SMART CONTRACT AUDIT REPORT

for

Plutos VirtualTrade

Prepared By: Yiqun Chen

Hangzhou, China
February 13, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	February 13, 2022	Shulin Bie	Final Release
1.0-rc	February 10, 2022	Shulin Bie	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

Given the opportunity to review the design document and related smart contract source code of the `Plutos VirtualTrade`, 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 Plutos VirtualTrade

`Plutos Network` is a multi-chain synthetic issuance & derivative trading platform, which introduces mining incentives and staking rewards to users. The `Plutos VirtualTrade` protocol, as an important feature of `Plutos Network`, is a decentralized virtual trade game, which allows the players to profit from the rise or fall of the virtual assets (The prices of the virtual assets vary with the prices of the real assets). The `Plutos VirtualTrade` protocol enriches the `Plutos Network` ecosystem.

Table 1.1: Basic Information of Plutos VirtualTrade

Item	Description
Target	Plutos VirtualTrade
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 13, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit only covers the `core` sub-directory.

- https://gitlab.com/asresearch/plutos-virtual-trade.git (e7acfbe)

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

- https://gitlab.com/asresearch/plutos-virtual-trade.git (4da6824)

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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.

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

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

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.

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 `Plutos VirtualTrade` 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	1	■
Medium	1	■
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 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 high-severity vulnerability, 1 medium-severity vulnerability, and 1 low-severity vulnerability.

Table 2.1: Key Plutos VirtualTrade Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Logic Of VirtualTrade::sell()	Business Logic	Fixed
PVE-002	Low	Duplicate Asset Detection and Prevention	Business Logic	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	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 Improper Logic Of VirtualTrade::sell()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `VirtualTrade`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `Plutos VirtualTrade` protocol, the `Entry` contract allows the user to deposit the supported `stable_token` and get in return the virtual stablecoin (i.e., `chip` token). Meanwhile, the `VirtualTrade` contract allows the user to buy the virtual assets with `chip` token and sell the virtual assets to get `chip` token. The user can profit from the rise or fall of the virtual assets (The prices of the virtual assets vary with the prices of the real assets). In particular, the `VirtualTrade::sell()` routine allows the user to sell the specified virtual asset to get `chip` token. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `VirtualTrade` contract. At the beginning of the `sell()` routine, the statement (i.e., `uint256 chip_amount = asset.position[owner].safeMul(oracle.get_asset_price(name)).safeDiv(1e18)`) (line 67) is executed to calculate the amount of the `chip` token that the user can receive by selling the certain amount (specified by the input `amount` parameter) of the virtual asset (specified by the input `name` parameter). However, we notice the `asset.position[owner]` that saves the user's total amount of the specified virtual asset rather than the input `amount` is incorrectly used in the above calculation, which directly undermines the assumption of the design. Given this, we suggest to correct the implementation as below: `uint256 chip_amount = amount.safeMul(oracle.get_asset_price(name)).safeDiv(1e18)` (line 67).

```
62    function sell(string memory name, uint256 amount, uint256 min_rec, address owner)
          public returns(uint256){
63        asset_info storage asset = all_assets[name];
```

```
        require(asset.exist, "invalid asset");
        require(owner == msg.sender  allowed[owner][msg.sender], "permission denied");
        require(asset.position[owner] >= amount, "not enough position");
        uint256 chip_amount = asset.position[owner].safeMul(oracle.get_asset_price(name)
            ).safeDiv(1e18);
        require(chip_amount >= min_rec, "Sell sllipage");
        uint256 before = asset.position[owner];
        asset.position[owner] = before.safeSub(amount);
        asset.total_position = asset.total_position.safeSub(amount);
        asset.invest[owner] = asset.invest[owner].safeMul(asset.position[owner]).safeDiv
            (before);
        TokenInterface(chip).generateTokens(owner, chip_amount);
        emit AssetSell(name, owner, chip_amount, amount);
        return chip_amount;
    }
```

Listing 3.1: `VirtualTrade::sell()`

Recommendation Correct the implementation of the `sell()` routine as above-mentioned.

Status The issue has been addressed by the following commit: `4da6824`.

3.2 Duplicate Asset Detection and Prevention

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VirtualTrade`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.1, the `VirtualTrade` contract allows the user to buy the virtual assets with `chip` token and sell the virtual assets to get `chip` token. In current implementation, there are several kinds of concurrent virtual assets that can be traded and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account).

The addition of a new kind of virtual asset is implemented in `add_asset()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new kind of virtual asset with a duplicate asset from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong asset introduction from human omissions.

```
    function add_asset(string memory name, address type_addr, bytes memory call_data)
        public onlyOwner{
        all_names.push(name);
```

```
        all_assets[name].exist = true;
        oracle.add_asset(name, type_addr, call_data);
        emit NewAsset(name, type_addr, call_data);
    }
```

Listing 3.2: `VirtualTrade::add_asset()`

Recommendation Detect whether the given asset for addition is a duplicate of an existing asset. The asset addition is only successful when there is no duplicate.

Status The issue has been addressed by the following commit: `4da6824`.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VirtualTrade`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `Plutos VirtualTrade` protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure the supported assets and price oracle). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function remove_asset(string memory name) public onlyOwner{
        uint index = index_of(name);
        all_names[index] = all_names[all_names.length - 1];

        delete all_names[all_names.length-1];
        all_names.length--;
        all_assets[name].exist = false;
        emit RemoveAsset(name);
    }
    function add_asset(string memory name, address type_addr, bytes memory call_data)
        public onlyOwner{
        all_names.push(name);
        all_assets[name].exist = true;
        oracle.add_asset(name, type_addr, call_data);
        emit NewAsset(name, type_addr, call_data);
    }
```

Listing 3.3: `VirtualTrade::remove_asset()&&add_asset()`

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 privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the Plutos VirtualTrade 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 intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team. The team intends to introduce multi-sig mechanism to mitigate this issue when the protocol is deployed on the mainnet.

4 Conclusion

In this audit, we have analyzed the `Plutos VirtualTrade` design and implementation. `Plutos Network` is a multi-chain synthetic issuance & derivative trading platform, which introduces mining incentives and staking rewards to users. The `Plutos VirtualTrade` protocol is a decentralized virtual trade game, which allows the players to benefit from their virtual trade. The `Plutos VirtualTrade` protocol enriches the `Plutos Network` ecosystem. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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