SMART CONTRACT AUDIT REPORT

for

Synnax Protocol

Prepared By: Xiaomi Huang

PeckShield

November 19, 2024

Document Properties

Client	Synnax
Title	Smart Contract Audit Report
Target	Synnax
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 19, 2024	Xuxian Jiang	Final Release
1.0-rc1	November 16, 2024	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 **4**

1.1 About Synnax . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 6

2 Findings **9**

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results **11**

3.1 Improved cook() Logic in CauldronV4 . . . 11

3.2 Revisited onlyCauldron() Modifier in DegenBox . . . 13

3.3 Suggested Adherence of Checks-Effects-Interactions . . . 14

3.4 Trust Issue of Admin Keys . . . 15

4 Conclusion **17**

References **18**

1 Introduction

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

Synnax is a decentralized finance protocol built on the SEI network, developed as fork of Abracadabra Money. It allows users to mint a stablecoin by collateralizing yield-bearing tokens, utilizing the proven mechanics of Collateralized Debt Positions (CDPs). Synnax enables users to unlock liquidity from their yield-generating assets without sacrificing interest earnings, providing a flexible and efficient means of capital utilization. By maintaining exposure to both stable assets and yield opportunities, Synnax offers a reliable and adaptable solution for decentralized finance participants on SEI. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Synnax

Item	Description
Target	Synnax
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 19, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers only the following contracts: `DegenBox.sol`, `CauldronV4.sol`, `ProxyOracle.sol`, `PythOracle.sol`, `iSei_PythOracle.sol`, `SyStrategy.sol`, `Vault.sol`, `syUSDSilo.sol`, and `syUSD.sol`.

- https://github.com/Synnax-Protocol/contract-protocol.git (e4b8cae)

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

- https://github.com/Synnax-Protocol/contract-protocol.git (a54c0b4)

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 implementation of the `Synnax` 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	2	🟦🟦
Informational	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 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 2 low-severity vulnerabilities.

Table 2.1: Key Synnax Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved cook() Logic in CauldronV4	Business Logic	Resolved
PVE-002	Low	Revisited onlyCauldron() Modifier in DegenBox	Business Logic	Resolved
PVE-003	Low	Suggested Adherence of Checks-Effects-Interaction in ERC46	Time and State	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved cook() Logic in CauldronV4

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CauldronV4`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Synnax` protocol has a core `CauldronV4` contract that enables the minting and management of stablecoins through collateralized debt positions. In the process of examining the accounting logic of managing user positions, we notice current implementation should be improved.

In particular, we show below the code snippet from the `cook()` routine. As the name indicates, this routine is used to execute a set of user actions and allows composability (contract calls) to other contracts. However, the specific actions of `ACTION_UPDATE_EXCHANGE_RATE` (line 497) and `ACTION_BENTO_SETAPPROVAL` (line 501) can be improved. Specifically, the `ACTION_UPDATE_EXCHANGE_RATE` is expected to change the update the collateralization value, which makes is necessary to validate the solvency at the end of executing these actions. Also, the `ACTION_BENTO_SETAPPROVAL` should validate the user input to ensure the parameter of `user` should be not `contract(this)` to disallow to approve others on the `CauldronV4` possession.

```
470     function cook(
471         uint8[] calldata actions,
472         uint256[] calldata values,
473         bytes[] calldata datas
474     ) external payable returns (uint256 value1, uint256 value2) {
475         CookStatus memory status;

477         for (uint256 i = 0; i < actions.length; i++) {
478             uint8 action = actions[i];
479             if (!status.hasAccrued && action < 10) {
480                 accrue();
```

```
                status.hasAccrued = true;
            }
            if (action == ACTION_ADD_COLLATERAL) {
                (int256 share, address to, bool skim) = abi.decode(datas[i], (int256, address, bool));
                addCollateral(to, skim, _num(share, value1, value2));
            } else if (action == ACTION_REPAY) {
                (int256 part, address to, bool skim) = abi.decode(datas[i], (int256, address, bool));
                _repay(to, skim, _num(part, value1, value2));
            } else if (action == ACTION_REMOVE_COLLATERAL) {
                (int256 share, address to) = abi.decode(datas[i], (int256, address));
                _removeCollateral(to, _num(share, value1, value2));
                status.needsSolvencyCheck = true;
            } else if (action == ACTION_BORROW) {
                (int256 amount, address to) = abi.decode(datas[i], (int256, address));
                (value1, value2) = _borrow(to, _num(amount, value1, value2));
                status.needsSolvencyCheck = true;
            } else if (action == ACTION_UPDATE_EXCHANGE_RATE) {
                (bool must_update, uint256 minRate, uint256 maxRate) = abi.decode(datas[i], (bool, uint256, uint256));
                (bool updated, uint256 rate) = updateExchangeRate();
                require((!must_update  updated) && rate > minRate && (maxRate == 0  rate < maxRate), "Cauldron: rate not ok");
            } else if (action == ACTION_BENTO_SETAPPROVAL) {
                (address user, address _masterContract, bool approved, uint8 v, bytes32 r, bytes32 s) =
                    abi.decode(datas[i], (address, address, bool, uint8, bytes32, bytes32));
                bentoBox.setMasterContractApproval(user, _masterContract, approved, v, r, s);
            }
            ...
    }
```

Listing 3.1: CauldronV4::cook()

Recommendation Revise the above routine to ensure two specific actions of `ACTION_UPDATE_EXCHANGE_RATE` and `ACTION_BENTO_SETAPPROVAL` are properly handled.

Status The issue has been fixed by this commit: `a54c0b4`.

3.2 Revisited onlyCauldron() Modifier in DegenBox

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DegenBox
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The Synnax protocol has another core DegenBox contract that stores funds, handles their transfers, supports flash loans and strategies. In the process of examining the specific modifier of onlyCauldron (), we notice current logic should be improved.

In the following, we show the implementation of this specific modifier. This modifier has a rather straightforward logic in validating the given user to be the approved CauldronV4. With that, there is a need to validate the given user's masterContractOf[from], not current masterContractOf[msg.sender] (line 151).

```
150    modifier onlyCauldron(address from) {
151        address masterContract = masterContractOf[msg.sender];
152        require(masterContract != address(0), "BentoBox: Only Cauldron");
153        require(whitelistedMasterContracts[masterContract], "BentoBox: Not whitelisted")
           ;
154        _;
155    }
```

Listing 3.2: DegenBox::onlyCauldron()

Recommendation Revise the above-mentioned modifier to properly validate the given user.

Status The issue has been fixed by this commit: a54c0b4.

3.3 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ERC4626
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

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` [11] exploit, and the `Uniswap/Lendf.Me` hack [10].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `ERC4626` as an example, the `withdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 86) starts before effecting the update on internal state (line 89), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same function. Note that there is no harm that may be caused to current protocol as it properly makes use of the `nonReentrant` modifier. However, it is still suggested to follow the known `checks-effects-interactions` best practice.

```
68     function withdraw(
69         uint256 assets,
70         address receiver,
71         address owner
72     ) public virtual returns (uint256 shares) {
73         shares = previewWithdraw(assets); // No need to check for rounding error,
               previewWithdraw rounds up.
74 
75         if (msg.sender != owner) {
76             uint256 allowed = allowance[owner][msg.sender]; // Saves gas for limited
                   approvals.
77 
78             if (allowed != type(uint256).max) {
79                 allowance[owner][msg.sender] = allowed - shares;
80             }
```

```
        }

        _beforeWithdraw(assets, shares);
        _burn(owner, shares);
        emit Withdraw(msg.sender, receiver, owner, assets, shares);
        _asset.safeTransfer(receiver, assets);

        unchecked {
            _totalAssets -= assets;
        }
    }
```

Listing 3.3: ERC4626::withdraw()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. Note another function `redeem()` within the same contract can also be similarly improved.

Status The issue has been fixed by this commit: `a54c0b4`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Synnax` protocol, there is a privileged account `owner` that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting, role assignment, and callee whitelisting). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function setTrustedCustodianWallet(address _trustedCustodianWallet) external
        onlyOwner {
        trustedCustodialWallets[_trustedCustodianWallet] = true;
        emit TrustedCustodianWalletSet(_trustedCustodianWallet);
    }

    /// @notice Unset the trusted custodian wallet
    /// @param _trustedCustodianWallet The address of the trusted custodian wallet
    function unsetTrustedCustodianWallet(address _trustedCustodianWallet) external
        onlyOwner {
        delete trustedCustodialWallets[_trustedCustodianWallet];
        emit TrustedCustodianWalletUnset(_trustedCustodianWallet);
    }

    /// @notice Run the strategy
    /// @param _trustedCustodianWallet The address of the trusted custodian wallet
    /// @param _amount The amount to transfer
    function run(address _trustedCustodianWallet, uint256 _amount) external payable
        onlyExecutor {
        if (!trustedCustodialWallets[_trustedCustodianWallet]) {
            revert InvalidTrustedCustodianWallet();
        }
        transferredAmounts[_trustedCustodianWallet] += _amount;
        strategyToken.safeTransfer(_trustedCustodianWallet, _amount);
    }

    /// @notice Update the transferred amount for a trusted custodian wallet
    /// @param _trustedCustodianWallet The address of the trusted custodian wallet
    /// @param _amount The amount to update
    function updateTransferredAmount(address _trustedCustodianWallet, uint256 _amount)
        external onlyOwner {
        transferredAmounts[_trustedCustodianWallet] = _amount;
    }
```

Listing 3.4: Example Privileged Functions in SyStrategy

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.

In the meantime, the protocol makes extensive use of proxy contracts to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

Recommendation Make the privileges explicit to the protocol users.

Status This issue has been mitigated as the team confirms the use of a multi-sig to manage the admin privilege.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Synnax` protocol, which is a decentralized finance protocol built on the `SEI` network. Developed as fork of `Abracadabra Money`, it allows users to mint a stablecoin by collateralizing yield-bearing tokens, utilizing the proven mechanics of `Collateralized Debt Positions (CDPs)`. `Synnax` enables users to unlock liquidity from their yield-generating assets without sacrificing interest earnings, providing a flexible and efficient means of capital utilization. By maintaining exposure to both stable assets and yield opportunities, `Synnax` offers a reliable and adaptable solution for decentralized finance participants on `SEI`. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

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

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