SMART CONTRACT AUDIT REPORT

for

MetroGalaxy

Prepared By: Yiqun Chen

PeckShield

January 4, 2022

Document Properties

Client	MetroGalaxy
Title	Smart Contract Audit Report
Target	MetroGalaxy
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 4, 2022	Yiqun Chen	Final Release
1.0-rc	January 4, 2022	Yiqun Chen	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `MetroGalaxy` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About MetroGalaxy

`MetroGalaxy` is a metaverse project that uniquely blends a social platform together with an online virtual game that lets its players role-play as anyone they want, do anything they like in an ever-expanding decentralized world. `Metronions` are citizens of `MetroGalaxy`, which are randomly generated with different traits and will be revealed after the NFT sale period ends. All `Metronions` can equip different kinds of accessories, with different scarcity. `Metronions` come in form of the `ERC721` standard, while accessories are `ERC1155`. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of MetroGalaxy

Item	Description
Name	MetroGalaxy
Type	Avalanche Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 4, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/metrogalaxy/metronion-sc.git (7c2dab4)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/metrogalaxy/metronion-sc.git (e7d1885)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `MetroGalaxy` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Market Bypass With Direct safeTransferFrom()	Business Logic	Confirmed
PVE-002	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-003	Low	Improved Sanity Checks In MetronionSale	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Market Bypass With Direct safeTransferFrom()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MetroGalaxyMarketplace
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

There are two kinds of tradable assets supported in the MetroGalaxy marketplace, (ERC721 and ERC1155), and each naturally has the standard implementation, e.g., transferFrom()/safeTransferFrom(). By design, each tradable asset listed for sale in MetroGalaxyMarketplace will be transferred from the owner to the market. Once sold, it will be transferred from the market to the buyer after the marketFee is collected. Note the current implementation does not support the royalty that may be credited to the original NFT creator.

```
    function buy(
        address assetAddr,
        uint256 assetId,
        address seller,
        uint256 priceInWei,
        uint256 amount
    ) external payable nonReentrant {
        require(!paused(), "MetroGalaxyMarketplace: contract is paused");

        address buyer = msg.sender;
        require(buyer != seller, "MetroGalaxyMarketplace: cannot buy your own assets");

        _requireAcceptedAssets(assetAddr);
        _requireValidAssetAmount(assetAddr, amount);

        bytes32 id = _getAssetId(assetAddr, assetId);
        Asset storage asset = listedAssets[id][seller];
        uint256 totalPrice = priceInWei * amount;
```

```
        require(asset.amount > 0 && amount <= asset.amount, "MetroGalaxyMarketplace: invalid amount");
        require(priceInWei == asset.priceInWei, "MetroGalaxyMarketplace: invalid price");

        uint256 marketFee = _getMarketFee(totalPrice);
        asset.amount -= amount;

        // transfer accepted token to seller
        acceptedToken.safeTransferFrom(buyer, seller, totalPrice - marketFee);
        acceptedToken.safeTransferFrom(buyer, owner(), marketFee);

        // transfer asset to buyer
        AcceptedAssets(assetAddr).safeTransferFrom(address(this), buyer, assetId, amount);

        emit AssetBought(assetAddr, assetId, buyer, seller, priceInWei, amount);
    }
```

Listing 3.1: CogiNFTMarket::buy()

To elaborate, we show above the `buy()` routine. This routine is used for buying a listed asset with proper tax payment. It comes to our attention that instead of paying the tax amount, it is possible for the current owner and the buyer to directly negotiate a price, without paying the `marketFee` (on behalf of `MetroGalaxyMarketplace`). The asset can then be arranged and delivered by the current owner to directly call `transferFrom()/safeTransferFrom()` with the buyer as the recipient.

Recommendation Implement a locking mechanism so that any assets need to be locked in the `MetroGalaxyMarketplace` contract in order to be available for public auction.

Status The discussion with the team has confirmed that it is allowed to trade assests directly without going through the marketplace. However, users should take responsibility for the risk of the trade.

3.2 Inconsistency Between Document and Implementation

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MetroGalaxyMarketplace`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found in line 102 of `MetroGalaxyMarketplace::offer()`, and line 135 of `MetroGalaxyMarketplace::cancelOffer()` of the same contract. Using the `offer()` routine as an example, the preceding function summary indicates that it can only be called by the account that is not the owner of the specified asset. However, our analysis shows that it can be called by anyone, although the owner can not take the offer after that.

```
99      /**
100      * @dev Place offer for assets
101      * If user want to update price or amount, user need to cancel current offer first
102      * Can only call by accounts that is not the owner
103      * @param assetAddr Asset address, should be in list supported address
104      * @param assetId Asset id
105      * @param priceInWei Price in wei
106      * @param amount Asset amount
107      */
108     function offer(
109         address assetAddr,
110         uint256 assetId,
111         uint256 priceInWei,
112         uint256 amount
113     ) external payable override nonReentrant {
114         require(!paused(), "MetroGalaxyMarketplace: contract is paused");
115         address buyer = msg.sender;
116         _requireAcceptedAssets(assetAddr);
117         _requireValidAssetAmount(assetAddr, amount);
118         require(priceInWei > 0, "MetroGalaxyMarketplace: invalid price");
119
120         bytes32 id = _getAssetId(assetAddr, assetId);
121         Asset storage asset = offeredAssets[id][buyer];
122         uint256 totalPrice = priceInWei * amount;
123
124         require(asset.amount == 0, "MetroGalaxyMarketplace: asset is already offered");
125
126         asset.priceInWei = priceInWei;
127         asset.amount = amount;
128         acceptedToken.safeTransferFrom(buyer, address(this), totalPrice);
129
130         emit AssetOffered(assetAddr, assetId, buyer, priceInWei, amount);
131     }
```

Listing 3.2: `MetroGalaxyMarketplace::offer()`

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been fixed by this commit: `e7d1885`.

3.3 Improved Sanity Checks In MetronionSale

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MetronionSale`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `MetroGalaxy` protocol is no exception. Specifically, if we examine the `MetronionSale` contract, it has defined a number of protocol-wide parameters, e.g., `_privateTime`, `publicTime`, and `endTime`. In the following, we show the constructor that configure them.

```
constructor(
      IMetronionNFT _nftContract,
      uint256 _versionId,
      uint256 _maxWhitelistSize,
      uint64 _privateTime,
      uint64 _publicTime,
      uint64 _endTime
  ) Whitelist(_maxWhitelistSize) {
      nftContract = _nftContract;
      versionId = _versionId;
      _saleConfig = SaleConfig({ privateTime: _privateTime, publicTime: _publicTime,
          endTime: _endTime });
  }
```

Listing 3.3: `MetronionSale::constructor()`

Our result shows the update logic on these time parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of the start time of the sale (`_privateTime_publicTime`) will revert the `buy()` operation.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `e7d1885`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `MetroGalaxy` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., pause/unpause the protocol and grant the `operator` role).

With great privilege comes great responsibility. In the following, we show representative privileged operations in the `MetroGalaxy` protocol.

```
    /**
     * @dev Call by only owner to pause the contract
     */
    function pause() external onlyOwner {
        _pause();
    }

    /**
     * @dev Call by only owner to unpause the contract
     */
    function unpause() external onlyOwner {
        _unpause();
    }
```

Listing 3.4: `MetroGalaxyMarketplace`

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that if current contracts need to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract.

Status The discussion with the team has confirmed that the ownership will be transferred to the multi-sig account with time-lock executions. And they will consider to transfer the ownership to DAO to improve the decentralization and security of the system.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `MetroGalaxy` protocol, which is a metaverse project that uniquely blends a social platform together. It has its own marketplace and token, and users can trade their own `Metronions` or accessories in the marketplace. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.