SMART CONTRACT AUDIT REPORT

for

Illuvium Land Sale

Prepared By: Xiaomi Huang

PeckShield

May 3, 2022

Document Properties

Client	Illuvium Land Sale
Title	Smart Contract Audit Report
Target	Illuvium Land Sale
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 3, 2022	Xuxian Jiang	Final Release
1.0-rc	May 1, 2022	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 Illuvium Land Sale 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 audited protocol 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 Illuvium Land Sale

Illuvium Land Sale, also known as the Land Sale Protocol, or Land Sale, powers the series of sales (a.k.a sale events) with selling in total up to around $100,000$ Land NFTs. The protocol operates either completely on Layer 1, or in a mixed Layer 1/2 mode (when sale happens in Layer 1 and NFT minting happens in Layer 2), and consists of a Land ERC721 token itself, sale supporting smart contracts (helpers), backend and frontend services powering the initial sale of the token and simplifying the interaction with it later on. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Illuvium Land Sale

Item	Description
Name	Illuvium Land Sale
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 3, 2022

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

- https://github.com/IlluviumGame/land-sale-core.git (b5d0bce)

- https://github.com/IlluviumGame/land-sale-periphery.git (ef3f06a)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/IlluviumGame/land-sale-core.git (5f58aa5)
- https://github.com/IlluviumGame/land-sale-periphery.git (ef3f06a)

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

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 [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 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 1.3: 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

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 `Illuvium Land Sale` 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 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	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 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, 2 low-severity vulnerabilities, and 1 informational recommendation,

Table 2.1: Key Illuvium Land Sale Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-002	Low	Improved Validation in RoyalERC721::setRoyaltyInfo()	Coding Practices	Resolved
PVE-003	Informational	Improved Resumed Event Generation in initialize()	Coding Practices	Resolved
PVE-004	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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the transfer() routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the transfer() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In current implementation, if we examine the `LandSale::rescueErc20()` routine that is designed to rescue accidentally `ERC20` tokens sent to the current contract. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 818).

```
809     function rescueErc20(address _contract, address _to, uint256 _value) public virtual
            {
810         // verify the access permission
811         require(isSenderInRole(ROLE_RESCUE_MANAGER), "access denied");

813         // verify rescue manager is not trying to withdraw sILV:
814         // we have a withdrawal manager to help with that
815         require(_contract != sIlvContract, "sILV access denied");

817         // perform the transfer as requested, without any checks
818         require(ERC20(_contract).transfer(_to, _value), "ERC20 transfer failed");
819     }
```

Listing 3.2: `LandSale::rescueErc20()`

In the meantime, we also suggest to use the safe-version of `transfer()` in other related contracts, including `ERC721Impl` and `UpgradeableERC721`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `08b072b`.

3.2 Improved Validation in RoyalERC721::setRoyaltyInfo()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RoyalERC721`
- 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 `Land Sale` protocol is no exception. Specifically, if we examine the `RoyalERC721` contract, it has defined a number of protocol-wide risk parameters, such as `royaltyReceiver` and `royaltyPercentage`. In the following, we show the corresponding routines that allow for their changes.

```
    function setRoyaltyInfo(address _royaltyReceiver, uint16 _royaltyPercentage) public
        virtual {
        // verify the access permission
        require(isSenderInRole(ROLE_ROYALTY_MANAGER), "access denied");

        // verify royalty percentage is zero if receiver is also zero
        require(_royaltyReceiver != address(0)  _royaltyPercentage == 0, "invalid
            receiver");

        // update the values
        royaltyReceiver = _royaltyReceiver;
        royaltyPercentage = _royaltyPercentage;

        // emit an event
        emit RoyaltyInfoUpdated(msg.sender, _royaltyReceiver, _royaltyPercentage);

    }
```

Listing 3.3: `RoyalERC721::setRoyaltyInfo()`

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, an unlikely mis-configuration of `royaltyPercentage` may charge unreasonably high royalty fee in the payment, hence incurring cost to borrowers or hurting the adoption of the protocol. With that, we suggest to add the following requirement: `require( _royaltyPercentage <= 100_00 )`.

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: `5f1a96d`.

3.3 Improved Resumed Event Generation in initialize()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LandSale
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the LandSale contract as an example. This contract has the initialize() function to set up sale parameters, all at once, or any subset of them. This function also emits two events Initialized and Resumed. The second event is used to indicate the resumption from a previously paused state. It comes to our attention that the second event can be improved by properly providing the pauseDuration information, which is currently simply set to 0 (line 590). For improvement, we need to properly emit the Resumed event with the right pauseDuration state and then reset the pauseDuration state to 0.

```
563     function initialize(
564         uint32 _saleStart,          // <<<--- keep type in sync with the body type(uint32).max !!!
565         uint32 _saleEnd,            // <<<--- keep type in sync with the body type(uint32).max !!!
566         uint32 _halvingTime,        // <<<--- keep type in sync with the body type(uint32).max !!!
567         uint32 _timeFlowQuantum,    // <<<--- keep type in sync with the body type(uint32).max !!!
568         uint32 _seqDuration,        // <<<--- keep type in sync with the body type(uint32).max !!!
569         uint32 _seqOffset,          // <<<--- keep type in sync with the body type(uint32).max !!!
570         uint96[] memory _startPrices // <<<--- keep type in sync with the body type(uint96).max !!!
571     ) public virtual {
572         // verify the access permission
573         require(isSenderInRole(ROLE_SALE_MANAGER), "access denied");

575         // Note: no input validation at this stage, initial params state is invalid anyway,
```

```
        //        and we're not limiting sale manager to set these params back to this
            state

        // set/update sale parameters (allowing partial update)
        // 0xFFFFFFFF, 32 bits
        if(_saleStart != type(uint32).max) {
            // update the sale start itself, and
            saleStart = _saleStart;

            // erase the cumulative pause duration
            pauseDuration = 0;

            // if the sale is in paused state (non-zero `pausedAt`)
            if(pausedAt != 0) {
                // emit an event first - to log old `pausedAt` value
                emit Resumed(msg.sender, pausedAt, now32(), 0);

                // erase `pausedAt`, effectively resuming the sale
                pausedAt = 0;
            }
        }
        // 0xFFFFFFFF, 32 bits
        if(_saleEnd != type(uint32).max) {
            saleEnd = _saleEnd;
        }
        // 0xFFFFFFFF, 32 bits
        if(_halvingTime != type(uint32).max) {
            halvingTime = _halvingTime;
        }
        // 0xFFFFFFFF, 32 bits
        if(_timeFlowQuantum != type(uint32).max) {
            timeFlowQuantum = _timeFlowQuantum;
        }
        // 0xFFFFFFFF, 32 bits
        if(_seqDuration != type(uint32).max) {
            seqDuration = _seqDuration;
        }
        // 0xFFFFFFFF, 32 bits
        if(_seqOffset != type(uint32).max) {
            seqOffset = _seqOffset;
        }
        // 0xFFFFFFFFFFFFFFFFFFFFFFFF, 96 bits
        if(_startPrices.length != 1  _startPrices[0] != type(uint96).max) {
            startPrices = _startPrices;
        }

        // emit an event
        emit Initialized(msg.sender, saleStart, saleEnd, halvingTime, timeFlowQuantum,
            seqDuration, seqOffset, startPrices);
    }
```

Listing 3.4: `LandSale::initialize()`

Recommendation Properly emit the Resumed event when the initialize() routine resumes from a previously paused state.

Status The issue has been fixed by this commit: 5f58aa5.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Illuvium Land Sale protocol, there are special administrative accounts (with the ROLE_ACCESS_MANAGER role). These accounts play a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters and execute privileged operations). They also have the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine their related privileged accesses in current protocol.

```
    function pause() public virtual {
        // check the access permission
        require(isSenderInRole(ROLE_PAUSE_MANAGER), "access denied");

        // check if sale is not in the paused state already
        require(pausedAt == 0, "already paused");

        // do the pause, save the paused timestamp
        // note for tests: never set time to zero in tests
        pausedAt = now32();

        // emit an event
        emit Paused(msg.sender, now32());
    }

    function setBeneficiary(address payable _beneficiary) public virtual {
        // check the access permission
        require(isSenderInRole(ROLE_WITHDRAWAL_MANAGER), "access denied");

        // update the beneficiary address
        beneficiary = _beneficiary;

        // emit an event
        emit BeneficiaryUpdated(msg.sender, _beneficiary);
```

```
    }
```

Listing 3.5: Example Privileged Operations in `LandSale`

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 Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team clarifies the use of the `Illuvium eDAO MultiSig` wallet which requires 4/6 signatures.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Illuvium Land Sale` protocol, which powers the series of sales with selling in total up to around $100,000$ `Land NFTs`. The protocol operates either completely on `Layer 1`, or in a mixed `Layer 1/2` mode (when sale happens in `Layer 1` and `NFT` minting happens in `Layer 2`), and consists of a `Land ERC721` token itself, sale supporting smart contracts (helpers), backend and frontend services powering the initial sale of the token and simplifying the interaction with it later on. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.