SMART CONTRACT AUDIT REPORT

for

UNCX Vesting

Prepared By: Xiaomi Huang

PeckShield

February 15, 2025

Document Properties

Client	UNCX
Title	Smart Contract Audit Report
Target	UNCX Vesting
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 15, 2025	Xuxian Jiang	Final Release
1.0-rc	January 30, 2025	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 vesting support in UNCX, 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 UNCX Vesting

UNCX Vesting provides the flexible vesting support in UNCX. The vesting schedules are supported with different vesting types (linear, exponential, interval) and the audited contracts provide necessary logic for computing and claiming vested token amounts based on these vesting schedules. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The UNCX Vesting

Item	Description
Name	UNCX
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 15, 2025

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

- https://github.com/uncx-private-repos/vesting-contract.git (e55c0d1)

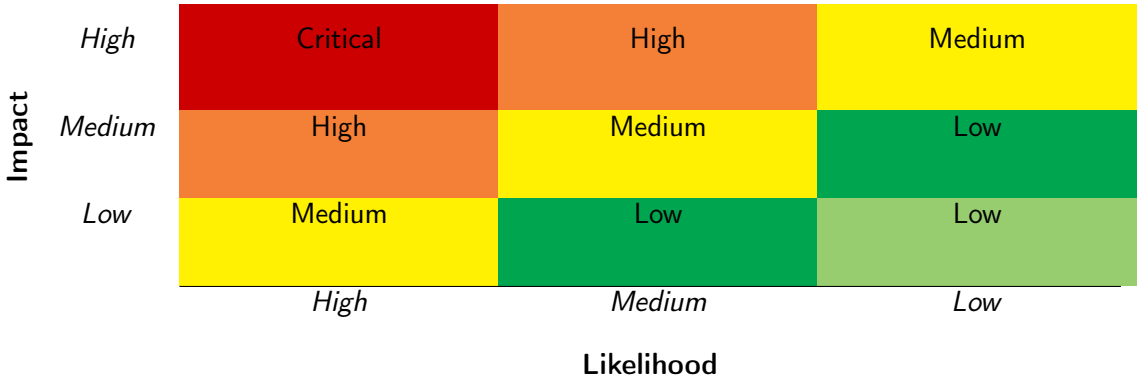
And here is the commit ID after fixes for the issues found in the audit have been checked in:

- https://github.com/uncx-private-repos/vesting-contract.git (8d1ff8d, 9887970)

1.2 About PeckShield

PeckShield Inc. [8] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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) [6], 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 design and implementation of the vesting support in UNCX. 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 logics, 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	0	
Low	2	■■
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 and 2 low-severity vulnerabilities.

Table 2.1: Key UNCX Vesting Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improved Validation of Function Arguments	Business Logic	Resolved
PVE-002	Low	Redundant State/Code Removal	Coding Practices	Resolved
PVE-003	Low	Revisited Share Computation in VestingManager	Numeric Issues	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved Validation of Function Arguments

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `VestingManager`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1109 [1]

Description

The vesting support in `UNCX` allows for flexible schedules, which may be cancelled or topped up. While reviewing the topup logic, we notice current implementation does not properly validate the user input and may be abused to drain funds from the vesting contract.

To elaborate, we show below the implementation of the related `topUpVesting()` routine. As the name indicates, this routine is used to top up a specific vesting schedule. The additional tranches for the topup is specified in the given input `additionalTranches`. However, it is not properly validated and a malicious one may be crafted to steal funds from the vesting contract. Specifically, an intermediate tranche may have a higher amount (than the final amount), which can then be exploited to redeem extra funds from the contract (by only depositing a smaller final amount). Also, it is better to validate the caller and have a thorough check on `additionalTranches`.

```
388     function topUpVesting(
389         uint256 vestingId,
390         IVestingCalculator.TimeAmount[] calldata additionalTranches
391     ) external nonReentrant {
392         VestingSchedule storage schedule = vestingSchedules[vestingId];
394         // Verify tranches start after current end
395         uint256 lastTime = schedule.tranches[schedule.tranches.length - 1].time;
396         if (additionalTranches[0].time < lastTime) revert InvalidTime();
398         // Get the final amount from the last tranche
```

```
399         uint256 newFinalAmount = additionalTranches[additionalTranches.length - 1].
                amount;

401         // The new final amount must be greater than current total amount
402         if (newFinalAmount <= schedule.totalAmount) revert InvalidAmount();

404         // Calculate top-up amount
405         uint256 topUpAmount = newFinalAmount - schedule.totalAmount;

407         // Update ratio before transfer
408         _updateShareRatio(schedule.token);

410         // Transfer tokens first to get actual amount
411         uint256 balanceBefore = IERC20(schedule.token).balanceOf(address(this));
412         IERC20(schedule.token).safeTransferFrom(msg.sender, address(this), topUpAmount);
413         uint256 actualTopUpAmount = IERC20(schedule.token).balanceOf(address(this)) -
                balanceBefore;

415         // Calculate new shares using current ratio
416         uint256 newShares = _toShares(schedule.token, actualTopUpAmount);

418         // Update state
419         schedule.totalAmount = newFinalAmount;
420         _depositedShares[vestingId] += newShares;
421         _totalTokenShares[schedule.token] += newShares;

423         // Update ratio after adding new shares
424         _updateShareRatio(schedule.token);

426         // Append new tranches
427         for (uint256 i = 0; i < additionalTranches.length; i++) {
428             schedule.tranches.push(additionalTranches[i]);
429         }

431         emit VestingToppedUp(vestingId, msg.sender, actualTopUpAmount,
                additionalTranches);
432     }
```

Listing 3.1: `VestingManager::topUpVesting()`

Recommendation Revisit the above routine to thoroughly validate the user input, especially the `additionalTranches` argument.

Status This issue has been fixed by the following commit: `88a94aa`.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Low
- Likelihood: N/A
- Impact: N/A
- Target: `VestingCalculator`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The vesting support in `UNCX` makes good use of a number of reference contracts, such as `ERC20`, `ERC721Burnable`, `SafeERC20`, and `ReentrancyGuard`, to facilitate its code implementation and organization. For example, the `VestingCalculator` smart contract has so far imported at least four reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `VestingCalculator` contract, there are a number of local variables or functions that are defined, but not used. Examples include the `owner` state defined in the parent contract `Ownable`, the `_calculateInterval()` function, as well as the `VestingTypeConfig` structure. In fact, the parent contract `Ownable` can be removed.

```
16 contract VestingCalculator is Ownable, IVestingCalculator {
17     using Math for uint256;
18
19
20     /**
21      * @dev Constructor to initialize the vesting calculator with default configurations
22      * @param initialOwner Address of the initial contract owner
23      */
24     constructor(address initialOwner) Ownable(initialOwner) {
25
26     }
27     ..
28 }
```

Listing 3.2: The `VestingCalculator` Contract

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed by the following commits: `896d775`, `f6729d9`, `c5cb835`, and `8d1ff8d`.

3.3 Revisited Share Computation in VestingManager

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VestingManager`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The vesting support in UNCX involves the conversion between vesting amount and respective share. Similar to the ERC4626 design, the amount/share conversion needs to be calculated in favor of the vesting protocol. While examining current logic for the amount/share conversion, we notice it is not always computed in favor of the vesting protocol. Moreover, the well-known inflation issue in ERC4626 is also applicable here.

To elaborate, we show below the implementation of the related routines, i.e., `_toShares()` and `_toAmount()`. The first routine is used to compute the share for the given token amount and the second routine is used to calculate the amount for the given share. Currently, the first routine computes the ceiling for the resulting share, which may need to be a flooring operation. The second routine also needs to be revised to take the flooring operation.

```
434     function _toShares(address token, uint256 amount) internal view returns (uint256) {
435         if (_totalTokenShares[token] == 0) {
436             // First deposit for this token, ratio starts at 1:1
437             return amount;
438         }
439
440         // Convert amount to shares using current ratio
441         // Multiply first, then divide to maintain precision
442         return (amount * _shareRatio[token] + 1e18 - 1) / 1e18;
443     }
444
445     function _toAmount(address token, uint256 shares) internal view returns (uint256) {
446         if (_totalTokenShares[token] == 0) return shares;
447
448         // Convert shares to amount using current ratio
449         // Add half divisor for proper rounding
450         return (shares * 1e18 + _shareRatio[token] / 2) / _shareRatio[token];
451     }
```

Listing 3.3: `VestingManager::_toShares()/_toAmount()`

Recommendation Revisit the above routine to properly convert between token amount and respective share.

Status This issue has been resolved by following the above suggestion.

4 Conclusion

In this audit, we have analyzed the design and implementation of the vesting support in UNCX. The vesting can flexibly supports various vesting schedules with different vesting types (linear, exponential, interval) and the audited contracts provide necessary logic for computing and claiming vested token amounts based on these vesting schedules. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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