



SMART CONTRACT AUDIT REPORT

for

Lucidao Protocol



Prepared By: Yiqun Chen

PeckShield
January 7, 2022

Document Properties

Client	Lucidao
Title	Smart Contract Audit Report
Target	Lucidao
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 7, 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	4
1.1	About Lucidao	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Trust Issue of Admin Keys	11
3.2	Minimum Delay Bypass In TimelockController	12
3.3	Possible Double Initialization From Initializer Reentrancy	14
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Lucidao 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 Lucidao

Lucidao is a decentralized autonomous organization which aims to bridge the crypto and real worlds by connecting crypto communities, traditional business owners (oracles), the technology, and the liquidity provided by the cryptocurrency and DeFi market. The team aims to create an open and community-governed multi-sided platform covering various industries and target groups to address diverse needs. The Lucidao is based on the OpenZeppelin Governance protocol which allows Lucidao members to propose and vote platform upgrades.

The basic information of Lucidao is as follows:

Table 1.1: Basic Information of Lucidao

Item	Description
Name	Lucidao
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 7, 2022

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

- <https://github.com/lucidao-developer/lucidao-smart-contracts.git> (4536355)

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

- <https://github.com/lucidao-developer/lucidao-smart-contracts.git> (4e02aa6)

1.2 About PeckShield

PeckShield Inc. [7] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

To evaluate the risk, we go through a checklist of 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) [5], 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 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 implementation of the `Lucidao` 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 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	2	
Low	1	
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 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 2 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Lucidao Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-002	Low	Possible Double Initialization From Initializer Reentrancy	Time and State	Fixed
PVE-003	Medium	Minimum Delay Bypass in Timelock-Controller	Time and State	Fixed

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 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: LucidaoGovernanceReserve
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `LucidaoGovernanceReserve` contract, 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., increase/decrease allowance for any accounts).

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

```

31     function approveToken(IERC20 token, uint256 weiAmount, address spender) external
        onlyOwner {
32         token.safeApprove(spender, weiAmount);
33     }

35     function increaseAllowanceToken(IERC20 token, uint256 weiAmount, address spender)
        external onlyOwner {
36         token.safeIncreaseAllowance(spender, weiAmount);
37     }

39     function decreaseAllowanceToken(IERC20 token, uint256 weiAmount, address spender)
        external onlyOwner {
40         token.safeDecreaseAllowance(spender, weiAmount);
41     }

```

Listing 3.1: `LucidaoGovernanceReserve`

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 This issue has been confirmed.

3.2 Minimum Delay Bypass In TimelockController

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `TimelockController`
- Category: Time and State [4]
- CWE subcategory: CWE-682 [2]

Description

The `TimelockController`, introduced in `OpenZeppelin Contracts 3.3`, is a smart contract that enforces a delay on all actions directed towards an owned contract. A typical setup is to position the `TimelockController` as the admin of an application smart contract such that, whenever a privileged action is to be executed, it has to wait for a certain time specified by the `TimelockController`.

The security benefits of the `TimelockController` are twofold. Firstly, it provides an extra layer of security to a project's team by giving a heads up on every privileged action anticipated in the system. This allows the team to detect and react to malicious calls by compromised admin accounts. Secondly, it protects the community from the project's governance itself, allowing members to exit the protocol if they disagree with any impending changes.

In particular, `scheduleBatch()` (a schedule function) and `executeBatch()` (an execute function), allow the caller to enqueue and execute proposals that run multiple calls in sequence. However, there is a vulnerability in their implementation that can be exploited by the malicious `EXECUTOR` to execute arbitrary tasks bypassing the minimum delay protection.

To elaborate, we show below the related code snippet of the `TimelockController` contract. A malicious `EXECUTOR` could execute a batch with the calling of `executeBatch()`, including a set of calls, i.e., the call to the `TimelockController` itself to clear the minimum delay and grant `PROPOSER` and `ADMIN` rights to an address under their control, the call to `scheduleBatch()` to enqueue the batch by the

controlled PROPOSER and the call to the arbitrary privileged functions under the TimelockController control. By doing so, the malicious EXECUTOR effectively takes full control of the TimelockController contract.

```

318     function executeBatch(
319         address[] calldata targets,
320         uint256[] calldata values,
321         bytes[] calldata datas,
322         bytes32 predecessor,
323         bytes32 salt
324     ) external payable virtual onlyRole(EXECUTOR_ROLE) {
325         require(
326             targets.length == values.length,
327             "TimelockController: length mismatch"
328         );
329         require(
330             targets.length == datas.length,
331             "TimelockController: length mismatch"
332         );
333
334         bytes32 id =
335             hashOperationBatch(targets, values, datas, predecessor, salt);
336         _beforeCall(predecessor);
337         for (uint256 i = 0; i < targets.length; ++i) {
338             _call(id, i, targets[i], values[i], datas[i]);
339         }
340         _afterCall(id);
341     }

```

Listing 3.2: TimelockController::executeBatch()

```

224     function scheduleBatch(
225         address[] calldata targets,
226         uint256[] calldata values,
227         bytes[] calldata datas,
228         bytes32 predecessor,
229         bytes32 salt,
230         uint256 delay
231     ) external virtual onlyRole(PROPOSER_ROLE) {
232         require(
233             targets.length == values.length,
234             "TimelockController: length mismatch"
235         );
236         require(
237             targets.length == datas.length,
238             "TimelockController: length mismatch"
239         );
240
241         bytes32 id =
242             hashOperationBatch(targets, values, datas, predecessor, salt);
243         _schedule(id, delay);
244         for (uint256 i = 0; i < targets.length; ++i) {

```

```
245         emit CallScheduled(  
246             id,  
247             i,  
248             targets[i],  
249             values[i],  
250             datas[i],  
251             predecessor,  
252             delay  
253         );  
254     }  
255 }
```

Listing 3.3: `TimelockController::scheduleBatch()`

Recommendation Considering the `OpenZeppelin` team has solved this vulnerability, we suggest to upgrade the `TimelockController` to the latest version.

Status The issue has been addressed in this commit: [92942b9](#).

3.3 Possible Double Initialization From Initializer Reentrancy

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LucidaoGovernor`
- Category: Time and State [\[4\]](#)
- CWE subcategory: CWE-682 [\[2\]](#)

Description

The `LucidaoGovernor` contract supports flexible contract initialization, so that the initialization task does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the `initializer()` modifier that protects an initializer function from being invoked twice. It becomes known that the popular `OpenZeppelin` reference implementation has an issue that makes it possible to re-enter `initializer()`-protected functions. In particular, for this to happen, one call may need to be a nested-call of the other, or both calls have to be subcalls of a common `initializer()`-protected function.

The reentrancy can be dangerous as the initialization is not part of the proxy construction, and it becomes possible by executing an external call to an untrusted address. As part of the fix, there is a need to forbid `initializer()`-protected functions to be nested when the contract is already constructed.

To elaborate, we show below the current `initializer()` implementation as well as the fixed implementation.

```

37     modifier initializer() {
38         require(!_initializing? _isConstructor() : !_initialized, "Initializable: contract
           is already initialized");
39
40         bool isTopLevelCall = !_initializing;
41         if (isTopLevelCall) {
42             _initializing = true;
43             _initialized = true;
44         }
45
46         _;
47
48         if (isTopLevelCall) {
49             _initializing = false;
50         }
51     }

```

Listing 3.4: Initializable::initializer()

```

37     modifier initializer() {
38         require(!_initializing? _isConstructor() : !_initialized, "Initializable:
           contract is already initialized");
39
40         bool isTopLevelCall = !_initializing;
41         if (isTopLevelCall) {
42             _initializing = true;
43             _initialized = true;
44         }
45
46         _;
47
48         if (isTopLevelCall) {
49             _initializing = false;
50         }
51     }

```

Listing 3.5: Revised Initializable::initializer()

Recommendation Enforce the `initializer()` modifier to prevent it from being re-entered.

Status This issue has been fixed in the following commit: 92942b9.

4 | Conclusion

In this audit, we have analyzed the `Lucidao` design and implementation. The `Lucidao` protocol is governed and driven by its community members, and allows its members to propose and vote platform upgrades. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, 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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.
- [5] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.