SMART CONTRACT AUDIT REPORT

for

Pandora TimeLockV2

Prepared By: Xiaomi Huang

PeckShield

May 20, 2022

Document Properties

Client	Pandora Protocol
Title	Smart Contract Audit Report
Target	TimeLockV2
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Liu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	May 20, 2022	Xuxian Jiang	Release Candidate #1

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 TimeLockV2 contract in the Pandora 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 Pandora

Pandora recognizes the shortcomings of existing DEXs (in mainly incentivizing farmers without rewards for traders) and proposes an inclusive reward system that offers traders and farmers sustainable income and multiple benefits in an attempt to maintain high user retention rates. By gamifying its protocol, Pandora attracts users and keeps them engaged as well as creates a user-centered decentralized ecosystem where all participants are properly incentivized and empowered to make decisions on governance. This audit focuses on the TimeLockV2 contract for the administrative operations. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of TimeLockV2

Item	Description
Name	Pandora Protocol
Website	https://pandora.digital/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 20, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit only covers the `TimeLockV2` contract.

- https://github.com/PandoraDigital/smart-contract.git (acb0754)

1.2 About PeckShield

PeckShield Inc. [5] 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 the OWASP Risk Rating Methodology [4]:

- 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 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

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

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) [3], 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 `TimeLockV2` contract in the `Pandora` 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	0	
Low	1	■
Informational	1	■
Total	2	

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 low-severity vulnerability and 1 informational recommendation.

Table 2.1: Key TimeLockV2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Proper Role Management in TimeLockV2	Coding Practices	
PVE-002	Low	Lack Of receive() handler in TimeLockV2	Coding Practices	

Besides 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 Proper Role Management in TimeLockV2

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: TimeLockV2
- Category: Coding Practices [2]
- CWE subcategory: CWE-1126 [1]

Description

The TimeLockV2 contract provides the functionality that basically delays function calls of another smart contract after a predetermined amount of time has passed. It is mostly used for various governance tasks. The contract defines a number of roles: PROPOSER_ROLE, EXECUTOR_ROLE, ADMIN_ROLE , and WALLET_ROLE. As their names indicate, the first role allows for the holder to propose a new governance task; the second role is capable of executing the proposed task after it is passed; the third one performs the administrative role; and the last one manages the above three roles.

To elaborate, we show below the related constructor() function from the TimeLockV2 contract. It has properly configured the above roles. It comes to our attention that the explicit role admin for them is not given. To avoid unnecessary confusion and improve readability and maintenance, we suggest to explicitly grant their role admin to WALLET_ROLE!

```
    constructor(address _admin, uint256 _minDelay) {
        _setupRole(PROPOSER_ROLE, _admin);
        _setupRole(EXECUTOR_ROLE, _admin);
        _setupRole(ADMIN_ROLE, _admin);
        _setupRole(WALLET_ROLE, address(this));
        required = 1;
        nAdmins = 1;
        minDelay = _minDelay;
    }
```

Listing 3.1: TimeLockV2::constructor()

Recommendation Revise the above-mentioned constructor() routine to properly set up their role admins.

Status

3.2 Lack Of receive() handler in TimeLockV2

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: TimelockV2
- Category: Coding Practices [2]
- CWE subcategory: CWE-1126 [1]

Description

The TimelockV2 contract comes with essential functions to schedule and execute administrative tasks. While examining the current set of functions, we notice the contract can be benefited by adding a receive() function, which may be needed in scenarios when the contract needs to receive or hold ETH in the administrative process.

To elaborate, we show below the execute() routine that basically transfers Ether from a sender to the contract. In fact, this routine is the only one that can receive Ether. Since the timelocked task supports the use of Ether, it is suggested to implement the alternative receive() function as well. Note the receive() method is used as a fallback function and is called when Ether is sent to a contract with no calldata.

```
    function execute(
        address _target,
        uint256 _value,
        bytes calldata _data,
        bytes32 _predecessor,
        bytes32 _salt
    ) external payable onlyRoleOrOpenRole(EXECUTOR_ROLE) {
        bytes32 _id = _hashOperation(_target, _value, _data, _predecessor, _salt);
        require(!isCanceled[_id], "Timelock: proposer already canceled");
        if (confirmations[_id] >= required) {
            _beforeCall(_id, _predecessor);
            _call(_id, 0, _target, _value, _data);
            _afterCall(_id);
        }
    }
```

Listing 3.2: TimelockV2::execute()

Recommendation Add the support of receive() in TimelockV2.

Status

4 Conclusion

In this audit, we have analyzed the design and implementation of the `TimeLockV2` contract in the `Pandora` protocol, which proposes an inclusive reward system that offers traders and farmers sustainable income and multiple benefits in an attempt to maintain high user retention rates. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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