Dussehra Security Audit Report by MK (16th DEC 2024)

High Risk Findings

[H-1] `ChoosingRam::isRamSelected` Is Never Set to `true` When Calling `ChoosingRam::increaseValuesOfParticipants` Resulting in Unintended Behavior and a Temporary DOS

Description: During `increaseValuesOfParticipants`, only `selectedRam` is set to the winner's address and `isRamSelected` is not set to `true`. This allows `selectedRam` to be overwritten, and creates a temporary DOS to `Dussehra::killRavana` and `Dussehra::withdraw`.

Impact: Due to `isRamSelected` not being set to `true`, subsequent calls to `increaseValuesOfParticipants` are allowed (since the `RamIsNotSelected` modifier passes), overwriting `selectedRam`. This is unintended behavior.

Furthermore, a DOS occurs in `Dussehra::killRavana` and `Dussehra::withdraw` since the `RamIsSelected` modifier is unable to pass. This DOS persists until the `organiser` is able to call `ChoosingRam::selectRamIfNotSelected` which does set `isRamSelected` to true. However, this also overwrites `selectedRam`.

Proof of Concept: When all characteristics are set, only `selectedRam` is set to the winner's address and `isRamSelected` is not set to `true`.

Recommended Mitigation: When setting `selectedRam`, also set `isRamSelected` to `true` during `increaseValuesOfParticipants` calls.

```
    function increaseValuesOfParticipants(uint256 tokenIdOfChallenger, uint256
tokenIdOfAnyPerticipent)
        public
        RamIsNotSelected
    {
        // ...
        if (random == 0) {
            // ...
            } else if
(ramNFT.getCharacteristics(tokenIdOfChallenger).isSatyavaakyah == false){
                ramNFT.updateCharacteristics(tokenIdOfChallenger, true, true,
true, true, true);
                selectedRam = ramNFT.getCharacteristics(tokenIdOfChallenger).ram;
+               isRamSelected = true;
            }
        } else {
            // ...
            } else if
(ramNFT.getCharacteristics(tokenIdOfAnyPerticipent).isSatyavaakyah == false){
                ramNFT.updateCharacteristics(tokenIdOfAnyPerticipent, true, true,
true, true, true);
```

```
                selectedRam =
ramNFT.getCharacteristics(tokenIdOfAnyPerticipent).ram;
+               isRamSelected = true;
            }
        }
    }
```

[H-2] Public `RamNFT::mintRamNFT` Function Allows for Direct Minting, Bypassing the `entranceFee` Check in `Dussehra::enterPeopleWhoLikeRam`

Description: Missing access controls for `mintRamNFT` allows anyone to mint NFTs for free, without paying the `entranceFee`.

Impact: NFTs can be minted for free, without paying the `entranceFee`.

Proof of Concept:

1. Improper access control for the `mintRamNFT` function allows anyone to mint NFTs.

2. Add the following test case to `Dussehra.t.sol`:

3. Test Case

```
    function test_publicRamNFTMint() public {
        uint256 NUM_NFTS_TO_MINT = 100;

        // assert that `player1` has no ether, so NFTs are minted for free
        assertEq(address(player1).balance, 0 ether);

        // mint NFTs
        vm.startPrank(player1);
        for (uint256 i; i < NUM_NFTS_TO_MINT; i++) {
            ramNFT.mintRamNFT(address(player1));
        }
        vm.stopPrank();

        // assert that the NFT balance of `player1` is the same
        assertEq(ramNFT.balanceOf(address(player1)), NUM_NFTS_TO_MINT);
    }
```

4. Then, run the test:

```
forge test --mt test_publicRamNFTMint -vvvvv
```

Recommended Mitigation:

1. Add the `onlyChoosingRamContract` modifier to `mintRamNFT` to ensure only the `ChoosingRam` contract can mint Ram NFTs.

```
-   function mintRamNFT(address to) public {
+   function mintRamNFT(address to) public onlyChoosingRamContract {
        uint256 newTokenId = tokenCounter++;
        _safeMint(to, newTokenId);

        Characteristics[newTokenId] = CharacteristicsOfRam({
            ram: to,
            isJitaKrodhah: false,
            isDhyutimaan: false,
            isVidvaan: false,
            isAatmavan: false,
            isSatyavaakyah: false
        });
    }
```

[H-3] The user can predict the outcome of the `ChoosingRam::increaseValuesOfParticipants` to become Ram and get the reward

Description: The function `ChoosingRam::increaseValuesOfParticipants` depends on a random value to select the participant to whom to increase the characteristics. This function generates the random number by using `block.timestamp`, `block.prevrandao` and the `msg.sender` values. Those values are considered a bad source of randomness. The users can predict the outcome and execute the function only if they will be the winners of the challenge. This will help them to become Ram.

Impact: The bad source of randomness gives a malicious user the opportunity to become Ram and to get the reward.

Proof of Concept: Using `block.timestamp` as a source of randomness is commonly advised against, as the outcome can be manipulated by calling contracts. Also, for some chains like zkSync `block.prevrandao` is a constant value. This will allow the users to predict the result of the calculated number in Line 52 of `ChoosingRam.sol`: `uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao, msg.sender))) % 2`. This will give them the chance to execute a challenge only if they are the winners.

The following code demonstrates how an attack can be executed.

```
function test_increaseValuesOfParticipantsIsNotRandom() public {
    Dussehra dussehra;
    RamNFT ramNFT;
    ChoosingRam choosingRam;
    address organiser = makeAddr("organiser");
    address player1 = makeAddr("player1");
    address player2 = makeAddr("player2");

    vm.startPrank(organiser);
```

```
        ramNFT = new RamNFT();
        choosingRam = new ChoosingRam(address(ramNFT));
        dussehra = new Dussehra(1 ether, address(choosingRam), address(ramNFT));
        ramNFT.setChoosingRamContract(address(choosingRam));
        vm.stopPrank();

        vm.startPrank(player1);
        vm.deal(player1, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        vm.stopPrank();

        // the second player will predict the outcomes and
        // will become the Ram
        vm.startPrank(player2);
        vm.deal(player2, 1 ether);
        dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
        uint256 winnings = 0;
        uint256 time = 1;
        // this loop will be executed until the second player
        // wins 5 times
        while (winnings < 5) {
            vm.warp(++time);
            if (
                uint256(
                    keccak256(
                        abi.encodePacked(
                            block.timestamp,
                            block.prevrandao,
                            player2
                        )
                    )
                ) %
                    2 ==
                0
            ) {
                // the following block will be executed only if the user
                // is gonna win the challenge
                ++winnings;
                choosingRam.increaseValuesOfParticipants(1, 0);
            }
        }
        vm.stopPrank();

        // as we can see the second player is now the Ram
        assertEq(choosingRam.selectedRam(), player2);
    }
```

Recommended Mitigation: Consider using a decentralized oracle for the generation of random numbers, such as Chainlinks VRF. The Chainlink VRF gives two methods to request randomness: subscription and direct funding method. They will have their added cost, but will solve the randomness issues of the `Dussehra` contract.

[H-4] Wrong check in `ChoosingRam::increaseValuesOfParticipants` allows users to become Ram by challenging an unexisted participant

Description: Wrong checks on Line 37 and Line 40 of the `ChoosingRam` contract allows the users to challenge an unexisted participant. This increases their chance to become Ram and get the reward.

Impact: The users might get the information from the `RamNFT::tokenCounter` and always challenge an unexisted token which will increase their chance to become Ram and get the reward.

Proof of Concept:

1. The `ChoosingRam::increaseValuesOfParticipants` function is expected to revert when an unexisted token id is challenged. For this purpose the `RamNFT` contract counts the number of minted NFT tokens and provides the function `RamNFT::tokenCounter`. The returned value of this function is not used correctly in the equation in Line 37 of the `ChoosingRam`. The equation must be `tokenIdOfChallenger >= ramNFT.tokenCounter()` instead of `tokenIdOfChallenger > ramNFT.tokenCounter()`. Also for Line 40 the equation must be `tokenIdOfAnyPerticipent >= ramNFT.tokenCounter()` and not `tokenIdOfAnyPerticipent > ramNFT.tokenCounter()`.

2. The following test shows that the `ChoosingRam::increaseValuesOfParticipants` is not reverting as expected.

```
function test_usersCanChallengeUnexistedToken() public {
    Dussehra dussehra;
    RamNFT ramNFT;
    ChoosingRam choosingRam;
    address organiser = makeAddr("organiser");
    address player1 = makeAddr("player1");

    vm.startPrank(organiser);
    ramNFT = new RamNFT();
    choosingRam = new ChoosingRam(address(ramNFT));
    dussehra = new Dussehra(1 ether, address(choosingRam), address(ramNFT));
    ramNFT.setChoosingRamContract(address(choosingRam));
    vm.stopPrank();

    vm.startPrank(player1);
    vm.deal(player1, 1 ether);
    dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
    vm.warp(2);
    // there is only one participant with only one RamNFT token
    // but the increaseValuesOfParticipants function will not revert
    // if token ids 0 and 1 are used
    choosingRam.increaseValuesOfParticipants(0, 1);
    choosingRam.increaseValuesOfParticipants(0, 1);
    choosingRam.increaseValuesOfParticipants(0, 1);
    choosingRam.increaseValuesOfParticipants(0, 1);
    choosingRam.increaseValuesOfParticipants(0, 1);
    vm.stopPrank();

    // and finally the only player will be Ram
```

```
        // without winning any challenge with another player
        // actually, even there is no another player who had
        // entered the contest
        assertEq(choosingRam.selectedRam(), player1);
    }
```

Recommended Mitigation: Fix the sign of the equations in Line 37 and Line 40 of the `ChoosingRam`. See the code below.

```
function increaseValuesOfParticipants(
    uint256 tokenIdOfChallenger,
    uint256 tokenIdOfAnyPerticipent
) public RamIsNotSelected {
-   if (tokenIdOfChallenger > ramNFT.tokenCounter()) {
+   if (tokenIdOfChallenger >= ramNFT.tokenCounter()) {
        revert ChoosingRam__InvalidTokenIdOfChallenger();
    }
-   if (tokenIdOfAnyPerticipent > ramNFT.tokenCounter()) {
+   if (tokenIdOfAnyPerticipent >= ramNFT.tokenCounter()) {
        revert ChoosingRam__InvalidTokenIdOfPerticipent();
    }
}
```

Medium Risk Findings

[M-1] `Dussehra::killRavana` function can be called twice in row which leads to rewards being denied for selected Ram.

Description: The `Dussehra::killRavana` function can be exploited by calling it multiple times within the valid time window, leading to the unintended transfer of funds to the organiser multiple times. This depletes the contract's balance, potentially leaving insufficient funds for the selected Ram to withdraw their rightful share.

Impact: If `Dussehra::killRavana` function is called twice in row by any random caller, selected Ram will not be able to withdraw his reward. In that case all entrance fees will go to organiser.

Proof of Concept:

1. `Dussehra::killRavana` function allows to kill Ravana and transfers 50% of all entrance fees to organiser. Problem is that `Dussehra::killRavana` function can be called twice in row, and second call will again send 50% of all entrance fees to organiser, which means contract will be empty (or almost empty). Then if selected Ram wants to call `Dussehra::withdraw` function, it will not be able to withdraw because there is no enough funds in `Dussehra` contract for selected Ram.
    1. Two players mint their Ram NFTs.
    2. After some time organiser calls `ChoosingRam::selectRamIfNotSelected` function and selects one of player as selected Ram.

3. Random caller calls `Dussehra::killRavana` function.
4. Random caller again calls `Dussehra::killRavana` function.
5. Assert that `Dussehra` contract is empty.
6. Selected Ram calls `Dussehra::withdraw` function, and it reverts.

```
    // @audit - can be called twice in row
@>  function killRavana() public RamIsSelected {
        // Oct 11 2024 23:57:49 (GMT)
        if (block.timestamp < 1728691069) {
            revert Dussehra__MahuratIsNotStart();
        }
        // Oct 13 2024 00:01:09 (GMT)
        if (block.timestamp > 1728777669) {
            revert Dussehra__MahuratIsFinished();
        }
        IsRavanKilled = true;
        uint256 totalAmountByThePeople = WantToBeLikeRam.length * entranceFee;
        totalAmountGivenToRam = (totalAmountByThePeople * 50) / 100;
        (bool success, ) = organiser.call{value: totalAmountGivenToRam}("");
        require(success, "Failed to send money to organiser");
    }
```

2. Place the following test into `Dussehra.t.sol`.

```
    function test_killRavanaCanBeCalledTwoTimesDenyingRewardForRam() public
participants {
        vm.warp(1728691200 + 1);

        vm.prank(organiser);
        choosingRam.selectRamIfNotSelected();

        vm.startPrank(makeAddr("randomCaller"));
        dussehra.killRavana();
        dussehra.killRavana();
        vm.stopPrank();

        assertTrue(address(dussehra).balance == 0);

        vm.prank(choosingRam.selectedRam());
        vm.expectRevert("Failed to send money to Ram");
        dussehra.withdraw();
    }
```

Recommended Mitigation:

1. Add modifier `RavanNotKilled` to `Dussehra::killRavana` function to prevent that function can be called twice.

```
+   modifier RavanNotKilled() {
+       require(!IsRavanKilled, "Ravan is already killed");
+       _;
+   }

    .
    .

-   function killRavana() public RamIsSelected {
+   function killRavana() public RamIsSelected RavanNotKilled {
        // Oct 11 2024 23:57:49 (GMT)
        if (block.timestamp < 1728691069) {
            revert Dussehra__MahuratIsNotStart();
        }
        // Oct 13 2024 00:01:09 (GMT)
        if (block.timestamp > 1728777669) {
            revert Dussehra__MahuratIsFinished();
        }
        IsRavanKilled = true;
        uint256 totalAmountByThePeople = WantToBeLikeRam.length * entranceFee;
        totalAmountGivenToRam = (totalAmountByThePeople * 50) / 100;
        (bool success, ) = organiser.call{value: totalAmountGivenToRam}("");
        require(success, "Failed to send money to organiser");
    }
```

[M-2] Lack of check in `ChoosingRam::increaseValuesOfParticipants` function allows that player can play against himself.

Description: Lack of check in `ChoosingRam::increaseValuesOfParticipants` function allows a player to play against himself, which should not be allowed.

Impact: Owner of Ram NFT token can easily increase value without chance of losing because he is playing against himself, it is win-win situation. This is not desired behavior, because that player could easily become selected Ram within few function calls.

Proof of Concept:

1. `ChoosingRam::increaseValuesOfParticipants` function allows to increase value of Ram NFT. Function accepts token id of challenger (caller) and token id of any participant that also holds Ram NFT. Problem arises because caller can input his token id both as challenger and participant and function does not have check for this scenario. This means challenger can play versus himself, which shouldn't be allowed.

    1. Player mints Ram NFT with token id 0.
    2. Assert that token id 0 is not Jita Krodhah.
    3. Player calls `ChoosingRam::increaseValuesOfParticipants` function with token id 0 as challenger and token id 0 as participant. Token increased value to Jita Krodhah.
    4. Player calls `ChoosingRam::increaseValuesOfParticipants` function again with token id 0 as challenger and token id 0 as participant. Token increased value to Dhyutimaan.

```
    // @audit - caller can play against himself
    function increaseValuesOfParticipants(uint256 tokenIdOfChallenger, uint256
tokenIdOfAnyPerticipent)
        public
        RamIsNotSelected
    {
        if (tokenIdOfChallenger > ramNFT.tokenCounter()) {
            revert ChoosingRam__InvalidTokenIdOfChallenger();
        }
        if (tokenIdOfAnyPerticipent > ramNFT.tokenCounter()) {
            revert ChoosingRam__InvalidTokenIdOfPerticipent();
        }
        if (ramNFT.getCharacteristics(tokenIdOfChallenger).ram != msg.sender) {
            revert ChoosingRam__CallerIsNotChallenger();
        }
        .
        .
    }
```

2. Place the following test into `Dussehra.t.sol`.

```
    function test_challengerCanPlayVsHimself() public participants {
        assertTrue(ramNFT.getCharacteristics(0).isJitaKrodhah == false);

        vm.startPrank(player1);
        choosingRam.increaseValuesOfParticipants(0, 0);

        assertTrue(ramNFT.getCharacteristics(0).isJitaKrodhah == true);

        choosingRam.increaseValuesOfParticipants(0, 0);

        assertTrue(ramNFT.getCharacteristics(0).isDhyutimaan == true);
    }
```

Recommended Mitigation:

1. Add additional check in `ChoosingRam::increaseValuesOfParticipants` function to prevent player from playing against himself.

```
+   error ChoosingRam__CannotPlayAgainstYourself();

    .
    .

    function increaseValuesOfParticipants(uint256 tokenIdOfChallenger, uint256
tokenIdOfAnyPerticipent)
        public
        RamIsNotSelected
```

```
    {
        if (tokenIdOfChallenger > ramNFT.tokenCounter()) {
            revert ChoosingRam__InvalidTokenIdOfChallenger();
        }
        if (tokenIdOfAnyPerticipent > ramNFT.tokenCounter()) {
            revert ChoosingRam__InvalidTokenIdOfPerticipent();
        }
        if (ramNFT.getCharacteristics(tokenIdOfChallenger).ram != msg.sender) {
            revert ChoosingRam__CallerIsNotChallenger();
        }
+       if (ramNFT.getCharacteristics(tokenIdOfAnyPerticipent).ram == msg.sender)
{
+           revert ChoosingRam__CannotPlayAgainstYourself();
+       }

        if (block.timestamp > 1728691200) {
            revert ChoosingRam__TimeToBeLikeRamFinish();
        }
```

Low Risk Findings

[L-1] The organiser can predict the outcome of `ChoosingRam::selectRamIfNotSelected` and select the user who will get the reward

Description: The function `ChoosingRam::selectRamIfNotSelected` depends on a random value to select the participant to be selected as Ram. This function generates the random number by using `block.timestamp` and `block.prevrandao` values. Those values are considered a bad source of randomness. The organiser can predict the outcome and execute the function only if the desired user will become Ram. The selected Ram will take the reward.

Impact: The bad source of randomness gives the opportunity to the organiser to select the specific user which will become Ram and which will get the reward. Although, the organiser is considered to be trusted, the desired logic of the contract is not implemented correctly. The random number is not really a random number.

Proof of Concept:

1. Using `block.timestamp` as a source of randomness is commonly advised against, as the outcome can be manipulated by calling contracts. Also, for some chains like zkSync `block.prevrandao` is a constant value. This will allow the users to predict the result of the calculated number in Line 90 of `ChoosingRam.sol`: `uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao))) % ramNFT.tokenCounter();`. This will give the organiser the opportunity to break the random selection of the Ram and to select a specific user who will collect the reward.

The following code demonstrates how an attack can be executed.

```
function test_ramSelectionIsNotRandom() public {
    Dussehra dussehra;
    RamNFT ramNFT;
    ChoosingRam choosingRam;
    address organiser = makeAddr("organiser");
    address player1 = makeAddr("player1");
    address player2 = makeAddr("player2");
    address player3 = makeAddr("player3");

    vm.startPrank(organiser);
    ramNFT = new RamNFT();
    choosingRam = new ChoosingRam(address(ramNFT));
    dussehra = new Dussehra(1 ether, address(choosingRam), address(ramNFT));
    ramNFT.setChoosingRamContract(address(choosingRam));
    vm.stopPrank();

    vm.startPrank(player1);
    vm.deal(player1, 1 ether);
    dussehra.enterPeopleWhoLikeRam{value: 1 ether}();
    vm.stopPrank();

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

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

    // the organiser wants player2 to become Ram
    vm.startPrank(organiser);
    uint256 time = 1728691200 + 1;
    // the loop will execute until player2 is the Ram
    while (true) {
        vm.warp(++time);
        uint256 random = uint256(
            keccak256(abi.encodePacked(block.timestamp, block.prevrandao))
        ) % ramNFT.tokenCounter();

        // the outcome of the random calculation is checked
        if (ramNFT.getCharacteristics(random).ram == player2) {
            // if the player2 will be the Ram then
            // selectRamIfNotSelected is executed
            choosingRam.selectRamIfNotSelected();
            break;
        }
    }
    vm.warp(time);
    vm.stopPrank();

    // it is confirmed that player2 is the Ram
```

```
        assertEq(choosingRam.isRamSelected(), true);
        assertEq(choosingRam.selectedRam(), player2);
    }
```

Recommended Mitigation: Consider using a decentralized oracle for the generation of random numbers, such as Chainlinks VRF. The Chainlink VRF gives two methods to request randomness: subscription and direct funding method. They will have their added cost, but will solve the randomness issues of the `Dussehra` contract.

[L-2] Timezone Discrepancies in Arbitrum Chain

Description: The `ChoosingRam::selectRamIfNotSelected` function in the protocol is susceptible to timing issues when deployed on the Arbitrum chain due to timestamp discrepancies.

Impact: In some timezone organiser could select ram before 12 October 2024.

Proof of Concept:

1. The function currently checks timestamps as follows:

```
function selectRamIfNotSelected() public RamIsNotSelected OnlyOrganiser {
    if (block.timestamp < 1728691200) {
        revert ChoosingRam__TimeToBeLikeRamIsNotFinish();
    }
    if (block.timestamp > 1728777600) {
        revert ChoosingRam__EventIsFinished();
    }
    uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp,
block.prevrandao))) % ramNFT.tokenCounter();
    selectedRam = ramNFT.getCharacteristics(random).ram;
    isRamSelected = true;
}
```

2. However, due to Arbitrum's timestamp lower boundary policy, which is 24 hours earlier than the current time, the function may allow the selection of Ram prematurely in certain timezones. For example, in timezones like Pacific/Honolulu, the organiser could incorrectly enable Ram selection on 11 October 2024 instead of 12 October 2024.

Recommended Mitigation: Specify in the protocol's documentation and function comments that all timestamps should be considered based on UTC time to maintain consistency across different timezones and chains.

[L-3] Incorrect timestamp used for event start and end in `Dussehra::killRavana` which deviates from the timestamp mentioned in docs

Description: Incorrect timestamps are used in `killRavana` to ensure that the time at which it is called is of Dussehra event. The documentation mentions that the Dussehra event starts from 12th Oct and will finish

before 13th Oct, but the timestamps that are used for check in `killRavana` function are different.

Impact: `killRavana` can be called outside of the time mentioned in the docs.

Proof of Concept:

1. The vulnerability is present in the `killRavana` function where it uses incorrect timestamps for checking the timestamp is between the start and end timestamp of Dussehra.
2. But the timestamp used in the function is different from the one which is mentioned in the documentation as a result of which the user can call the function in different time which deviates from the documentation.

```
if (block.timestamp < 1728691069) {
    revert Dussehra__MahuratIsNotStart();
}
if (block.timestamp > 1728777669) {
    revert Dussehra__MahuratIsFinished();
}
```

3. The docs mentions that the `killRavana` will be available after 12th October 2024 and before 13th October 2024.
4. The actual timestamp that corresponds to 12th October 2024 is `1728691200` and to 13th October is `1728777600` but different timestamps are used.

Recommended Mitigation:

1. Correct the timestamps according to the docs:

```
- if (block.timestamp < 1728691069) {
+ if (block.timestamp < 1728691200) {
     revert Dussehra__MahuratIsNotStart();
 }
- if (block.timestamp > 1728777669) {
+ if (block.timestamp > 1728777600) {
     revert Dussehra__MahuratIsFinished();
 }
```

[L-4] 1 wei will remain in `Dussehra` contract if both entrance fee and number of participants are odd numbers

Description: The code in `Dussehra` contract calculates the rewards for the Ram and the Organiser by division of the total amount over 2 and uses the same value for both rewards. It does not consider if the total amount value is even or odd.

Impact: Dust amount of 1 wei will remain in the Dussehra contract if the calculated `totalAmountByThePeople` is odd number. This will happen always for odd entrance fee and number of participants but only 1 wei will be lost.

Proof of Concept:

1. The code in `Dussehra` contract calculates the the total amount as `uint256 totalAmountByThePeople = WantToBeLikeRam.length * entranceFee;` Line 75, Dussehra.sol. This value might be odd number if the number of participants and the entrance fee are both odd numbers. The calculated rewards in the next line of code will be rounded and 1 wei will remain in the contract.

2. The following test demonstrates the vulnerability.

```
function test_dustAmountStaysInDussehraContract() public {
    Dussehra dussehra;
    RamNFT ramNFT;
    ChoosingRam choosingRam;
    address organiser = makeAddr("organiser");
    address player1 = makeAddr("player1");
    address player2 = makeAddr("player2");
    address player3 = makeAddr("player3");

    vm.startPrank(organiser);
    ramNFT = new RamNFT();
    choosingRam = new ChoosingRam(address(ramNFT));
    dussehra = new Dussehra(1 wei, address(choosingRam), address(ramNFT));
    ramNFT.setChoosingRamContract(address(choosingRam));
    vm.stopPrank();

    vm.startPrank(player1);
    vm.deal(player1, 1 wei);
    dussehra.enterPeopleWhoLikeRam{value: 1 wei}();
    vm.stopPrank();

    vm.startPrank(player2);
    vm.deal(player2, 1 wei);
    dussehra.enterPeopleWhoLikeRam{value: 1 wei}();
    vm.stopPrank();

    vm.startPrank(player3);
    vm.deal(player3, 1 wei);
    dussehra.enterPeopleWhoLikeRam{value: 1 wei}();
    vm.stopPrank();

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

    // Organiser gets his reward
    vm.startPrank(player2);
    dussehra.killRavana();
    vm.stopPrank();

    // The Ram gets his reward
    vm.startPrank(player3);
```

```
        dussehra.withdraw();
        vm.stopPrank();

        // 1 wei still remains in the contract
        assertEq(address(dussehra).balance, 1 wei);
    }
```

Recommended Mitigation:

1. Calculate the reward of the Ram by using subtraction. See the code below.

```
function killRavana() public RamIsSelected {
    if (block.timestamp < 1728691069) {
        revert Dussehra__MahuratIsNotStart();
    }
    if (block.timestamp > 1728777669) {
        revert Dussehra__MahuratIsFinished();
    }
    IsRavanKilled = true;
    uint256 totalAmountByThePeople = WantToBeLikeRam.length * entranceFee;
-   totalAmountGivenToRam = (totalAmountByThePeople * 50) / 100;
+   uint256 totalAmountGivenToOrganiser = totalAmountByThePeople * 50 / 100;
+   totalAmountGivenToRam = totalAmountByThePeople - totalAmountGivenToOrganiser;
-   (bool success, ) = organiser.call{value: totalAmountGivenToRam}("");
+   (bool success, ) = organiser.call{value: totalAmountGivenToOrganiser}("");
    require(success, "Failed to send money to organiser");
}
```