will have to wait for the cooldown period to unstake, and in that period, the unrealized profits can be lost, the repetition of this process in the long term is statistically profitable.

Consider the following scenario:

- `Staking` contract has 100 tokens deposited
- Alice bets 50 tokens on a coin flip
- In the following block, Bob sees in the mempool the transaction where `PythRandomnessProvider.entropyCallback` is called and verifies that the random number will make Alice lose
- Bob front-runs the transaction and stakes 10,000 tokens
- The game is completed and Alice’s bet amount is transferred to `Staking` contract
- Bob’s stake is now worth 10,049 tokens
- Bob repeats the process every time he detects a losing game

In the same way, a staker who’s cooldown period has elapsed, can track the mempool and unstake before a game is completed with a win for the player, and avoid the loss.

Recommendations

There is no straightforward solution to this issue. A possible mitigation would pass for accounting for the new amount staked is added to `info.totalStaked` and `stakedBalances[token][msg.sender]` only after `n` blocks have passed since the user staked the tokens. For unstaking, it would be required to add a delay between the request and the actual unstaking process. However, such an implementation would require taking special care to avoid a denial of service for the stakers.

8.4. Low Findings

[L-01] Excessive `msg.value` is not transferred to the player after the `requestRandomness` call

The `pythRandomnessProvider::requestRandomness` function is called by the `Flip::flip` contract to retrieve the `random number` to compute whether the player has won the game. The `requestRandomness` function calls the `entropy::requestWithCallback` function with the respective `fee amount for the default provider` as shown below:

```
uint64 sequenceNumber = entropy.requestWithCallback{value: msg.value}
        (entropyProvider, userRandomNumber);
```

The `feeAmount` is passed onto this call via the `Flip::flip` as shown below:

```
uint256 fee = randomnessProvider.getFee();

            require(msg.value >= fee, “Insufficient fee for randomness”);
```

The issue here is that the returned fee amount by the `randomnessProvider.getFee()` changes based on the prevailing gas price on-chain as mentioned in the pyth documentation.

> The following tables shows the total fees payable when using the default provider. Note that the fees shown below will vary over time with prevailing gas prices on each chain.

https://docs.pyth.network/entropy/current-fees

Hence by the time the `flip transaction is executed` if the `default provider fee is decreased` the excessive `msg.value` will be stuck in the `Flip.sol` contract.

Hence it is recommended to transfer the `excessive fee amount (msg.value - fee)` back to the `player (msg.sender)` after the `randomnessProvider.requestRandomness` call is made.

[L-02] Coinflip game does not work with `fee-on-transfer` tokens

The `Flip.sol` contract of the `Coinflip game` has the core logic of the game play. In the `Flip.sol` contract the `acceptedTokens` mapping is used to whitelist a token for betting by the `owner`.

When an user plays the game he will call the `Flip::flip` function which transfers the `betAmount` of the respective accepted token from the `msg.sender` to the `Flip.sol` contract as shown below:

```
token.safeTransferFrom(msg.sender, address(this), betAmount);
```

And this same `betAmount` is used to create the `pendingGame` without accounting for any transfer fees (in the event token used is a fee-on-transfer token). And the same `betAmount` is used to refund to the user in the event `randomnessProvider.requestRandomness` call fails. Both above logics are shown below:

```
pendingGames[gameId] = PendingGame({
            player: msg.sender,
            betAmount: betAmount,
            numberOfCoins: numberOfCoins,
            headsRequired: headsRequired,
            netPayout: netPayout,
            token: token,
            gameStartTime: block.timestamp // Record game start time
        });
```

```
token.safeTransfer(msg.sender, betAmount);
                revert(string.concat("Randomness request failed: ", reason));
```

If the token is `fee-on-transfer` then the actual amount transferred to the contract will be less than the `betAmount` and as a result `using exact betAmount` for `pendingGame creation` and `refund` is erroneous which could lead to `DoS due to lack of funds` or `could steal funds` from other user staked amounts.

Hence it is recommended to update the `betAmount` transfer logic in the `Flip::flip` function to compute the correct balance of the tokens transferred to the contract. This can be done by taking `before and after balance` during the `token transfer`. If the `fee-on-transfer` tokens are not `accepted by the coinflip game` then it is recommended to document it properly.

[L-03] Accepted tokens in `Staking` cannot be removed

The owner of the `Staking` contract can add accepted tokens for staking. However, there is no way to remove them. If an accepted token is removed in the `Flip` contract, the `Staking` contract will still accept staking the token, even though it will not generate new rewards.

Consider adding a way to remove accepted tokens for staking.

[L-04] Loss of precision in the calculation of user's share in `earned()`

To calculate the user's share in the `Staking.earned()` function, the user's stake balance is divided by the total staked balance before multiplying by the total token balance. This can lead to a loss of precision in the calculation.

```
        // Multiply by 1e18 first to maintain precision
        uint256 userShare =
  (stakedBalances[token][user] * 1e18 / info.totalStaked) * totalTokenBalance / 1e18;
```

Consider the following example:

- user's stake balance: 1e18
- total staked balance: 2e18 + 1
- total token balance: 2e18 + 1
- stakedBalances[token][user] * 1e18 / info.totalStaked = 1e18 * 1e18 / (2e18 + 1) = 499999999999999999
- userShare = 499999999999999999 * (2e18 + 1) / 1e18 = 999999999999999998

The user's share is calculated as 999999999999999998 instead of 1e18 due to the loss of precision in the calculation.

The calculation of the user's share should be done in the same way it is done in the `unstake()` function to avoid the loss of precision.

```
-  uint256 userShare =
- (stakedBalances[token][user] * 1e18 / info.totalStaked) * totalTokenBalance / 1e18;
+
+   uint256 userShare = stakedBalances[token][user] * totalTokenBalance / info.totalSt
```

[L-05] Native tokens sent to `Flip` will be stuck in the contract

The `Flip` contract implements a `receive()` function that allows anyone to send native tokens to the contract. However, the contract does not have a mechanism to withdraw these tokens, meaning that they will be stuck in the contract forever.

Consider using these native tokens to pay for the fees of the randomness provider or removing the `receive()` function.

[L-06] `Stake.timeUntilUnstake()` does not check if the user has staked any tokens

`Stake.timeUntilUnstake()` returns how long a user must wait before they can unstake their tokens. The function returns 0 if the user can unstake their tokens immediately. However, it is not checked if the user has staked any tokens. If the user has not staked any tokens `lastStakeTime` will be 0, so it will be considered that the cooldown period has passed and the function will return 0.

Consider reverting the transaction if the user has not staked any tokens.

[L-07] Same `userRandomNumber` can be used for different requests to pyth entropy

The game ID is calculated in `Flip.flip()` as follows:

```
bytes32 gameIdBytes =
            keccak256(abi.encodePacked
                (msg.sender, block.timestamp, numberOfCoins, headsRequired, nonce++));
    string memory gameId = string(abi.encodePacked(gameIdBytes));
```

This value is used in `PythRandomnessProvider.requestRandomness()` to calculate the user random number that will be sent in the call to `entropy.requestWithCallback()`:

```
bytes32 userRandomNumber = keccak256(abi.encodePacked
         (gameId, block.timestamp));

    uint64 sequenceNumber = entropy.requestWithCallback{value: msg.value}
    //(entropyProvider, userRandomNumber); // CHANGED FROM GETFEE
```

There is the possibility that the same `userRandomNumber` is used for different requests if a user initiates multiple games from different contracts and uses the same parameters, in the same block. If this was the case, a malicious entropy provider would have it easier to manipulate the result of the randomness request.

It is recommended to include the address of the game contract in the `userRandomNumber` calculation to avoid this issue.

```
-    bytes32 userRandomNumber = keccak256(abi.encodePacked
- (gameId, block.timestamp));
+    bytes32 userRandomNumber = keccak256(abi.encodePacked
+ (gameId, block.timestamp, msg.sender));
```

[L-08] Use of `transfer` and `transferFrom` could revert transaction

There are multiple occassions in the `staking.sol` contract where the `IERC20::transferFrom` and `IERC20::transfer` functions are used to transfer the `tokens` between `msg.sender` and contract itself.

But the issue is if the `token` being transferred does not return `bool` the above transfer transaction will fail since the `IERC20` implementation expects a `bool` value as the return value.

Since the `coin flip` game expects to work with multiple tokens whitelisted by the owner, this could be an issue.

Hence this will break the core functionalities of the game when such tokens are used in the game.

Hence it is recommended to use the `safeTransfer` and `safeTransferFrom` from the openzeppelin `SafeERC20` library.

[L-09] Shares burned in `Staking.unstake()` can be underestimated

In the `Staking.unstake()` function, the calculation of the shares to be burned for the user is done as follows:

```
79:         uint256 userShare =
  (userStaked * totalTokenBalance) / info.totalStaked;
80:         require(amount <= userShare, "Cannot unstake more than share");
81:         uint256 fraction = (amount * 1e18) / userShare;
82:
83:         // Decrease user's staked balance proportionally
84:         uint256 sharesToBurn = (userStaked * fraction) / 1e18;
```

The results of the division operations in lines 81 and 84 are rounded down due to the truncation of the division operation, meaning that the shares burned by the user might be underestimated.

Let's consider the following example:

- amount = 1
- userStaked = 10
- totalStaked = 100
- totalTokenBalance = 110
- userShare = 10 * 110 / 100 = 11
- fraction = 1 * 1e18 / 11 = 90909090909090909
- sharesToBurn = 10 * 90909090909090909 / 1e18 = 0

In this case, the user is able to withdraw 1 token, but the shares burned are 0.

The minimum unit of a token is usually a negligible amount, but in the case of tokens with few decimals and high values, the amount can be significant. For example, in WBTC (8 decimals) the minimum unit is valued at approximately 0.001 USD (at current prices) and, the minimum unit of GUSD (2 decimals) is valued at 0.01 USD. In an extreme case where a token with few decimals and high value is used, this attack vector could be exploited to drain the staking pool.

Round up the calculation of `fraction` and `sharesToBurn`, capping the final value to the staked balance of the user.

[L-10] The entropyCallback may fail in blacklist transfer edge case

Pyth entropy documentation states:

> The entropyCallback function should never return an error. If it returns an error, the keeper will not be able to invoke the callback.

There can be an unfortunate case where the `entropyCallback` function reverts because the user is blacklisted from receiving his rewards.

When `entropyCallback()` is called in the RandomnessProvider contract, it calls `IGame(info.gameContract).completeGame(info.gameId, random);` which transfers tokens to the player if the player wins:

```
Flip.sol

            if (managerFee > 0) {
                game.token.safeTransfer(manager, managerFee);
            }
            if (stakingFee > 0) {
                game.token.safeTransfer(address(stakingContract), stakingFee);
            }

>           stakingContract.transferPayout(game.token, game.player, netPayout);

Staking.sol

function transferPayout(
  addresstoken,
  addressrecipient,
  uint256amount
) external nonReentrant returns (bool
        require(authorizedGames[msg.sender] || msg.sender == owner
          (), "Caller not authorized");
        require(acceptedTokens[token], "Token not supported");
        require(amount > 0, "Cannot transfer 0");
>       require(IERC20(token).transfer
  (recipient, amount), "Payout transfer failed");
        return true;
    }
```

In an edge case, if the user gets blacklisted by the token in the duration of waiting for the callback, the callback will fail since the transfer will fail, and there will be no retry.

For example, a player calls `flip()` with USDC as his token and gets blacklisted immediately after. During the `entropyCallback()`, funds cannot be transferred to the player and `entropyCallback()` will revert.

Recommend having a retry mechanism to ensure maximum fairness, since in this case the player had won. Although the player can call `cancelGame()` to retrieve his tokens after the blacklist has been lifted, it would not be fair to the player.