title: Puppy Raffle Protocol Audit Report author: Mauro Júnior date: February 8th 2024 header-includes:

- \usepackage
- \usepackage

\begin \centering \begin[h] \centering \includegraphics[width=0.5\textwidth] \end \vspace*{2cm} {\Huge\bfseries Protocol Audit Report\par} \vspace{1cm} {\Large Version 1.0\par} \vspace{2cm} {\Large\itshape Cyfrin.io\par} \vfill {\large \today\par} \end

\maketitle

Prepared by: [Mauro Júnior] Lead Auditors: Mauro Júnior

Table of Contents

Protocol Summary

Puppy Raffle

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.

Getting Started

Requirements

- git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)
  - You'll know you did it right if you can run `git --version` and you see a response like `git version x.x.x`
- foundry (https://getfoundry.sh/)
  - You'll know you did it right if you can run `forge --version` and you see a response like `forge 0.2.0 (816e00b 2023-03-16T00:05:26.396218Z)`

Quickstart

```
git clone https://github.com/Cyfrin/4-puppy-raffle-audit
cd 4-puppy-raffle-audit
make
```

Optional Gitpod

If you can't or don't want to run and install locally, you can work with this repo in Gitpod. If you do this, you can skip the `clone this repo` part.

Open in Gitpod (https://gitpod.io/#github.com/Cyfrin/4-puppy-raffle-audit)

Usage

Testing

```
forge test
```

Test Coverage

```
forge coverage
```

and for coverage based testing:

```
forge coverage --report debug
```

Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

Known Issues

None

Disclaimer

The Mauro Júnior 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 (https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity) severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

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

This is an first flight audit from CodeHawks, really good to learn security auditing.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	9
Total	15

Findings

Highs:

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

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

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after it check for some update in state.

The function which has vulnerability:

```
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 enter the raffle can have a `receive` or `fallback` function in which, by calling `PuppyRaffle::refund` again and claim another refund, they could just repeat this until all money funded in the contract balance is drained

Impact All fees paid by players that entered the raffle could be stolen from a malicious participant!

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`

3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the PuppyRaffle balance.

▶ PoC Code

You can just do

```
forge test --mt testCanGetRefundReentrancy
```

To run this test!

Recomended Mitigations To prevent all this, we should have the `PuppyRaffle:refund` function be updated before making the external call, also adding CEI(Checks, Effects, Interactions) would prevent it. Additionally, we also should move the event emission because it can be manipulated!

```
    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(entranceFees);
-   players[playerIndex] = address(0);
-   emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in ``PuppyRaffle:selectWinner` allows users and/or miners to influence or predict a winner

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

This additionally means users could front-run this function and call `refund` if they see they are not the winner. And of course, users or miners could be influenced to hack the protocol, benefits themselves.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war to choose whoever wins the raffle.

Proof of Concept:

1. Validators and miners can know the values of `block.timestamp` and `block.difficulty` ahead of time and use that to predict when/how to participate. You can see the blog solidity blog on prevrandao (https://soliditydeveloper.com/prevrandao). there, `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 (https://betterprogramming.pub/how-to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space, resulting in many protocols getting hacked from the past couple of years.

Recommended Mitigation: Consider using a cryptographically provable random number generator like the Chainlink VRF (https://docs.chain.link/vrf) to have more randomness into your selectWinner function.

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

Description: In solidity versions prior to `0.8.0` like the one used in `PuppyRaffle.sol` integers were subject to integer overflows and would not revert if they are unchecked.

~~~
```js
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = 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

▶ Here is the 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 Mitigations You can choose to change the version of solidity to something like 0.8.28 so that when an int type reaches it's peak it will automatically revert.

Or you can just make the int bigger 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;
```

Mediums

[M-1] Looping through players array to check for duplicates in `PuppyRaffle:enterRaffle` is a potential Denial Of Service attack, incrementing gas cost for future entrants

Description The `PuppyRaffle:enterRaffle` function loops through an list of addresses in the players array to check out for duplicates.However, as more people enter the Raffle it will become expensive to enter because it will be looping through a bigger list of addresses, since the more checks an new player will have to make to see if they already entered. This means the ones who enter the Raffle right when it starts the gas will be lower than those who enter after some time. Every additional address in `players` array is an additional check the loop will have to make.

Here is the code that has vulnerability:

```
// @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 The gas cost for raffle entrants will greatly increase as more people enter the raffle, this will lead to an general discouragement to those who enter later on, meaning it would cause an rush to be the first to enter. An attacker might make the `PuppyRaffle: entrants` array so big, that no one elses will enter the raffle and they(attackers) will likely win.

Proof of Concept If we have two sets of 100 players entering the raffle, the gas cost of one will be greater than another. Ex: 1st 100 players: ~6000000 gas Ex: 2nd 100 players: ~20000000 gas This will be 3x more expensive for the later players to enter like the 199th enter.

▶ PoC

[M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new raffle

Description: The `PuppyRaffle::selectWinner` function is the one that is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment (doesn't have `receive` neither `fallback` functions to receive ETH), 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, since it's doing for loop in `PuppyRaffle:enterRaffle` to check for duplicates, creating a DoS attack.

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, the ones that really are the winners would not be able to get paid out, and then someone else could just reenter and be the winner and take all their money because they do have a fallback or receive function.

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: Options:

1. Do not allow smart contract wallet entrants (not recommended) in the raffle.
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)

Lows

[L-1] `PuppyRaffle:getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing the player at index 0 think they are not in the array

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 zero if the player is NOT in the array.

~~~
```javascript
// @return the index of player in the array, if the player is not active, it will return 0
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 think they have not entered the raffle, and attempt to enter the raffle again, wasting some gas

Proof of Concept

1. User enters the raffle, they are the first entrant at index 0
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation
4. User may try to enter again wasting gas

Recomended Mitigation One option is to revert if the player is not in the array instead of just returning 0.

Another option is to reserve the 0th position for any competition.

The one i would use is to just return an `int256` where the function returns -1 if the player is not active.

Informational

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

Details Solidity version in `PuppyRaffle.sol` should be marked as specific in your contracts instead of a wide version. For example, instead of the flotating `pragma solidity ^0.8.28;`, consider replacing this for a more specific version like this `pragma solidity 0.8.28;`

Found in `PuppyRaffle.sol` file.

[I-2] Using outdated version of solidity is not recommended.

Details Consider using a newer version of solidity like `0.8.28` which is the latest version. The recommendation take into account - Risks related to recent releases - Risks of complex code generation changes - Risks of new language features - Risks of known bugs

[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 (src/PuppyRaffle.sol#L69)

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 159 (src/PuppyRaffle.sol#L159)

  ```
          previousWinner = winner;
  ```

- Found in src/PuppyRaffle.sol Line: 182 (src/PuppyRaffle.sol#L182)

  ```
          feeAddress = newFeeAddress;
  ```

[I-4] `PuppyRaffle:selectWinner` doesn't follow CEI, which is not a best practice

It's best to keep code more readable and clean so we recommend to follow CEI (Checks, Effects, Interactions). This would be the code following the CEI standard.

~~~
```diff
~~~

- (bool success,) = winner.call("");
- require(success, "PuppyRaffle: Failed to send prize pool to winner"); _safeMint(winner, tokenId);

- (bool success,) = winner.call("");
- require(success, "PuppyRaffle: Failed to send prize pool to winner");

```

```

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

It can be quite confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name. Magic numbers are numbers just floating around, so we can just add a constant variable for them.

Examples:

```
    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 one of the best practices 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 not used once in the codebase and therefore should be removed(DEAD CODE)

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;
-    }
```
~~~

Consider removing all these lines.

Gas

[G-1] Unchanged statle variables should be declare as constant or immutable for more gas optimization

Details There are severly state variables that doesn't change and it could lead to gas inefficiency due to the fact it's being read from storage. So, for gas efficiency these variables should be either constant or immutable.

Here are the variables that don't change:

- `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

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+ uint256 playersLength = players.length;
- for (uint256 i = 0; i < players.length - 1; i++) {
+ for (uint256 i = 0; i < playersLength - 1; i++) {
-    for (uint256 j = i + 1; j < players.length; j++) {
+    for (uint256 j = i + 1; j < playersLength; j++) {
      require(players[i] != players[j], "PuppyRaffle: Duplicate player");
}
}
```

So, consider adding a new variable that represents `the players.lenght` as showed above