HAECHI AUDIT

Iskra - NFT Marketplace

Smart Contract Security Analysis

Published on : Nov 17, 2022

Version v1.1

HAECHI AUDIT

Smart Contract Audit Certificate

Iskra - NFT Marketplace

Security Report Published by HAECHI AUDIT
v1.1 Nov 17, 2022 - add final commit
v1.0 Nov 4, 2022

Auditor : Paul Kim, Jinu Lee

[illegible] [illegible]

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	-	-	-	-
High	-	-	-	-
Medium	1	1	-	-
Low	1	1	-	-
Tips	3	3	-	-

TABLE OF CONTENTS

ABOUT US

The most reliable web3 security partner.

HAECHI AUDIT is a flagship service of HAECHI LABS, the leader of the global blockchain industry. We bring together the best Web2 and Web3 experts. Security Researchers with expertise in cryptography, leaders of the global best hacker team, and blockchain/smart contract experts are responsible for securing your Web3 service.

We have secured the most well-known web3 services including 1inch, SushiSwap, Klaytn, Badger DAO, SuperRare, Netmarble, Klaytn and Chainsafe. We have secured $60B crypto assets on over 400 main-nets, Defi protocols, NFT services, P2E, and Bridges.

HAECHI AUDIT is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@haechi.io

Website: audit.haechi.io

Executive Summary

Purpose of this report

This report was prepared to audit the security of the NFT Marketplace contracts developed by the Iskra team. HAECHI AUDIT conducted the audit focusing on whether the system created by the Iskra team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the NFT Marketplace.
In detail, we have focused on the following

- Arbitrary NFT Takeover.
- Project availability issues like Denial of Service.
- Off-Chain Oracle Data.

Codebase Submitted for the Audit

The codes used in this Audit can be found on GitHub (https://github.com/iskraworld/iskra-contracts).

The last commit of the code used for this Audit is "bc467fab9b248b9a747062944e02522012593bc2".

For the patch review, the last commit is "3d5a6d443d22d950b69c9e6b00835b9b19ba939b"
Patch review only applies to previous audit reports. A newly developed feature (MysteryCard) was not audited.

Audit Timeline

Date	Event
2022/10/17	Audit Initiation (Iskra MKP)
2022/11/04	Delivery of v1.0 report.

Findings

HAECHI AUDIT found 1 medium and 1 Low severity issues. There are 3 Tips issues explained that would improve the code's usability or efficiency upon modification.

Severity	Issue	Status
Medium	increaseAmount and decreaseAmount function has no cardtype check logic.	(Found - v1.0) (Resolved - v1.1)
TIPS	batchUnregisterCards don't need to check the owner of given _cardIds.	(Found - v1.0) (Resolved - v1.1)
Low	Lack of signed data can cause signature replay attack.	(Found - v1.0) (Resolved - v1.1)
TIPS	Contract doesn't check the return value of the transfer function.	(Found - v1.0) (Resolved - v1.1)
TIPS	TokenPricer contract must validate expiration value.	(Found - v1.0) (Resolved - v1.1)

#ID	Title	Type	Severity	Difficulty
1	increaseAmount and decreaseAmount function has no cardtype check logic.	Logic Error	**Medium**	**Medium**
2	batchUnregisterCards don't need to check the owner of given _cardIds.	N/A	**Tips**	**N/A**
3	Lack of signed data can cause signature replay attack.	Input Validation	**Low**	**Medium**
4	Contract doesn't check the return value of the transfer function.	Logic Error	**Tips**	**Medium**
5	TokenPricer contract must validate expiration value.	Input Validation	**Tips**	**N/A**

Remarks

The project has some contracts like iskra token, pioneer and vesting. The audit scope is only for the mkp (Marketplace).

OVERVIEW

Protocol overview

Iskra MKP have the following access control mechanisms. Most of the functions of Iskra MKP are managed by the Iskra team.

- onlyOwner: Iskra team
- onlyGameOwner: Game Owner(must be added by Iskra team)
- onlyMarket: MarketPlace Contract
- user: user can purchase NFTs registered by GameOwner.

The owner has permissions that can change the crucial part of the system. It is highly recommended to maintain the private key as securely as possible and strictly monitor the system state changes.

▪ GameContractManager

GameContractManager is a contract for managing information management of GameContract and sales revenue of Game NFT. Except for the view function of GameContractManager, all functions can only be executed by privileged users and privileged contracts. GameContract information setting can only be executed by the owner (Iskra team). Functions related to sales revenue can only be called by the MarketPlace contract.

The contract is deployed using the UUPS proxy contract.

▪ MarketPlace

MarketPlace is a contract that can trade NFTs. To sell NFT, it must be an NFT of a game registered in GameContractManager, and only GameOwner can register for sale. Anyone registered for sale can purchase NFTs, but the *PurchaseCheckerByVerifying* contract allows only authenticated users to purchase NFTs. Anyone can purchase NFTs that are registered for sale. If the seller(GameOwner) has set up *PurchaseCheckerByVerifying* , only authorized users can purchase NFTs. The price of NFT is set in USD, and it can be purchased using the payment method (erc20-based token) registered by Iskra Team and Game Owner. In MarketPlace, when purchasing NFTs, Oracle receives price information from off-chain to calculate the value of the payment method in USD. **The off-chain price oracle part is not included in the audit scope.**

The contract is deployed using the UUPS proxy contract.

Scope

```
contracts
└── mkp
    ├── GameContractManager.sol
    ├── IPurchaseChecker.sol
    ├── ITokenPricer.sol
    ├── MarketCardParser.sol
    ├── MarketPlace.sol
    ├── MarketPlaceConstant.sol
    ├── PurchaseCheckerByVerifying.sol
    ├── ServerSignVerifier.sol
    └── TokenPricerByVerifying.sol
```

FINDINGS

1. increaseAmount and decreaseAmount function has no cardtype check logic.

ID: ISKRA-MKP-01
Type: Logic Error
File: contracts/mkp/Marketplace.sol

Severity: Medium
Difficulty: Medium

Issue

The *increaseAmount()* and *decreaseAmount()* functions don't have cardtype checking logic. This will change *marketCards[_cardId].availableCardAmount* value of ERC721.

```
   function increaseAmount(uint256 _cardId, uint256 _cardAmount) external nonReentrant
onlyCardOwner(_cardId) {
        require(_cardAmount > 0, "the increasing card amount is zero");

        transferToken(
            marketCards[_cardId],
            _cardAmount,
            gameManager.getOwner(marketCards[_cardId].gameContract),
            address(this)
        );

        // change marketCards
        marketCards[_cardId].availableCardAmount += _cardAmount;

        // event
        emit AmountIncreased(_cardId, _cardAmount,
marketCards[_cardId].availableCardAmount);
    }

    function decreaseAmount(uint256 _cardId, uint256 _cardAmount) external nonReentrant
onlyCardOwner(_cardId) {
        require(_cardAmount > 0, "the decreasing card amount is zero");

        transferToken(
            marketCards[_cardId],
            _cardAmount,
            address(this),
            gameManager.getOwner(marketCards[_cardId].gameContract)
        );

        // change marketCards
        marketCards[_cardId].availableCardAmount -= _cardAmount;
        // event
        emit AmountDecreased(_cardId, _cardAmount,
marketCards[_cardId].availableCardAmount);
    }
```

[https://github.com/iskraworld/iskra-contracts/blob/bc467fab9b248b9a747062944e02522012593bc2/contracts/mkp/MarketPlace.sol#L511]

Recommendation

We recommend the iskra team to add the logic for checking whether the cardtype is ERC721 or not.

2. batchUnregisterCards doesn’t need to check the owner of given _cardIds.

ID: ISKRA-MKP-02
Type: N/A
File: contracts/mkp/Marketplace.sol

Severity: Tips
Difficulty: N/A

Issue

The *batchUnregisterCards()* function is checking the owner of _cardIds but *unregisterCard()* function already has the *onlyCardOwner(_cardId)* modifier.

```
   function batchUnregisterCards(uint256[] memory _cardIds) public {
         require(_cardIds.length > 0, "the card id array is empty");

         for (uint256 i = 0; i < _cardIds.length; i++) {
             address _gameContract = getGameContract(_cardIds[i]);
             require(gameManager.getOwner(_gameContract) == msg.sender, "invalid card
id");

             unregisterCard(_cardIds[i]);
         }
     }
```

[https://github.com/iskraworld/iskra-contracts/blob/bc467fab9b248b9a747062944e02522012593bc2/contracts/mkp/MarketPlace.sol#L500]

Recommendation

```
             address _gameContract = getGameContract(_cardIds[i]);
             require(gameManager.getOwner(_gameContract) == msg.sender, "invalid card
id");
```

We recommend removing the above code.

3. Lack of signed data can cause signature replay attack.

ID: ISKRA-MKP-03
Type: Input Validation
File: contracts/mkp/ServerSignVerifier.sol

Severity: Low
Difficulty: Medium

Issue

Iskra MKP is using *ServerSignVerifier* contract to use token price and whitelist buy ticket data from Off-chain. When iskra specifies a trusted signer, and the data in the Off-Chain is signed by that trusted signer and passed to the On-Chain, the contract is implemented to verify that the signer of the data is a trusted signer in the ServerSignVerifier contract. There are two contracts that inherit the *ServerSignVerifier* contract: *PurchaseCheckerByVerifying* and *TokenPricerByVerifying*. Signature data used by both contracts does not include purpose and scope. If the contracts inheriting the *ServerSignVerifier* contract use the same signer, a signature replay attack may occur.

Recommendation

We recommend you apply the two methods below.

- Use signers differently for contracts that inherit the ServerSignVerifier. This will minimize damage if the key is leaked due to an accident. You can manage signers to avoid duplication through logging in Off-Chain, or you can create a signer registry in On-Chain for use in Server SignVerifier contracts.
- Change the signature data structure as below or apply EIP712. Signature information becomes clearer and cannot be reused by other contracts.

```
// bytes32 public constant SIG = keccak256("TokenPricerByVerifying");
// PriceParams memory _params
abi.encode(SIG, address(this), _params);
bytes32 sig;
address addr;
(
    sig,
    addr,
    _params.amountOfKeyToken,
    _params.keyTokenAddress,
    _params.amountOfDestinationToken,
    _params.destinationTokenAddress,
    _params.expiration
) = abi.decode(_encoded, (bytes32, address, uint256, address, uint256, address,
uint64));

require(sig == SIG, "check sig type");
require(addr == address(this), "check sig type");
```

4. Contract doesn’t check the return value of the transfer function.

ID: ISKRA-MKP-04

Type: Logic Error

File: contracts/mkp/Marketplace.sol

Severity: Tips

Difficulty: Low

Issue

When transferring ERC20 from MarketPlace Contract, the return value is not checked. If you use a token that returns false without reverting when the *transfer* function fails, the function can be executed even if the *transfer* function fails.

```
    function calculatePayment(
        uint256 _cardId,
        uint256 _payment,
        address _tokenAddress
     )
        internal
        returns (
            uint256 _purchaserFee,
            uint256 _sellerRevenue,
            uint256 _salesFee
        )
     {
       ...
        IERC20(_tokenAddress).transferFrom(msg.sender, iskraIncomeWallet,
_purchaserFee);
        IERC20(_tokenAddress).transferFrom(msg.sender, address(this), _payment -
_purchaserFee);
...
     function claimRevenue(address _gameContract) external nonReentrant
onlyGameOwnerOrOwner(_gameContract) {
        ...
        for (uint256 i = 0; i < _revenueTokens.length; i++) {
            // transfer seller's revenue to game owner
            IERC20(_revenueTokens[i]).transfer(_gameOwner, _revenues[i]);

            // transfer sales fee to iskra income wallet
            IERC20(_revenueTokens[i]).transfer(iskraIncomeWallet, _salesFees[i]);
```

[https://github.com/iskraworld/iskra-contracts/blob/bc467fab9b248b9a747062944e02522012593bc2/contracts/mkp/MarketPlace.sol#L594]

Recommendation

Use SafeERC20 when the contracts *transfer* ERC20 token.

5. TokenPricer contract must validate expiration value.

ID: ISKRA-MKP-05	Severity: Tips
Type: Input Validation	Difficulty: N/A
File: contracts/mkp/TokenPricerByVerifying.sol	

Issue

The client is currently setting the expiration value. The *TokenPricerByVerifying* contract does not know when the data was created and signed. If there is a price change with a client setting the expiration high, it can cause a problem.

```
    function getPrice(bytes memory _data) external view override returns (PriceParams
memory _params) {
        bytes memory _encoded = verifyDataAndGetEncoded(_data);

        (
            _params.amountOfKeyToken,
            _params.keyTokenAddress,
            _params.amountOfDestinationToken,
            _params.destinationTokenAddress,
            _params.expiration
        ) = abi.decode(_encoded, (uint256, address, uint256, address, uint64));
```

[https://github.com/iskraworld/iskra-contracts/blob/bc467fab9b248b9a747062944e02522012593bc2/contracts/mkp/TokenPricerByVerifying.sol#L15]

Recommendation

Add a signature creation time (expected to be the same as the data generation time) to the data used by the *TokenPricerByVerifying* contract and verify that the expiration value is appropriate. As a result of asking before opening the issue on github, I was informed that Iskra uses 1 minute as expiration due to policy. It is recommended that the expansion limit is also implemented on the contract to verify that it is set within 1 minute or less.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure smart contracts, correction of discovered problems and sufficient testing thereof are required.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	**Medium**	**High**
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	▪ Integer under/overflow vulnerability ▪ floating point and rounding accuracy
Access & Privilege Control	▪ Manager functions for emergency handle ▪ Crucial function and data access ▪ Count of calling important task, contract state change, intentional task delay
Denial of Service	▪ Unexpected revert handling ▪ Gas limit excess due to unpredictable implementation
Miner Manipulation	▪ Dependency on the block number or timestamp. ▪ Frontrunning
Reentrancy	▪Proper use of Check-Effect-Interact pattern. ▪Prevention of state change after external call ▪ Error handling and logging.
Low-level Call	▪ Code injection using delegatecall ▪ Inappropriate use of assembly code
Off-standard	▪ Deviate from standards that can be an obstacle of interoperability.
Input Validation	▪ Lack of validation on inputs.
Logic Error/Bug	▪ Unintended execution leads to error.
Documentation	▪Coherency between the documented spec and implementation
Visibility	▪ Variable and function visibility setting
Incorrect Interface	▪ Contract interface is properly implemented on code.

End of Document