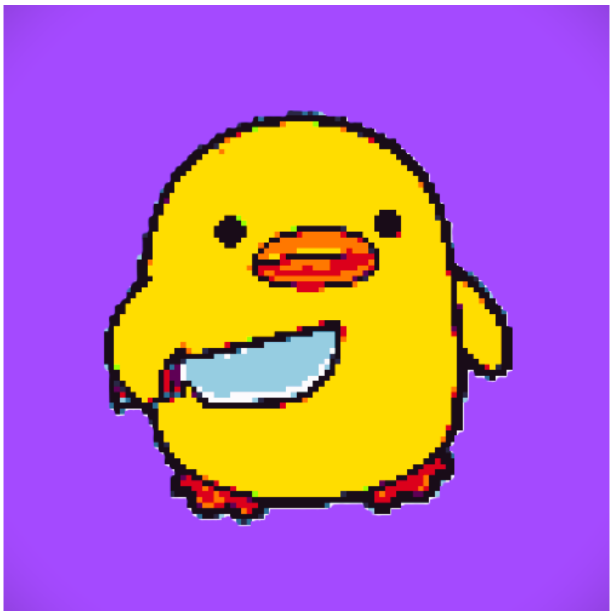

TSwap Protocol Audit Report

Version 1.0

Vesko.io

September 29, 2024

Protocol Audit Report

Vesko.io

June 18, 2024

Prepared by: Vesko Lead Auditors: - Veselin Vachkov

Table of Contents

* [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants
* [L-1] Minimum Pragma Version for Using `openzeppelin` is Now 0.8.21
* [G-1] Gas Optimization in Active Player Check
* [I-1] Non-Descriptive Function Name
* [I-2] Move Static Parts of JSON to Constants

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The Veselin's team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

```
./src/
Puppyraffle.sol
```

Roles

- Owner: The user who can set the password and read the password.
- Outsiders: No one else should be able to set or read the password.

Executive Summary

Issues found

Severtity	Number of issues found
High	9
Medium	1
Low	1
Info	3
Total	14

[H-1] The `newPlayers` variable is passed that is not checked for duplicates instead use `players`

Description: The enterRaffle function is designed to add new players to a raffle and ensure that there are no duplicate entries. However, the current implementation has a couple of issues:

Checking for Duplicates After Adding Players:

- The players are added to the players array before checking for duplicates. This can lead to temporary states where duplicates exist within the players array, which might be problematic for other parts of the contract or during interactions within the same transaction.

Emission of RaffleEnter Event with Incorrect Data:

- The event RaffleEnter is emitted with newPlayers instead of the players array. This does not accurately reflect the state of the raffle participants, particularly after the duplicate check.

Impact: Duplicate Entries: - Adding duplicate players before performing the check can lead to inconsistencies and potential errors in other parts of the contract or during interactions within the same transaction. Incorrect Event Emission: - Emitting the event with the incorrect player list can mislead external systems or observers that rely on the event data for accurate state representation.

Recommended Mitigation: Check for Duplicates Before Adding Players:

- Ensure that the new players are checked for duplicates before adding them to the players array.

Emit Event with Correct Data:

- Emit the event using the updated players array to accurately reflect the current state of the raffle participants.

Here is the revised implementation:

```
function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
        Must send enough to enter raffle");

    // Check for duplicates in newPlayers before adding them to players
    for (uint256 i = 0; i < newPlayers.length - 1; i++) {
        for (uint256 j = i + 1; j < newPlayers.length; j++) {
            require(newPlayers[i] != newPlayers[j], "PuppyRaffle:
                Duplicate player in new entries");
        }
    }

    // Add new players to the players array
    for (uint256 i = 0; i < newPlayers.length; i++) {
        players.push(newPlayers[i]);
    }

    // Emit the event with the updated players array
    emit RaffleEnter(players);
}
```

[H-2] Transaction verification

Description:

The code snippet provided below showcases a potential vulnerability in a smart contract where the completion of a transaction is not verified after invoking the `sendValue` function. Additionally, the comment in the code suggests that there is no check performed to ensure the success of the transaction.

Impact:

The absence of transaction verification can result in funds being lost in the event that the transaction fails. This could lead to financial losses for both the contract and the users involved.

Proof of Concept:

```
payable(msg.sender).sendValue(entranceFee);

players[playerIndex] = address(0);
// @audit it is not checked if the transaction went through
emit RaffleRefunded(playerAddress);
```

Recommended Mitigation:

[H-3] Predictable Randomness

Description: These values are either known or can be influenced by certain parties (like miners), making the resulting randomness predictable and exploitable.

Impact: Predictable randomness can allow attackers to manipulate the outcome, selecting winners unfairly and undermining the integrity of the raffle.

Proof of Concept:

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender,
    block.timestamp, block.difficulty))) % players.length;
```

Recommended Mitigation: Use a more secure randomness source, such as Chainlink VRF (Verifiable Random Function) to ensure unpredictability.

[H-4] Modulo Bias in Random Winner Selection

Description: When using the modulo operator to select a winner from the players array, there is a potential for bias if the range of the hash does not evenly divide the length of the array. This can result in some indices being slightly more likely than others.

Impact: This bias can result in an unfair advantage for certain players, affecting the fairness of the raffle.

Proof of Concept:

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender,
    block.timestamp, block.difficulty))) % players.length;
```

Recommended Mitigation: Implement a bias-resistant selection algorithm or use an off-chain randomness source.

[H-5] Typecasting Overflow

Description: Typecasting may lead to overflow and incorrect fee calculation.

Impact: Overflow can result in incorrect fee allocation, potentially leading to loss of funds.

Proof of Concept:

```
totalFees = totalFees + uint64(fee);
```

Recommended Mitigation: Use safe math operations or ensure proper type handling to prevent overflow.

[H-6] Predictable Randomness for Token Rarity

Description: The randomness used to determine token rarity can be predictable and manipulated by the miner or user.

Impact: Predictable randomness can allow attackers to manipulate the rarity of tokens, undermining the rarity distribution.

Proof of Concept:

```
uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.
    difficulty))) % 100;
```

Recommended Mitigation: Use a more secure randomness source, such as Chainlink VRF, to ensure unpredictability.

[H-7] Active Players Array Not Reset

Description: The active players array is not reset after the winner is selected.

Impact: Failing to reset the players array can lead to incorrect raffle results in subsequent rounds.

Proof of Concept:

```
delete players;
```

Recommended Mitigation: Ensure the players array is properly reset after each raffle to maintain correct state.

[H-8] Zero Address Validation for Fee Address Change

Description: The new fee address is not validated to ensure it is not a zero address.

Impact: Setting feeAddress to a zero address would result in losing all future fees, as they would be sent to a non-recoverable address.

Proof of Concept:

```
feeAddress = newFeeAddress;
```

Recommended Mitigation: Add a validation check to ensure newFeeAddress is not the zero address.

[H-9] Incorrect Handling of Non-Active Players in getActivePlayerIndex Function

Description: The getActivePlayerIndex function is intended to return the index of a player in the players array. If the player is not found, it returns 0. This can lead to a critical issue where it always considers the player at index 0 as the one not found, potentially causing unintended behavior, such as incorrect refunds.

Impact: Incorrect Refunds: - Returning 0 for non-active players can lead to incorrect assumptions that the player at index 0 is inactive, causing potential refunds or actions to be incorrectly directed at the player in that position.

Potential Security Risks: - Mismanagement of funds and incorrect player identification can lead to financial losses and degrade trust in the contract's reliability.

Recommended Mitigation: Return a Distinct Value for Non-Active Players:

- Return a value that clearly indicates the player was not found, such as type(uint256).max.

Handle Non-Active Players Appropriately in Calling Functions:

- Ensure that functions calling getActivePlayerIndex handle the distinct non-active value correctly.

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle stats will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
// @audit DoS Attack
@>          for (uint256 i = 0; i < players.length - 1; i++) {
              for (uint256 j = i + 1; j < players.length; j++) {
                  require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
              }
          }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle:: entrants` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concept If we have 2 sets of 100 players enter, the gas cost will be as such: - 1st 100 players ~6342045 gas - 2nd 100 players ~18068134 gas

This is more then 3x more expensive for the second 100 players.

PoC

Place the following test into `PuppyRaffleTest.t.sol`

```
function testEnterRaffleIsGasInefficient() public {
  vm.startPrank(owner);
  vm.txGasPrice(1);

  /// First we enter 100 participants
  uint256 firstBatch = 100;
  address[] memory firstBatchPlayers = new address[](firstBatch);
  for(uint256 i = 0; i < firstBatchPlayers; i++) {
    firstBatch[i] = address(i);
  }

  uint256 gasStart = gasleft();
  puppyRaffle.enterRaffle{value: entranceFee * firstBatch}(
      firstBatchPlayers);
  uint256 gasEnd = gasleft();
  uint256 gasUsedForFirstBatch = (gasStart - gasEnd) * txPrice;
  console.log("Gas cost of the first 100 partipants is:",
      gasUsedForFirstBatch);

  /// Now we enter 100 more participants
  uint256 secondBatch = 200;
  address[] memory secondBatchPlayers = new address[](secondBatch);
  for(uint256 i = 100; i < secondBatchPlayers; i++) {
    secondBatch[i] = address(i);
  }

  gasStart = gasleft();
  puppyRaffle.enterRaffle{value: entranceFee * secondBatch}(
      secondBatchPlayers);
  gasEnd = gasleft();
  uint256 gasUsedForSecondBatch = (gasStart - gasEnd) * txPrice;
  console.log("Gas cost of the next 100 participant is:",
      gasUsedForSecondBatch);
  vm.stopPrank(owner);

}
```

Recommended Mitigation: Here are some of recommendations, any one of that can be used to mitigate this risk.

1. User a mapping to check duplicates. For this approach you to declare a variable uint256 raffleID, that way each raffle will have unique id. Add a mapping from player address to raffle id to keep of users for particular round.

```

+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+           usersToRaffleId[newPlayers[i]] = true;
        }

        // Check for duplicates
+       for (uint256 i = 0; i < newPlayers.length; i++){
+           require(usersToRaffleId[i] != raffleID, "PuppyRaffle:
    Already a participant");

-        for (uint256 i = 0; i < players.length - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
-            }
        }

        emit RaffleEnter(newPlayers);
    }
.
```

```
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;
    }
```

1. Allow duplicates participants, As technically you can't stop people participants more than once. As players can use new address to enter.

```
function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
        }

        emit RaffleEnter(newPlayers);
    }
```

[L-1] Minimum Pragma Version for Using `openzeppelin` is Now 0.8.21

Description: The current implementation of the smart contract is using an outdated pragma version. With the release of the new version of OpenZeppelin, it is essential to update the minimum pragma version to 0.8.21. The pragma directive specifies the compiler version to be used, and using an outdated version can lead to incompatibility issues and missed improvements or bug fixes.

Impact: Using an outdated pragma version can lead to:

- Incompatibility with the latest versions of OpenZeppelin contracts.
- Missing out on critical security patches, optimizations, and new features introduced in the latest compiler versions.
- Potential vulnerabilities and unexpected behavior due to outdated compiler features and bugs.

Proof of Concept:

Recommended Mitigation: Update the pragma directive in the smart contract to use version 0.8.21 or later. This ensures compatibility with the latest OpenZeppelin contracts and benefits from the latest improvements and security patches.

[G-1] Gas Optimization in Active Player Check

Description: The loop used to check if msg.sender is an active player is not optimized for gas efficiency.

Impact: Inefficient gas usage can lead to higher transaction costs, especially if the players array grows large.

Proof of Concept:

```
for (uint256 i = 0; i < players.length; i++) {
    if (players[i] == msg.sender) {
        return true;
    }
}
```

Recommended Mitigation: Consider using a more gas-efficient data structure or optimization techniques to reduce the gas cost of this operation.

[I-1] Non-Descriptive Function Name

Description: The function name _isActivePlayer is not descriptive enough, which could lead to misunderstandings about its purpose.

Impact: Non-descriptive function names can cause confusion for developers, leading to potential misuse or misunderstanding of the code's functionality.

Proof of Concept:

```
function _isActivePlayer() internal view returns (bool) {...
```

Recommended Mitigation: Rename the function to `_isTheSenderActivePlayer` to clearly convey its purpose.

[I-2] Move Static Parts of JSON to Constants

Description: The static parts of the JSON string in tokenURI function are hardcoded within the function, reducing readability and increasing potential for errors.

Impact: Hardcoding static parts of the JSON makes the function harder to read and maintain, and may introduce errors if not handled properly.

Proof of Concept:

```
return string(
    abi.encodePacked(
        _baseURI(),
        Base64.encode(
            bytes(
                abi.encodePacked(
                    '{"name":"',
                    name(),
                    '", "description":"An adorable puppy!", ',
                    '"attributes": [{"trait_type": "rarity", "value": '
                        ,
                    rareName,
                    '}], "image":"',
                    imageURI,
                    '"}'
                )
            )
        )
    )
);
```

Recommended Mitigation: Define the static JSON string as constant to improve readability and maintainability:

```
string constant JSON_TEMPLATE = '{"name":"{name}", "description":"An
    adorable puppy!", "attributes": [{"trait_type": "rarity", "value":
    "{rareName}"}], "image":"{imageURI}"}';
```