PuppyRaffle Audit Report

Prepared by: Stunspotx

Table of Contents

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 Stunspotx 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
Likelihood	High	H	H/M	M
	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: `22bbbb2c47f3f2b78c1b134590baf41383fd354f`

Scope

- In 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	4

Findings

High

[H-1] `PuppyRaffle::refund` Violates CEI Pattern Leading to Reentrancy and Fund Drain

Description:
The `PuppyRaffle::refund` function sends ETH to the caller before updating the contract's internal state.

This violates the Checks-Effects-Interactions (CEI) pattern and leaves the function vulnerable to reentrancy. Because the player is not removed or marked as refunded prior to the external call, a malicious contract can reenter the `PuppyRaffle::refund` function while it is still considered an active participant.

Impact:
An attacker can repeatedly call `PuppyRaffle::refund` for the same raffle entry, allowing them to:

- Drain ETH belonging to other raffle participants
- Empty the contract balance
- Compromise the integrity and fairness of the raffle

This results in a critical loss of user funds.

Proof of Concept:
A malicious contract can exploit this issue by reentering the `PuppyRaffle::refund` function during the ETH transfer:

```
    receive() external payable {
        address[] memory players = raffle.players();
        uint256 playerIndex = raffle.getActivePlayerIndex(address(this));

        for (uint256 i = 0; i < players.length; i++) {
            raffle.refund(playerIndex);
        }
    }
```

Because state updates occur after the transfer, each reentrant call succeeds.

Recommended Mitigation:

- Enforce the **Checks-Effects-Interactions** pattern by updating all internal state (e.g., removing or marking the player as refunded) before transferring ETH.
- Add a **reentrancy guard** (e.g., `nonReentrant`) to the `PuppyRaffle::refund` function.
- Consider implementing a **pull-based withdrawal** pattern to isolate external calls from state changes.

[H-2] `PuppyRaffle::selectWinner` Uses Weak On-Chain Randomness Allowing Winner Manipulation and Guaranteed Profit

Description:
The `PuppyRaffle::selectWinner::winnerIndex` is derived from predictable on-chain values such as `msg.sender`, `block.timestamp`, and `block.difficulty`. These values are either known in advance, controllable by the caller, or observable by anyone monitoring the mempool.

Because `PuppyRaffle::selectWinner` is permissionless, an attacker can locally compute the exact value of `PuppyRaffle::selectWinner::winnerIndex` before deciding whether to call the function. This allows the attacker to deterministically decide when they will win the raffle.

Additionally, the absence of checks preventing calls to `PuppyRaffle::refund` after the raffle has ended allows the attacker to repeatedly enter and exit the raffle at no financial risk.

Impact:
An attacker can:

- Predict the outcome of `PuppyRaffle::selectWinner` with certainty
- Only finalize the raffle when they are guaranteed to win
- Refund their entry whenever they are predicted to lose
- Drain the entire raffle pot while only paying gas fees

This results in **complete loss of fairness**, **guaranteed attacker profit**, and **loss of funds for honest participants**.

Proof of Concept:

1. The attacker enters the raffle using `PuppyRaffle::enterRaffle::newPlayers`:

```
address[] memory attacker = new address[](1);
attacker[0] = attackerAddress;

puppyRaffle.enterRaffle{value: 1 ether}(attacker);
```

2. The attacker determines their index in the `PuppyRaffle::players` array:

```
// if puppyRaffle.players().length == 6
// attackerIndex = 6 - 1 = 5
// players[5] = attackerAddress
uint256 attackerIndex = puppyRaffle.players().length - 1;
```

3. The attacker waits until `block.timestamp >= raffleStartTime + raffleDuration`.

4. The attacker monitors the mempool for transactions calling `PuppyRaffle::selectWinner`.

5. The attacker locally computes `PuppyRaffle::selectWinner::winnerIndex` using the same logic as the contract:

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

6. If `PuppyRaffle::selectWinner::winnerIndex != attackerIndex`, the attacker calls `PuppyRaffle::refund` to recover their funds and repeats the process.

7. If `PuppyRaffle::selectWinner::winnerIndex == attackerIndex`, the attacker calls `PuppyRaffle::selectWinner` and wins the entire pot.

Recommended Mitigation:

- Remove reliance on predictable on-chain values for randomness.
- Integrate a verifiable randomness source such as **Chainlink VRF**.
- Restrict who can call `PuppyRaffle::selectWinner` or decouple winner selection from transaction senders.
- Prevent calls to `PuppyRaffle::refund` after the raffle has ended.

These changes will restore fairness and prevent deterministic winner manipulation.

[H-3] Unbounded Array Length That Can Result To a DOS Attack.

Description: `PuppyRaffle::enterRaffle::newPlayers` is an array of addresses that can enter the raffle when the `PuppyRaffle::enterRaffle` function is called. The array has no bound or limit which can lead to a severe distortion of the functionality of the protocol.

Impact: A single malicious player can cause a DOS attack to the protocol by entering the raffle countless times with a `PuppyRaffle::enterRaffle::newPlayers` length of 0 and can do so without ETH. Every other player will forcefully be prevented from entering the raffle.

Proof of Concept: The proof of code below shows how a malicious player can exploit and cause severe damage to the `PuppyRaffle::enterRaffle` functionality:

▶ Details
code

```
    function testPlayerCanEnterRaffleWithoutEth() public {
        address[] memory players = new address[](0);
        // players.length will ofcourse return 0
        console.log("players length:", players.length);

        for (uint256 i = 0; i < 1000000; i++) {
            vm.prank(playerOne);
            // entranceFee * 0 = 0
            puppyRaffle.enterRaffle{value: 0}(players);
        }
    }
```

Recommended Mitigation: This attack can be mitigated in atleast 2 ways:

1. Set a limit for the `PuppyRaffle::enterRaffle::newPlayers` array:

   ```
   function enterRaffle(address[1] memory newPlayers) public payable {}
   ```

   This will make sure the `PuppyRaffle::enterRaffle::newPlayers` array length is not 0 and the correct amount of eth is paid for every entry.

2. Check that `PuppyRaffle::enterRaffle::newPlayers` length is not 0. Add this check below to the `PuppyRaffle::enterRaffle` function:

```
+       if (newPlayers.length == 0) {
+           revert("newPlayers length can't be 0");
+       }
```

[H-4] Missing Raffle-End Check in `PuppyRaffle::refund` Allows Refunds After Raffle Completion

Description:
The `PuppyRaffle::refund` function lacks a validation mechanism to ensure that the raffle is still active before processing refunds. There is no check verifying that `block.timestamp` is less than `raffleStartTime + raffleDuration`. As a result, players are able to successfully request refunds even after the raffle has ended.

This violates the expected raffle lifecycle, where once the raffle duration has elapsed, participant funds should be locked in for winner selection.

Impact:
An attacker or participant can:

- Enter the raffle and wait until it has ended
- Call `PuppyRaffle::refund` after the raffle duration
- Recover their full entry fee despite the raffle being finalized

This enables **risk-free participation**, undermines raffle fairness, and can lead to **loss of funds for honest participants**, especially when combined with other vulnerabilities such as weak randomness.

Proof of Concept:
The following test scenario demonstrates that `PuppyRaffle::refund` can be called after the raffle has ended:

```
address[] memory attacker = new address[](1);
attacker[0] = attackerAddress;

vm.startPrank(attackerAddress);
puppyRaffle.enterRaffle{value: 1 ether}(attacker);

vm.warp(block.timestamp + duration + 1);
vm.roll(block.number + 1);

uint256 attackerIndex = puppyRaffle.getActivePlayerIndex(attackerAddress);
puppyRaffle.refund(attackerIndex);

assertEq(attackerAddress.balance, 1 ether);
```

The attacker successfully retrieves their entry fee even though the raffle duration has elapsed.

Recommended Mitigation:
Add a raffle-end validation at the beginning of `PuppyRaffle::refund` to prevent refunds after the raffle

has concluded:

```
+    if (block.timestamp > raffleStartTime + raffleDuration) {
+       revert("No refunds after raffle ends");
+    }
```

This ensures refunds are only processed while the raffle is active and preserves the intended raffle lifecycle.