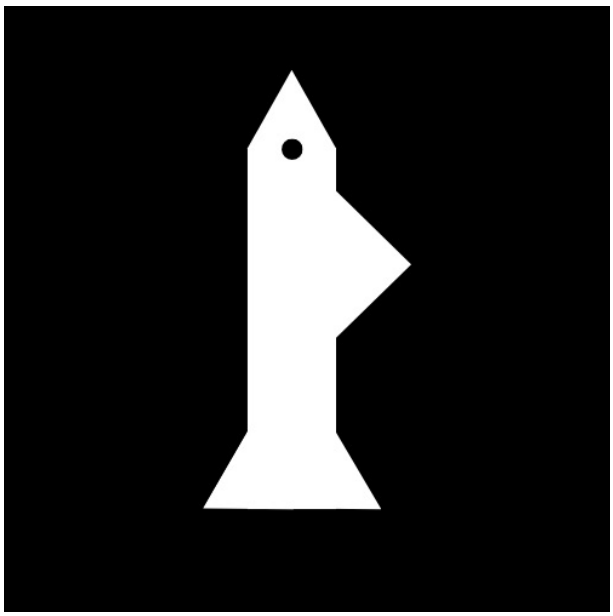

PuppyRaffle Audit Report

Version 1.0

Ryberg.io

August 4, 2024

PuppyRaffle Audit Report

Ryberg.io

August 4, 2024

Prepared by: Ryberg Lead Auditors: - Kirill Rybkov

Table of Contents

    * [M-2] Mishandling ETH in `PuppyRaffle::withdrawFees`, owner will not be able to withdraw collected fees.
    * [M-3] Checking for duplicate players loop in `PuppyRaffle::enterRaffle` causes DoS attack, incrementing gas costs for future entrants
  - Informational
    * [I-1] Function `PuppyRaffle::_isActivePlayer` is never used, increasing gas costs.
    * [I-2] Potentially erroneous active player index
    * [I-3] Zero address may be erroneously considered an active player
    * [I-4] Unchanged variables should be constant or immutable
    * [I-5] Floating pragmas
    * [I-6] Zero address validation
    * [I-7] Test Coverage

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Ryberg 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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope: ## Scope

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

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function. # Executive Summary I loved auditing this codebase! Patrick is so great at writing intentionally bad code! ## Issues found | Severity | Number of issues found | | ——– | ——————- | | High | 2 | | Medium | 3 | | Low | 0 | | Info | 7 | | Total | 12 | # Findings ## High ### [H-1] Reentrancy at `PuppyRaffle::refund`, attackers can withdraw all the funds from the treasury.

Description The `PuppyRaffle::refund` function does not follow CEI/FREI-PI and as a result, enables participants to drain the contract balance.

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

```
1       function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
4           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
5
6   @>      payable(msg.sender).sendValue(entranceFee);
7
8   @>      players[playerIndex] = address(0);
9           emit RaffleRefunded(playerAddress);
10      }
```

Impact Malicious actors will be able to withdraw all of the funds from the contract treasury.

Proof of Concepts 1. Initial Call: The attacker calls the `PuppyRaffle::refund` function. 2. External Call Execution: Inside the refund function, there is an external call to the attacker's contract (sendValue). 3. Fallback Function Invocation: The attacker's contract contains a fallback function that calls `PuppyRaffle::refund` again before the state `players[]` is changed. 4. Reentrancy Loop: The refund function is called recursively, allowing the attacker to drain funds or execute unintended operations. The following test was written and placed inside `PuppyRaffleTest.t.sol`.

Code

```
1       function test_reentrancyRefund() public {
2           address[] memory players = new address[](4);
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9           ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUser = makeAddr("attackUser");
11          vm.deal(attackUser, 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          // attack
17          vm.prank(attackUser);
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("starting attacker contract balance: ",
                startingAttackContractBalance);
21          console.log("starting contract balance: ",
                startingContractBalance);
```

```
22
23          console.log("ending attacker contract balance: ", address(
                attackerContract).balance);
24          console.log("ending contract balance: ", address(puppyRaffle).
                balance);
25      }
```

Recommended mitigation Change state (`players[]`) before an external call happens.

```
1       function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
4           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
5
6   +       players[playerIndex] = address(0);
7   +       emit RaffleRefunded(playerAddress);
8
9           payable(msg.sender).sendValue(entranceFee);
10
11  -       players[playerIndex] = address(0);
12  -       emit RaffleRefunded(playerAddress);
13      }
```

[H-2] Using `keccak256` hashing function in `PuppyRaffle::selectWinner`doesn't generate truly random number, can adjust the winner index.

Description The function `keccal256` doesn't generate a truly random number since it can be influenced by the `msg.sender` address, `block.timestamp` and `block.difficulty`.

Impact By manipulating either of these variables, attacker may produce a more favourable outcome for themselves. For example, the could change these variables to let `winnerIndeex` be themselves or, change the `rarity`.

Recommended mitigation Recommended to use Chainlink VRF (Verifiable Random Function) to achieve true randomness. ## Medium ### [M-1] Overflow attack in `PuppyRaffle:selectWinner`, owner will not be able to withdraw collected fees **Description** At

```
1   totalFees = totalFees + uint64(fee);
```

`totalFees` is of type uint64, while variable `fee` is of type uint256 thus, causing an overflow attack which can happen when the fees generated are greater than ~18 ether.

Impact Owner will not be able to withdraw their generated fees due to a check in `PuppyRaffle::withdrawFees`:

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

Proof of Concepts When entering 89 players, the fees generated are less than when entering 4 players which should not have happened.

Code

```
    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 Declare `totalFees` as of type uint256.

[M-2] Mishandling ETH in `PuppyRaffle::withdrawFees`, owner will not be able to withdraw collected fees.

Description

```
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          // slither-disable-next-line arbitrary-send-eth
6          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
7          require(success, "PuppyRaffle: Failed to withdraw fees");
8      }
```

1. A malicious actor has pushed some ETH into the `PuppyRaffle` contract through self-destruct of another smart contract.
2. Due to this check, since the on-chain balance of the contract will be different to the `totalFees` variable, the transaction will be reverting with "PuppyRaffle: There are currently players active!"

Impact Owner will not be able to withdraw fees generated by the raffle.

Proof of Concepts 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 line of code that checks for that condition.

```
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          // slither-disable-next-line arbitrary-send-eth
6          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
7          require(success, "PuppyRaffle: Failed to withdraw fees");
8      }
```

[M-3] Checking for duplicate players loop in `PuppyRaffle::enterRaffle` causes DoS attack, incrementing gas costs for future entrants

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

```
1           // @audit DoS
2   @>      for (uint256 i = 0; i < players.length - 1; i++) {
3               for (uint256 j = i + 1; j < players.length; j++) {
4                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
5               }
6           }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

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

Proof of Code:

if we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players ~6252128 gas - 2nd 100 players ~18068218 gas

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

PoC

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

```
1       function testEnterRaffleDoS() public {
2           uint256 numPlaters = 100;
3
4           address[] memory players = new address[](numPlaters);
5           for(uint256 i = 0; i < numPlaters; i++){
6               players[i] = address(i);
7           }
8           uint256 gasStart = gasleft();
9           puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
10          uint256 gasFinish = gasleft();
11          uint256 gasUsed = (gasStart - gasFinish);
12          console.log("Gas used 100 players: ",gasUsed);
13
14          address[] memory players2 = new address[](numPlaters);
15          for(uint256 i = 0; i < numPlaters; i++){
16              players2[i] = address(i + numPlaters);
17          }
18          uint256 gasStart2 = gasleft();
19          puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
                players2);
20          uint256 gasFinish2 = gasleft();
21          uint256 gasUsed2 = (gasStart2 - gasFinish2);
22          console.log("Gas used 100 players: ",gasUsed2);
```

```
23
24          assert(gasUsed < gasUsed2);
25      }
```

Recommended Mitigation: There are a few recommendations.

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

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
14 +       // Check for duplicates only from the new players
15 +       for (uint256 i = 0; i < newPlayers.length; i++) {
16 +          require(addressToRaffleId[newPlayers[i]] != raffleId, "
   PuppyRaffle: Duplicate player");
17 +       }
18 -       for (uint256 i = 0; i < players.length; i++) {
19 -           for (uint256 j = i + 1; j < players.length; j++) {
20 -               require(players[i] != players[j], "PuppyRaffle:
   Duplicate player");
21 -           }
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. ## Informational ### [I-1] Function `PuppyRaffle::_isActivePlayer` is never used, increasing gas costs.

Description Function `PuppyRaffle::_isActivePlayer` is never used.

Impact Increased gas costs at deployment.

Recommended mitigation Remove `PuppyRaffle::_isActivePlayer`.

[I-2] 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-3] 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.

[I-4] 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-5] 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-6] 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-7] 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.