SMART CONTRACT AUDIT REPORT

for

Asteriod X

Prepared By: Xiaomi Huang

PeckShield

October 23, 2024

Document Properties

Client	Asteriod X
Title	Smart Contract Audit Report
Target	Asteriod X
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 23, 2024	Xuxian Jiang	Final Release
1.0-rc	October 20, 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 design document and related smart contract source code of the Asteriod X launchpad, 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 Asteriod X

Asteriod X is an innovative mining investment platform that connects mining companies with global investors through blockchain technology. All NFTs on the platform adhere to the ERC-1155 standard, ensuring versatility and efficient transactions. Asteroid X offers a transparent trading mechanism and high-quality investment opportunities, creating a win-win scenario for mining companies and crypto investors alike. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	Asteriod X
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 23, 2024

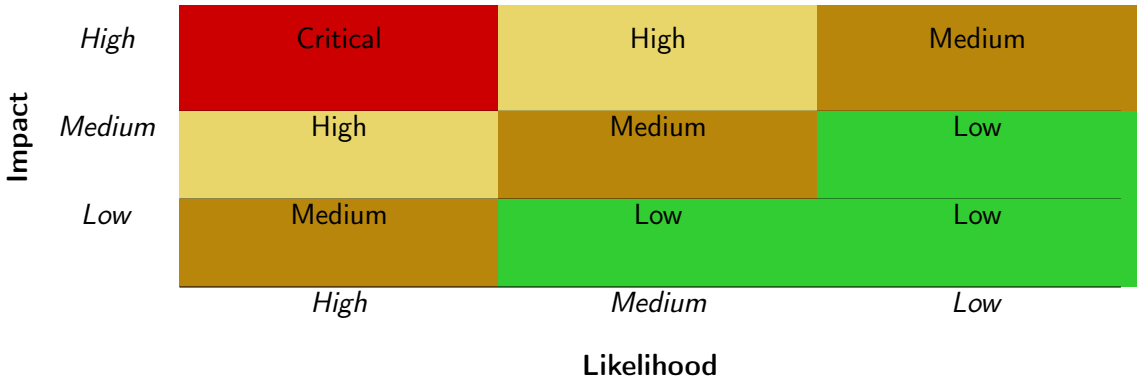
In the following, we show the audited contract code deployed at the following addresses:

- ERC1155Asteroid: https://sepolia.etherscan.io/address/0xa667b0dbB6A88982A98B8a6f85Cfb39586BF29c8
- LaunchPadAsteroid: https://sepolia.etherscan.io/address/0x536c6ad7808608742a3f84f5B0D55B715C3F4fB5

1.2 About PeckShield

PeckShield Inc. [9] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 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

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

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) [7], 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 implementation of the `Asteriod X` launchpad. 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 logic, 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	0	
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 that 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 and 2 low-severity vulnerabilities.

Table 2.1: Key Asteriod X Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation of Function Parameters	Coding Practices	Resolved
PVE-002	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Improved Validation of Function Parameters

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC1155Asteroid`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The audited launchpad contract is no exception. Specifically, if we examine the `ERC1155Asteroid` contract, it has defined a number of `NFT`-specific risk parameters, such as `tokenMinAmounts` and `tokenMaxAmounts`. In the following, we show the corresponding routine that initializes their values.

```
227     function validateParameters(
228         uint256 _id,
229         uint256 _initialSupply,
230         uint256 _initialRaisedAmounts,
231         uint256 _initialMinAmount,
232         uint256 _initialMaxAmount,
233         address _initialFundsWallet
234     ) internal view {
235         require(!_exists(_id), "token _id already exists");
236         require(
237             _initialFundsWallet != address(0),
238             "Invalid initialFunds Wallet"
239         );
240         require(
241             _initialSupply > 0 &&
242                 _initialRaisedAmounts > 0 &&
243                 _initialMinAmount > 0 &&
244                 _initialMaxAmount > 0,
245             "Invalid amount"
```

```
246         );
247         require(
248             _initialRaisedAmounts >= _initialSupply,
249             "Invalid funds and shares raised"
250         );
251         uint256 perShares = perShareValue(
252             _initialRaisedAmounts,
253             _initialSupply
254         );
255         require(_initialMinAmount >= perShares, "Invalid shares");
256     }
```

Listing 3.1: ERC1155Asteroid::validateParameters()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these 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, the above routine can be improved by enforcing the following requirement, i.e., `require(_initialMaxAmount >= _initialMinAmount)` (line 244).

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

Status The issue has been resolved by following the above suggestion.

3.2 Improved Precision By Multiplication And Division Reordering

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: ERC1155Asteroid
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [2]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (mul) and division (div) are involved.

In particular, we use the `ERC1155Asteroid::calculQuantities()` as an example. This routine is used to calculate the resulting quantity of a particular purchase.

```
308     function calculQuantities(uint256 id, uint256 amount)
309         internal
310         view
311         returns (uint256)
312     {
313         uint256 totalAmounts = tokenRaisedAmounts[id];
314         uint256 supply = tokenSupply[id];
315         (, uint256 perSV) = Math.tryDiv(totalAmounts, supply);
316         (, uint256 quantity) = Math.tryDiv(amount, perSV);
317         return quantity;
318     }
```

Listing 3.2: `ERC1155Asteroid::calculQuantities()`

We notice the calculation of the resulting quantity (line 317) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `amount * totalAmounts / supply` . Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been resolved by following the above suggestion.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

Description

The audited launchpad contract has a privileged account (either `owner` or the account with the `DEFAULT_ADMIN_ROLE` role) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters, assign other roles, and recover stuck funds). In the following, we show the representative functions potentially affected by the privilege of this account.

```
96      function setPaymentCoin(IERC20Metadata _iERC20Metadata) external onlyOwner {
97          require(
98              paymentCoin != _iERC20Metadata,
99              "LaunchPadAsteroid#setERC1155AsteroidContract: PaymentCoin has been already
                    configured"
```

```
100         );
101         paymentCoin = _iERC20Metadata;
102     }
103
104     /**
105      * @dev set to asteroid contract
106      */
107     function setERC1155AsteroidContract(IERC1155Asteroid _iERC1155Asteroid)
108         external
109         onlyOwner
110     {
111         require(
112             iERC1155Asteroid != _iERC1155Asteroid,
113             "LaunchPadAsteroid#setERC1155AsteroidContract: ERC1155Asteroid has been
                already configured"
114         );
115         iERC1155Asteroid = _iERC1155Asteroid;
116     }
117
118     function pause() external onlyOwner {
119         _pause();
120     }
121
122     function unpause() external onlyOwner {
123         _unpause();
124     }
125
126     /**
127      * @dev If this contract has other funds, emergency withdrawal
128      */
129     function withdraw(
130         address to,
131         address erc20Address,
132         uint256 amount
133     ) external onlyOwner {
134         uint256 balance = IERC20Metadata(erc20Address).balanceOf(address(this));
135         require(balance >= amount, "Insufficient balance!");
136         IERC20Metadata(erc20Address).safeTransfer(to, amount);
137     }
```

Listing 3.3: Example Privileged Operations in `LaunchPadAsteroid`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive vault parameters, which directly undermines the assumption of the vault design.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the in-

tended trustless nature and high-quality distributed governance.

Status The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the launchpad in Asteriod X, which is an innovative mining investment platform that connects mining companies with global investors through blockchain technology. All NFTs on the platform adhere to the ERC-1155 standard, ensuring versatility and efficient transactions. Asteroid X offers a transparent trading mechanism and high-quality investment opportunities, creating a win-win scenario for mining companies and crypto investors alike. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

[6] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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