SMART CONTRACT AUDIT REPORT

for

ArgonAtomicCollection

Prepared By: Yiqun Chen

PeckShield

September 24, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 24, 2021	Xuxian Jiang	Final Release

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 smart contract source code of **ArgonAtomicCollection**, 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 Argon

`Argon` is an innovative blockchain-based freelancer platform on the `Binance Smart Chain (BSC)`. It is designed to work with smart contracts and fully decentralized. The audited `ArgonAtomicCollection` is a smart contract that allows for not only atomic collection of `non-fungible tokens (NFTs)`, but also the minting of a variety of customized `NFTs`.

The basic information of ArgonAtomicCollection is as follows:

Table 1.1: Basic Information of ArgonAtomicCollection

Item	Description
Name	Argon
Website	https://argon.run/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 24, 2021

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

- https://github.com/Argon-Foundation/argon-atomic-collection-nft.git (e8c304c)

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

- https://github.com/Argon-Foundation/argon-atomic-collection-nft.git (48bc86e)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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

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

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.
- 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) [6], 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 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 `ArgonAtomicCollection` 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

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, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Conflicts Between sendMint() and mint()	Business Logic	Fixed
PVE-002	Informational	Suggested Constant/Immutable Usages For Gas Efficiency	Coding Practices	Confirmed
PVE-003	Low	Unused State/Code Removal	Coding Practices	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 Possible Conflicts Between sendMint() and mint()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: ArgonAtomicCollection
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

The ArgonAtomicCollection smart contract allows for not only atomic collection of NFTs, but also the minting of a variety of customized NFTs. While examining the current mint logic, we notice certain conflicts in existing implementation.

To elaborate, we show below two related functions: sendMint() and mint(). The second function properly charges the mint fee (line 106) while the first function avoids paying any mint fee. Moreover, the first function allows for the user specify any tokenId that is intended to be minted, while the second function only mints an available one (line 112). These conflicts suggest the first function is included for debugging purpose and should be removed before production.

```
    function sendMint(uint256 id) public {
        _mint(msg.sender, id);
    }
```

Listing 3.1: ArgonAtomicCollection::sendMint()

```
104     function mint() external payable nonReentrant mustNotPaused {
105         require(remaining.length > 0, "there is no nft");
106         require(msg.value >= price);
107         uint256 newItemIndex = SafeMath.mod(
108             generateRandom(),
109             remaining.length,
110             "SafeMath: error"
111         );
112         uint256 newItemId = remaining[newItemIndex];
```

```
        // require(msg.value >= price);
        _mint(msg.sender, newItemId);
        _setTokenURI(
            newItemId,
            string(
                abi.encodePacked(baseURI, Strings.toString(newItemId), ".json")
            )
        );

        remaining[newItemIndex] = remaining[remaining.length - 1];
        remaining.pop();
        tokenCount.increment();
        feeAddress.transfer(msg.value);
    }
```

Listing 3.2: `ArgonAtomicCollection::mint()`

Recommendation Remove the first function `sendMint()` as it causes unnecessary conflicts.

Status This issue has been fixed in the following commit: `48bc86e`.

3.2 Suggested Constant/Immutable Usages For Gas Efficiency

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ArgonAtomicCollection`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show a number of key state variables defined in `ArgonAtomicCollection`, including `price` and `feeAddress`. If there is no need to dynamically update these key state variables, they can be declared as either constants or `immutable` for gas efficiency. In particular, `price` can be declared as `constant` while `feeAddress` can be defined as `immutable`.

```
    uint256 public price = 2 ether;
    uint256[] public remaining = [1];
    string public baseURI;
    bool public paused;
    address payable public feeAddress;
```

Listing 3.3: ArgonAtomicCollection.sol

Recommendation Revisit the state variable definition and make extensive use of `constant`/`immutable` states.

Status The issue has been confirmed.

3.3 Unused State/Code Removal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ArgonAtomicCollection
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [2]

Description

The `ArgonAtomicCollection` smart contract makes good use of a number of reference contracts, such as `ERC721`, `Ownable`, `SafeMath`, and `ReentrancyGuard`, to facilitate its code implementation and organization. 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 current contract implementation, there are a number of functions that are defined, but not used. Examples include the `remove()` and `transferFromTokens()` functions. Both functions are defined as private, but are not called anywhere in current contract. Therefore, they can be safely removed.

```
    function remove(uint256 index) private returns (uint256[] memory) {
        for (uint256 i = index; i < remaining.length - 1; i++) {
            remaining[i] = remaining[i + 1];
        }
        remaining.pop();
        return remaining;
    }
```

Listing 3.4: ArgonAtomicCollection::remove()

```
    function transferFromTokens(uint256[] memory tokenIDs)
        private
        nonReentrant
    {
        for (uint256 i = 0; i < tokenIDs.length; i++) {
            safeTransferFrom(msg.sender, address(this), tokenIDs[i]);
        }
    }
```

Listing 3.5: ArgonAtomicCollection::transferFromTokens()

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been confirmed.

4 | Conclusion

In this audit, we have analyzed the documentation and implementation of `ArgonAtomicCollection`. The audited system is a smart contract that allows for not only atomic collection of `non-fungible tokens (NFTs)`, but also the minting of a variety of customized `NFTs`. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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