SMART CONTRACT AUDIT REPORT

for

Mint Blockchain

Prepared By: Xiaomi Huang

PeckShield

March 4, 2024

Document Properties

Client	Mint Blockchain
Title	Smart Contract Audit Report
Target	MintID/MintGenesisNFT
Version	1.1
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	March 4, 2024	Xuxian Jiang	Post-Final Release #1
1.0	January 15, 2024	Xuxian Jiang	Final Release
1.0-rc	January 14, 2024	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the source code of the MintID/MintGenesisNFT smart contracts, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract exhibits no ERC721 compliance issues or security concerns. This document outlines our audit results.

1.1 About Mint Blockchain

Mint Blockchain is dedicated to advancing innovation in NFT standards and the mass adoption of NFT assets in real-world business scenarios, unleashing the real value of the NFT market. This audit covers the ERC721 compliance of the MintID/MintGenesisNFT token contracts. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of Mint Blockchain

Item	Description
Name	Mint Blockchain
Type	ERC721 Smart Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	March 4, 2024

In the following, we show the Git repositories and the commit hash values used in this audit:

- https://github.com/Mint-Blockchain/Contract-MintID.git (312adde)
- https://github.com/Mint-Blockchain/Contract-MintGenesisNFT.git (2247cc9)

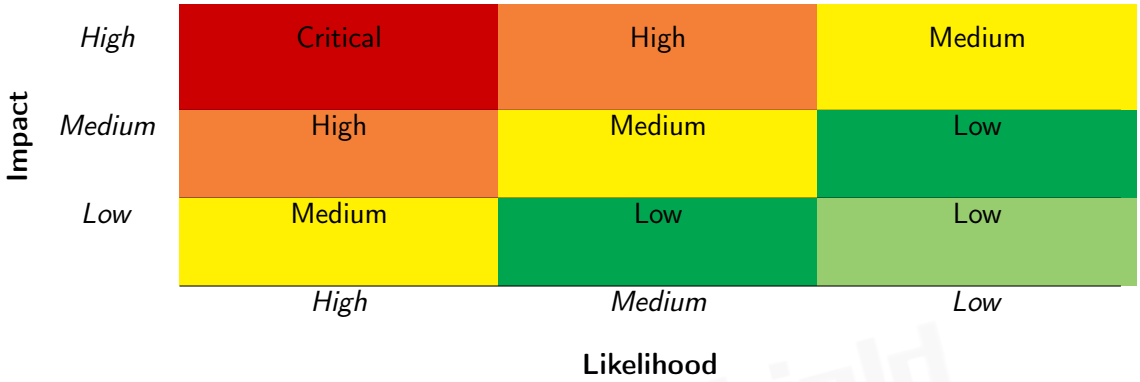
And here come the Git repositories and commit hashes after all fixes for the issues found in the audit have been checked in:

- https://github.com/Mint-Blockchain/Contract-MintID.git (ec49c40)
- https://github.com/Mint-Blockchain/Contract-MintGenesisNFT.git (4c52227)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 classified into four categories accordingly, 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

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
	Approve / TransferFrom Race Condition
ERC721 Compliance Checks	Compliance Checks (Section 3)
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
	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

contract is considered safe regarding the check item. For any discovered issue, we might further 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.
- ERC721 Compliance Checks: We also validate whether the implementation logic of the audited smart contract(s) follows the standard ERC721 specification and other best practices.
- 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.

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 Logic	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 `MintID/MintGenesisNFT` contract design and implementation. 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 ERC721-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	3	■■■
Informational	1	■
Total	4	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC721 specification and other known best practices, and validate its compatibility with other similar ERC721 tokens and current DeFi protocols. The detailed ERC721 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC721 compliance issue was found and our detailed checklist can be found in Section 3. Note that the smart contract implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Audit Findings of MintID/MintGenesisNFT

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Contract Construction/Init. Logic in MintID	Coding Practices	Resolved
PVE-002	Low	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Informational	Redundant Code Removal in MintID	Coding Practices	Resolved
PVE-004	Low	Improved supportsInterface() Logic in MintID	Business Logic	Resolved

In the meantime, we also need to emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks.

3 | ERC721 Compliance Checks

The ERC721 standard for non-fungible tokens, also known as deeds. Inspired by the ERC-20 token standard, the ERC721 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC721-compliant. Naturally, we examine the list of necessary API functions defined by the ERC721 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC721 Specification

Item	Description	Status
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
ownerOf()	Is declared as a public view function	✓
	Returns the address of the owner of the NFT	✓
getApproved()	Is declared as a public view function	✓
	Reverts while '_tokenId' does not exist	✓
	Returns the approved address for this NFT	✓
isApprovedForAll()	Is declared as a public view function	✓
	Returns a boolean value which check '_operator' is an approved operator	✓

Our analysis shows that there is no ERC721 inconsistency or incompatibility issue found in the audited `MintID/MintGenesisNFT` token contracts. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC721 specification.

Table 3.2: Key `State-Changing` Functions Defined in The ERC721 Specification

Item	Description	Status
safeTransferFrom()	Is declared as a public function	✓
	Reverts while 'to' refers to a smart contract and not implement IERC721Receiver-onERC721Received	✓
	Reverts unless 'msg.sender' is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while '_tokenId' is not a valid NFT	✓
	Reverts while '_from' is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
transferFrom()	Is declared as a public function	✓
	Reverts unless 'msg.sender' is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while '_tokenId' is not a valid NFT	✓
	Reverts while '_from' is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
approve()	Is declared as a public function	✓
	Reverts unless 'msg.sender' is the current owner, an authorized operator, or the approved address for this NFT	✓
	Emits Approval() event when tokens are approved successfully	✓
setApprovalForAll()	Is declared as a public function	✓
	Reverts while not approving to caller	✓
	Emits ApprovalForAll() event when tokens are approved successfully	✓
Transfer() event	Is emitted when tokens are transferred	✓
Approval() event	Is emitted on any successful call to approve()	✓
ApprovalForAll() event	Is emitted on any successful call to setApprovalForAll()	✓

4 Detailed Results

4.1 Improved Contract Construction And Initialization Logic in MintID

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MintID`, `MintGenesisNFT`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Both `MintID` and `MintGenesisNFT` contracts follow the `ERC721` specification. In addition, to allow for future upgradeability, these token contracts are deployed as proxies. While examining the related contract construction and initialization logic, we notice current implementation can be improved.

In the following, we use the `MintID` contract as an example and shows its constructor and related initialization routine. It comes to our attention that the constructor can be improved by adding the following statement, i.e., `_disableInitializers();`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initializer()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
45      constructor() {}

47      function initialize(
48          address _address
49      ) public initializerERC721A initializer {
50          __ERC721A_init("MintID", "MintID");
51          __UUPSUpgradeable_init();
52          __Pausable_init();
53          __Ownable_init(_msgSender());
```

```
55          royalty = 50;
56          treasuryAddress = address(_address);
57      }
```

Listing 4.1: MintID::constructor()/initialize()

In addition, the above initialize() routine can also be improved by calling __ReentrancyGuard_init(). The reason is that this token contract is also inherited from the ReentrancyGuardUpgradeable contract and there is a need to call the parent's initialization routine.

Recommendation Improve the above-mentioned constructor and initialization routines.

Status The issue has been fixed by the following commits: 7e4ce4c and 4c52227

4.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MintID, MintGenesisNFT
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the audited token contracts, there exist a privileged owner account that plays important roles in governing and regulating the contract-wide operations. In the following, we examine this privileged account and the related privileged accesses in current contract.

In particular, the privileged functions in the MintID contract allows for the owner to configure the token sale price, extract the sale proceeds, as well as upgrade the token contract.

```
111     function setMintConfig(
112         uint64 _price,
113         uint32 _startTime,
114         uint32 _endTime
115     ) external onlyOwner {
116         require(_endTime > _startTime, "MP: MUST(end time  > Start time)");
117         mintConfig = MintConfig(_price, _startTime, _endTime);
118     }
119
120     function setRoyalty(uint256 _royalty) external onlyOwner {
121         require(
122             _royalty <= 100 && _royalty >= 0,
123             "MP: Royalty can only be between 0 and 10%"
124         );
125         royalty = _royalty;
126     }
```

```
127
128     function setTreasuryAddress(address _addr) external onlyOwner {
129         require(_addr != address(0x0), "MP: Address not be zero");
130         treasuryAddress = _addr;
131     }
132
133     function withdraw() external onlyOwner nonReentrant {
134         require(
135             treasuryAddress != address(0x0),
136             "MP: Must set withdrawal address"
137         );
138         (bool success, ) = treasuryAddress.call{value: address(this).balance}(
139             ""
140         );
141         require(success, "MP: Transfer failed");
142     }
143
144     function supportsInterface(
145         bytes4 interfaceId
146     ) public view override(ERC721AUpgradeable, IERC165) returns (bool) {
147         return ERC721AUpgradeable.supportsInterface(interfaceId);
148     }
149
150     function _authorizeUpgrade(
151         address newImplementation
152     ) internal override onlyOwner {}
```

Listing 4.2: Privileged Operations in GoArtSBT

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to these privileged accounts explicit to the token users.

Status This issue has been mitigated with a multi-sig account to manage the admin key.

4.3 Redundant Code Removal in MintID

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MintID`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `MintID` token contract makes good use of a number of reference contracts, such as `OwnableUpgradeable`, `PausableUpgradeable`, `ReentrancyGuardUpgradeable`, and `UUPSUpgradeable`, to facilitate its code implementation and organization. For example, the `MintID` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `MintID` contract, there is an inherited parent contract, i.e., `ReentrancyGuardUpgradeable`, which is not functionally necessary. In particular, it is only used in the `withdraw()` routine, which has no security risk if we remove this modifier completely.

```
133     function withdraw() external onlyOwner nonReentrant {
134         require(
135             treasuryAddress != address(0x0),
136             "MP: Must set withdrawal address"
137         );
138         (bool success, ) = treasuryAddress.call{value: address(this).balance}(
139             ""
140         );
141         require(success, "MP: Transfer failed");
142     }
```

Listing 4.3: `ReentrancyGuardUpgradeable::withdraw()`

Recommendation Consider the removal of the redundant modifier with a simplified, consistent implementation.

Status The issue has been fixed by the following commit: `7e4ce4c`.

4.4 Improved supportsInterface() Logic in MintID

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MintID`
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [4]

Description

The `MintID` contract follows the `ERC2981` specification by defining a standardized way to retrieve royalty payment information for non-fungible tokens (NFTs) to enable universal support for royalty payments across all `NFT` marketplaces and ecosystem participants. While reviewing the exported functions in `MintID`, we notice the `supportsInterface()` function can be improved.

To elaborate, we show below the implementation of this `supportsInterface()` function. This function takes an interface identifier as an argument and returns a `Boolean` value indicating whether the contract supports that interface. The interface identifier is a unique 32-byte hash generated from the interface's function signatures. Since `MintID` contract supports `ERC2981`, there is a need to revise the `supportsInterface()` function to include the `ERC2981` interface support, i.e., `ERC721AUpgradeable.supportsInterface(interfaceId)|| interfaceId == type(IERC2981).interfaceId`.

```
151     function supportsInterface(
152         bytes4 interfaceId
153     ) public view override(ERC721AUpgradeable, IERC165) returns (bool) {
154         return ERC721AUpgradeable.supportsInterface(interfaceId);
155     }
```

Listing 4.4: `MintID::supportsInterface()`

Recommendation Revise the above function to include the `IERC2981` interface support. An example revision is shown as below:

```
151     function supportsInterface(
152         bytes4 interfaceId
153     ) public view override(ERC721AUpgradeable, IERC165) returns (bool) {
154         return ERC721AUpgradeable.supportsInterface(interfaceId)  interfaceId == type(
                IERC2981).interfaceId;
155     }
```

Listing 4.5: Revised `MintID::supportsInterface()`

Status The issue has been fixed by the following commit: `ec49c40`.

5 Conclusion

In this security audit, we have examined the `MintID/MintGenesisNFT` contract design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC721 specification and other known ERC721 pitfalls/vulnerabilities and found no issue in these areas. We then proceeded to examine other areas such as coding practices and business logics. Overall, we found one low-severity issue and three informational recommendations which are promptly addressed by the team. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

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

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.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.