SMART CONTRACT AUDIT REPORT

for

Plutos V1

Prepared By: Yiqun Chen

PeckShield

September 20, 2021

Document Properties

Client	Plutos Network
Title	Smart Contract Audit Report
Target	Plutos V1
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	September 20, 2021	Shulin Bie	Final Release

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 V1, 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 V1

Plutos Network is a multi-chain synthetic issuance & derivative trading platform, which introduces mining incentives and staking rewards to users. The Plutos V1 protocol is an important feature of Plutos Network, which provides decentralized PLUT/pUSD mortgage and lending service. The Plutos V1 protocol enriches the Plutos Network ecosystem and also presents a unique contribution to current DeFi ecosystem.

The basic information of Plutos V1 is as follows:

Table 1.1: Basic Information of Plutos V1

Item	Description
Target	Plutos V1
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 20, 2021

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

- https://gitlab.com/asresearch/plutos-eth-contract.git (0777815)

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-eth-contract.git (5a5601d)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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

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

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) [9], 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 V1` 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	
Undetermined	0	
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 Plutos V1 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Costly *lp*_token From Improper Staking Initialization	Time and State	Mitigated
PVE-002	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-003	Low	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-004	Low	Potential Reentrancy Risk In Plutos Implementation	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 Possible Costly *lp_token* From Improper Staking Initialization

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: ERC20Staking
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The ERC20Staking contract allows users to stake the supported target_token tokens and get in return lp_token tokens to represent the pool shares. While examining the share calculation with the given stakes, we notice an issue that may unnecessarily make the pool token extremely expensive and bring hurdles (or even causes loss) for later stakers.

To elaborate, we show below the related code snippet of the ERC20Staking contract. The stake() routine is used for participating users to stake the supported asset and get respective lp_token in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function stake(uint256 _amount) public returns(uint256){
        uint256 amount = 0;
        uint256 prev = IERC20(target_token).balanceOf(address(this));
        IERC20(target_token).safeTransferFrom(msg.sender, address(this), _amount);
        amount = IERC20(target_token).balanceOf(address(this)).safeSub(prev);

        if(amount == 0){
            return 0;
        }

        uint256 lp_amount = 0;
        {
            if(IERC20(lp_token).totalSupply() == 0){
```

```
52                lp_amount = amount.safeMul(uint256(10)**ERC20Base(lp_token).decimals()).
                      safeDiv(uint256(10)**ERC20Base(target_token).decimals());
53            }else{
54                uint256 t2 = IERC20(lp_token).totalSupply();
55                lp_amount = amount.safeMul(t2).safeDiv(prev);
56            }
57        }
58        if(lp_amount == 0){
59            return 0;
60        }

62        TokenInterface(lp_token).generateTokens(msg.sender, lp_amount);

64        if(address(callback) != address(0x0)){
65            callback.onStake(msg.sender, amount, lp_amount);
66        }

68        emit ERC20Stake(msg.sender, amount, lp_amount);
69        return lp_amount;
70    }
```

Listing 3.1: `ERC20Staking::stake()`

Specifically, when the pool is being initialized, the `lp_amount` share value directly takes the value of `amount` (line 52), which is under control by the malicious actor. As this is the first stake, the current total supply equals the calculated `lp_amount = amount.safeMul(uint256(10)**ERC20Base(lp_token).decimals()).safeDiv(uint256(10)**ERC20Base(target_token).decimals())= 1WEI`. With that, the actor can further transfer a huge amount of `target_token` tokens to `ERC20Staking` contract with the goal of making the `lp_token` extremely expensive.

An extremely expensive pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for staked assets. If truncated to be zero, the staked assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `stake()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first stake to avoid being manipulated.

Status The issue has been addressed by the following commit: `5a5601d`.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PLiquidateAgent
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

In the Plutos implementation, the PLiquidateAgent contract is designed to provide the interface used to liquidate assets for the PMintBurn contract. While examining the logics of it, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

To elaborate, we show below the related code snippet of the PLiquidateAgent contract. The public target_fee_pool storage variable is declared (line 17), but is not used in the contract. We suggest to remove it safely to keep the Plutos implementation clean.

```
11  contract PLiquidateAgent is Ownable{

13      using SafeMath for uint256;
14      address public target_token;
15      address public target_token_pool;
16      address public stable_token;
17      address public target_fee_pool;

19      ...

21  }
```

Listing 3.2: PLiquidateAgent

Recommendation Consider the removal of the redundant state.

Status The issue has been addressed by the following commit: 5a5601d.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Plutos implementation, there is a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function changeCallback(address addr) public onlyOwner returns(bool){
        address old = address(callback);
        callback = ERC20StakingCallbackInterface(addr);
        emit ERC20StakingChangeCallback(old, addr);
        return true;
    }
```

Listing 3.3: ERC20Staking::changeCallback()

```
    function resetTarget(bytes32 _key, address _target) public onlyOwner{
        address old = address(targets[_key]);
        targets[_key] = _target;
        emit TargetChanged(_key, old, _target);
    }
```

Listing 3.4: PDispatcher::resetTarget()

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

Recommendation Promptly transfer the privileged owner 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 privileged owner account will be managed by a multi-sig account.

3.4 Potential Reentrancy Risk In Plutos Implementation

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact:Medium
- Target: `Multiple Contracts`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

In the `Plutos` implementation, we notice there are several functions that have potential reentrancy risk. In the following, we take the `PMintBurn::deposit()` routine as an example. To elaborate, we show below the code snippet of the `deposit()` routine in the `PMintBurn` contract. In the function, the `IERC20(target_token).safeTransferFrom(msg.sender, pool, _amount)` is called (line 54) to transfer the `target_token` to the `pool`. If the `target_token` faithfully implements the ERC777-like standard, then the `PMintBurn::deposit()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

```
47     function deposit(uint256 _amount) public returns(bytes32){
48         bytes32 hash = hash_from_address(msg.sender);
49         IPMBParams param = IPMBParams(dispatcher.getTarget(param_key));
50 
51         require(_amount >= param.minimum_deposit_amount(), "need to be more than minimum
               amount");
52 
53         uint256 prev = IERC20(target_token).balanceOf(pool);
54         IERC20(target_token).safeTransferFrom(msg.sender, pool, _amount);
55         uint256 amount = IERC20(target_token).balanceOf(pool).safeSub(prev);
56 
57         deposits[hash].from = msg.sender;
58         deposits[hash].exist = true;
59         deposits[hash].target_token_amount = deposits[hash].target_token_amount.safeAdd(
               amount);
60         emit PDeposit(msg.sender, hash, amount, deposits[hash].target_token_amount);
61         return hash;
62     }
```

Listing 3.5: `PMintBurn::deposit()`

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `IERC20(target_token).safeTransferFrom(msg.sender, pool, _amount)` (line 54). By doing so, we can effectively keep `prev` intact (used for the calculation of actual `target_token` amount transferred to the `pool` at line 55). With a lower `prev`, the re-entered `PMintBurn::deposit()` is able to obtain more deposit credits. It can be repeated to exploit this vulnerability for gains, just like earlier Uniswap/imBTC hack [12].

Note the `ERC20Staking::stake()/claim()` routines share the same issue.

Moreover, we also suggest to add necessary reentrancy guards for other public functions, i.e., `PMintBurn::borrow()/repay()/withdraw()/liquidate()` as `UniswapV2` does.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been addressed by the following commit: `5a5601d`.

4 | Conclusion

In this audit, we have analyzed the `Plutos V1` 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 V1` protocol is an important feature of `Plutos Network`, which provides decentralized `PLUT/pUSD` mortgage and lending service. The `Plutos V1` protocol enriches the `Plutos Network` ecosystem and also presents a unique contribution to current DeFi ecosystem. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.