PlatoFarm Audit Report

Version 1.0.3

Serial No. 2021101400042017

Presented by Fairyproof

October 14, 2021

FAIRYPROOF

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the PlatoFarm project, at the request of the SecondSpace team.

Audit Start Time:

Sep 15, 2021

Audit End Time:

Sep 18, 2021

Audited Source Files:

The calculated SHA-256 values for the audited files when the audit was done are as follows:

```
Address.sol        :
0x0a15051336746f717ac5e409b5187e02b3909e518be4ff9676fa951864ee2562
BlindBox.sol       :
0xedfad704fb29d79f68b56d28ae13c13980c8574f29096584a51c96e82fb7893c
Context.sol        :
0xb6157e2f04c5f5a3978f0b1f58adcaee8f8cd6c6588255565e255e4321ed8962
EnumerableMap.sol  :
0xd642f5550907a3761087b3c0ed755d0bea2f5e818a771407562173cab38cdc3f
EnumerableSet.sol  :
0x34d250b8ad1e340a4895e675d50fd61dc7c4c1af71e78f181df92bf5515bc6f5
FreeMarket.sol     :
0xc6e98b665d63e05cf4205907f89a379eb3a2a80dc3e73f75e91c6813f4a9d2e4
IERC20Interface.sol:
0x87af74c7b29a4c0efb1e79d82dc47dce70930dbf22c7323563d878159b6e4c9a
Ownable.sol        :
0x5049e01e4ff811da2a334cbbcfe699ab7ae6822adb119308d4b573ab3ef71970
PlatoNFT.sol       :
0xe32e44b414d175ad6fe51286a7cb7ab38c5d9038d06feaf92a4b6b9cf4a6adc4
ReentrancyGuard.sol:
0xddcca2fb3f5b9dfdee407b871d17b609ded65cdd9ebdb47adf164bd254d3816e
SafeERC20.sol      :
0x4f165e3b6b22e05c5f9ab8b3229a5a473c2171c3908eefe9e9a125a51898dcfd
SafeMath.sol       :
0xcad066aa4b1b82da17688185639e991dc63185986a66e97d63a0afda12e7aca5
Strings.sol        :
0xb5eb7dc2ca7755e912808f59306cd3139f5cec60952983abeaf0a34280e9f0ec
```

The source files audited include all the files with the extension "sol" as follows:

```
./
├── Address.sol
├── BlindBox.sol
```

```
├── Context.sol
├── EnumerableMap.sol
├── EnumerableSet.sol
├── FreeMarket.sol
├── IERC20Interface.sol
├── Ownable.sol
├── PlatoNFT.sol
├── ReentrancyGuard.sol
├── SafeERC20.sol
├── SafeMath.sol
└── Strings.sol

13 files
```

The goal of this audit is to review PlatoFarm's solidity implementation for an NFT game, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

This audit only applies to the specified code, software or any materials supplied by the SecondSpace team for specified versions. Whenever the code, software, materials, settings, enviroment etc is changed, the comments of this audit will no longer apply.

— Disclaimer

Note that as of the date of publishing, the contents of this report reflect the current understanding of known security patterns and state of the art regarding system security. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. If the audited source files are smart contract files, risks or issues introduced by using data feeds from offchain sources are not extended by this review either.

Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites,

any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

— Methodology

The above files' code was studied in detail in order to acquire a clear impression of how the its specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

The Fairyproof auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Fairyproof to make sure we understand the size, scope, and functionality of the project's source code.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Fairyproof describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run the test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the source code to improve maintainability, security, and control based on the established industry and academic practices, recommendations, and research.

— Structure of the document

This report contains a list of issues and comments on all the above source files. Each issue is assigned a severity level based on the potential impact of the issue and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

For this audit, we used the following sources of truth about how the NFT game should work:

https://www.secondspace.game

These were considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the SecondSpace team or reported an issue.

— Comments from Auditee

No vulnerabilities with critical, high, medium or low-severity were found in the above source code.

02. About Fairyproof

Fairyproof is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Fairyproof has developed industry security standards for designing and deploying blockchain applications.

03. Introduction to PlatoFarm

Plato Farm is a planting NFT management chain game based on HECO. The game is very interesting. Players own a small farm in the game, where they can grow (raise) animals and plants, harvest agricultural products, and sell them to buy other NFT props, building the barren land into a beautiful and fertile village or city. Players can also establish their own game conference.

04. Major functions of audited code

The audited code implements the following functions:

- Issurance of NFT Tokens
  - ERC-721 based
  - Interface for Token Issurance
  - NFT's tokenId is generated based on tokenType, tokenItem and a serial id. A serial id is incremental.
  - The max supply for each NFT token under a specific tokenType is `1e11`
  - The Admin can define new or disable existing tokenTypes
- NFT Market
  - Users can trade NFT tokens

    - Users can create an order including multiple NFT tokens of a same type and a same item and set a uniform price for these NFT tokens
    - Users can change NFT token's price
    - Users need to pay a transaction fee for each NFT token transaction. By default the transaction fee is set to 0.1%
    - The Admin can change the transaction fee and the max transaction fee is 1%.
- NFT Blind Box
    - `tokenType` and `tokenItem` are randomly generated and used to issue tokens
    - Able to mint additional tokens by authorization

05. Key points in audit

During the audit, we worked closely with the SecondSpace team, helped fix some issues and refine some code. Here are the main work items:

- Fixed a Re-entrancy Issue in Blind Box

Source and Description:

In line 149 of `BlindBox.sol` , the `BlindBox.random` function had the following code section:

```
        uint256 _seed = uint256(
            keccak256(
                abi.encodePacked(
                    block.timestamp
                    .add(block.difficulty)
                    .add(uint256(keccak256(abi.encodePacked(block.coinbase))) / now)
                    .add(block.gaslimit)
                    .add(uint256(keccak256(abi.encodePacked(msg.sender))) / now)
                    .add(block.number)
                )
            )
        );
```

All the parameters excepte msg.sender are predictable. It is very likely that an attacker exploits this vulnerability to use multiple contracts as msg.senders to repeatedly call this function and get desirable seeds to do re-entrancy attacks while revert transactions which generate undesirable seeds.

Recommendation:

Consider disallowing contracts to call this function and only allowing external accounts to call this function.

Update: it has been fixed in the latest code.

- Refined Insecure Randomness in Blind Box

Source and Description:

In line 149 of `BlindBox.sol`, the `BlindBox.random` function used insecure seeds to generate random numbers. Among all the parameters that constitued a seed, coinbase, block.number and timestamp are predictable. In blockchains such as BSC and HECO, which use POA as their consensus, `block.difficulty` is predictable therefore a seed based on these parameters could be predicted.

Recommendation:

Consider using a VRF algorithm or Chainlink to generate a seed for random numbers

Update: the SecondSpace team has made some changes in the latest code and will fix this issue in the next upgrade.

- Fixed a Bug in the _remove Function in the MarketOrders Contract, Which Could Cause an Order Not Properly Removed

Source and Description:

In line 61 of `FreeMarket.sol`, the `MarketOrders._remove` function had the following code section:

```
    /**
     * @dev Removes the order from order list.
     * Returns true if the order was removed from the list.
     */
    function _remove(uint256 _orderId) internal returns (bool) {
        if (orders[_orderId].owner != address(0)) {
            uint256 lastIndex = orderList.length - 1;
            orderList[lastIndex].id = _orderId;
            orderList[_orderId] = orderList[lastIndex];
            orderList.pop();
            delete orders[lastIndex];
            return true;
        } else {
            return false;
        }
    }
```

The desired behavior of this function was to swap a specified `Order` in `orderList` with the one at the end of `OrderList` and then the specified `Order` would be removed by calling `pop`. However the implementation didn't do so, it just swapped two orders' `orderIds`, not their `Order` entities.

Recommendation:

Consider swapping two `Orders` not just their `order ids`. In addition consider referring to Openzeppelin's implementation for `remove` in the `EmerationSet` contract.

We don't suggest swapping `ids` since an order's id is in general strictly associated with the order it specifies.

Update: it has been fixed in the latest code.

- Refined Order's Data Structure in the MarketOrders Contract to Reduce Gas Consumption

Source and Description:

In line 40 of `FreeMarket.sol`, the `MarketOrders.orders` variable had the following definition:

```
    Order[] public orderList;

    mapping (uint256 => Order) public orders;    // Order by index of order list
```

This definition had both array and mapping. This would incur relatively high gas consumption.

Recommendation:

Consider putting `Order` in a `list` and defining the `value` in the `mapping` as index. Please refer to Openzeppelin's design for `EmerationSet`.

Update: it has been fixed in the latest code.

- Refined Implementation of the orderList.remain Function in FreeMarket.buyOrder

Source and Description:

In line 215 of the `FreeMarket.sol` contract, the `MarketOrders.buyOrder` function had the following code section:

```
        for (uint256 i = order_.remain - 1; i >= order_.remain - amount; i--) {
            //...
            orderList[orderId].remain--;
            orders[orderId].remain--;
        }
```

The directives with `remain` can be simplied and removed from the loop to reduce the gas consumption.

Recommendation:

Consider removing the directives with `remain` from the loop and use `orderList[orderId].remain - amount` to replace them.

Update: it has been fixed in the latest code.

- Added Boundaries to FreeMarket's Transaction Fees

Source and Description:

In line 260 of the `FreeMarket.sol` contract, the `MarketOrders.setFee` function didn't have boundaries for the transaction fees.

Recommendation:

Consider adding boundaries to the transaction fees. When the transaction fee was set to 0, a buy order could fail. If the transaction fee was set to be greater than 0 this issue could be fixed.

Update: it has been fixed in the latest code.

- Fixed a Bug in Algorithm to Calculate a Token's Item

Source and Description:

In line 143 of the `FreeMarket.sol` contract, the `MarketOrders.getItem` function had the following code section:

```
function getItem(uint256 _tokenId) public pure returns(uint256) {
    return _tokenId.div(itemMax).mod(10);
}
```

Based on the rules defined in the `PlatoNFT` contract, which generate a `tokenId`, each `type` has 100 items. However the `getItem` function only returned 10 items. If various types are considered the number of items should be:

$$item - num = type - num * 100$$

So the function's algorithm was incorrect.

Recommendation:

Consider doing either of the two ways:

If various types are considered, the number of items returned are `_tokenId.div(itemMax)` otherwise the number of items are `_tokenId.div(itemMax).mod(100)`.

Update: the SecondSpace team agrees that the algorithm to calculate `item` should take various types into account and the correct algorithm should be `_tokenId.div(itemMax)`. It has been fixed in the latest code.

- Fixed An Issue in Constant Definition Which Caused Confusion

Source and Description:

The following code had an issue:

```
        uint256 tokenId = tokenType
                        .mul(typeMax * 10)
                        .add(tokenItem.mul(itemMax * 10))
                        .add(currentSupplyCount.add(1));
```

The `itemMax` constant was defined both in `FreeMarket` and `PlatoNFT` and the values were `1e9` and `1e8` respectively. In `PlatoNFT`, the algorithm which used `itemMax` mutiplied `itemMax` by 10. Therefore `itemMax` could be defined as a universal value used in both.

Recommendation:

Consider defining `itemMax` as a constant value used in both `FreeMarket` and `PlatoNFT` to prevent confusion.

Update: it has been fixed in the latest code.

- Fixed an Overflow Issue in tokenItem in PlatoNFT

Source and Description:

In line 657 of the `PlatoNFT.sol` contract, the `PlatoNFT.mint` function had the following code section:

```
    function mint(address to, uint256 tokenType, uint256 tokenItem, string
memory tokenURI) public onlyMinter returns(uint256) {

        require(typeActive[tokenType], "token type not exist");
        require(itemSupply[tokenType][tokenItem] < itemMaxSupply, "Mint exceeds
the maximum limit of item");
```

In the code each `type` of `token` had 100 items. The value of `tokenItem` was a 2-digit number and the range was from 0~99. However if the value of `tokenItem` was greater than100, an overflow would happen and the `tokenType` would be unexpected.

Recommendation:

Consider adding a constraint to `tokenItem` and restricting its value to be less than 100

Update: it has been fixed in the latest code.

06. Coverage of issues

The issues that the Fairyproof team covered when conducting the audit include but are not limited to the following ones:

- Re-entrancy Attack
- DDos Attack
- Integer Overflow
- Function Visibility
- Logic Vulnerability
- Uninitialized Storage Pointer
- Arithmetic Precision
- Tx.origin
- Shadow Variable
- Design Vulnerability
- Token Issurance
- Asset Security
- Access Control

07. Severity level reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

08. List of issues by severity

A. Critical

- N/A

B. High

- N/A

C. Medium

- N/A

D. Low

- Insecure Randomness in Randome Number Generation

09. Issue descriptions

- Insecure Randomness in Randome Number Generation: Low

Source and Description:

The seed that is used to generate randome numbers for NFT issurance uses onchain data which doesn't provide secure randomness.

Recommendation:

Consider using a VRF algorithm or Chainlink to generate a seed.

Update: the SecondSpace team plans to to use offchain random numbers in the next upgrade.

10. Recommendations to enhance the overall security

We list some recommendations in this section. They are not mandatory but will enhance the overall security of the system if they are adopted.

- N/A