



# SMART CONTRACT AUDIT REPORT

for

Vault Tec



Prepared By: Patrick Lou

PeckShield  
April 2, 2022

## Document Properties

Client	Vault.Inc
Title	Smart Contract Audit Report
Target	Vault Tec
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Confidential

## Version Info

Version	Date	Author(s)	Description
1.0-rc1	April 2, 2022	Xuxian Jiang	Release Candidate #1

## Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Vault.inc . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Revisited Logic in TokenSaver::saveToken() . . . . .	11
3.2	Improved Validation on Function Arguments . . . . .	12
3.3	Reentrancy Consistency Between TimeLockPool vs. MultiRewardsTimeLockPool . . . . .	13
3.4	Trust Issue of Admin Keys . . . . .	15
<b>4</b>	<b>Conclusion</b>	<b>17</b>
	<b>References</b>	<b>18</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Vault Tec` 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 Vault.inc

`Vault.inc` plays a symbiotic role with projects that the platform integrates. In particular, it has multiple liquidity provision strategies ranging from `Token` and `NFT` staking to protocol-owned-liquidity bonding. It also provides on-chain analysis and leaderboard for projects that are doing `IDO` on `Copper Launch`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `vault.inc` Protocol

Item	Description
Issuer	Vault.Inc
Website	<a href="https://vault.inc">https://vault.inc</a>
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 2, 2022

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

- <https://github.com/vault-tec-team/vault-tec-core.git> (b24e22c)

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

- <https://github.com/vault-tec-team/vault-tec-core.git> (eeadb9f)

## 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 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

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) [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 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit



Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 `Vault Tec` 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 logics, 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 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 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Vault Tec Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic in Token-Saver::saveToken()	Business Logic	Mitigated
PVE-002	Low	Improved Validation on Function Arguments	Coding Practices	Fixed
PVE-003	Low	Reentrancy Consistency Between Time-LockPool vs. MultiRewardsTimeLock-Pool	Time And State	Fixed
PVE-004	Medium	Trust on Admin Keys	Security Features	Mitigated

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 Revisited Logic in TokenSaver::saveToken()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: TokenSaver
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

#### Description

The Vault Tec protocol has a common `TokenSaver` contract which, as the name indicates, is designed to save tokens that may be accidentally sent to the contract. While reviewing the rescue logic, we notice the current implementation needs to be improved.

To elaborate, we show below the `saveToken()` routine, which has a simple logic in retrieving the given tokens from the contract. Note that this `TokenSaver` contract is inherited by a number of other contracts that may be receiving user stakes in the form of various tokens. It comes to our attention that this `saveToken()` routine does not the built-in mechanism to exclude the staking token! While this function is a privileged one and can only be called by trusted entities, it is helpful to ensure the trusted entities are not able to withdraw users' funds at the smart contract level.

```
24     function saveToken(address _token, address _receiver, uint256 _amount) external  
        onlyTokenSaver {  
25         IERC20(_token).safeTransfer(_receiver, _amount);  
26         emit TokenSaved(_msgSender(), _receiver, _token, _amount);  
27     }
```

Listing 3.1: `TokenSaver::saveToken()`

**Recommendation** Revise current `saveToken()` logic so that user stake funds may be excluded.

**Status** The issue has been confirmed. The team clarifies that the `TokenSaver` role will be assigned to a multisig contract. by properly following a guarded launch process.

## 3.2 Improved Validation on Function Arguments

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Vault Tec` protocol is no exception. Specifically, if we examine the `MultiRewardsBasePool` contract, it has defined a number of protocol-wide risk parameters, such as `escrowPortions` and `escrowDurations`. In the following, we show the corresponding routines that allow for their changes.

These protocol parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `escrowPortions` may greatly affect the escrow reward amount calculation in the `claimRewards()` operation, hence incurring cost to withdraw rewards or hurting the adoption of the protocol.

```

112     function addRewardToken(address _reward, address _escrowPool, uint256 _escrowPortion
      , uint256 _escrowDuration) external onlyAdmin {
113         if (!rewardTokensList[_reward]) {
114             rewardTokensList[_reward] = true;
115             rewardTokens.push(_reward);
116             escrowPools[_reward] = _escrowPool;
117             escrowPortions[_reward] = _escrowPortion;
118             escrowDurations[_reward] = _escrowDuration;

120             if(_reward != address(0) && _escrowPool != address(0)) {
121                 IERC20(_reward).safeApprove(_escrowPool, type(uint256).max);
122             }
123         }
124     }

```

Listing 3.2: `MultiRewardsBasePool::addRewardToken()`

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

**Status** The issue has been fixed by this commit: `eeadb9f`.

### 3.3 Reentrancy Consistency Between TimeLockPool vs. MultiRewardsTimeLockPool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

#### 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 recent Uniswap/Lendf.Me hack [10].

From another perspective, the `nonReentrant` modifier is also widely used to prevent re-entrancy. While examining current mitigation in the implementation, it comes to our attention that there is inconsistency in adopting the reentrancy defense mechanisms. For example, if we examine the two contracts `TimeLockPool` and `MultiRewardsTimeLockPool`, both have the `withdraw()` function. However, the former one does not have the `nonReentrant` modifier while the latter does have! As mentioned earlier, it is suggested to apply the `nonReentrant` modifier in both functions.

```

67     function withdraw(uint256 _depositId, address _receiver) external {
68         require(_receiver != address(0), "TimeLockPool.withdraw: receiver cannot be zero
           address");
69         require(_depositId < depositsOf[_msgSender()].length, "TimeLockPool.withdraw:
           Deposit does not exist");
70         Deposit memory userDeposit = depositsOf[_msgSender()][_depositId];
71         require(block.timestamp >= userDeposit.end, "TimeLockPool.withdraw: too soon");
72
73         //                               No risk of wrapping around on casting to uint256 since
           deposit end always > deposit start and types are 64 bits
74         uint256 shareAmount = userDeposit.amount * getMultiplier(uint256(userDeposit.end
           - userDeposit.start)) / 1e18;
75
76         // remove Deposit
77         depositsOf[_msgSender()][_depositId] = depositsOf[_msgSender()][depositsOf[
           _msgSender()].length - 1];
78         depositsOf[_msgSender()].pop();
79
80         // burn pool shares

```

```

81     _burn(_msgSender(), shareAmount);
82
83     // return tokens
84     depositToken.safeTransfer(_receiver, userDeposit.amount);
85     emit Withdrawn(_depositId, _receiver, _msgSender(), userDeposit.amount);
86 }

```

Listing 3.3: TimeLockPool::withdraw()

```

67     function withdraw(uint256 _depositId, address _receiver) external nonReentrant {
68         require(_receiver != address(0), "TimeLockPool.withdraw: receiver cannot be zero
           address");
69         require(_depositId < depositsOf[_msgSender()].length, "TimeLockPool.withdraw:
           Deposit does not exist");
70         Deposit memory userDeposit = depositsOf[_msgSender()][_depositId];
71         require(block.timestamp >= userDeposit.end, "TimeLockPool.withdraw: too soon");
72
73         // No risk of wrapping around on casting to uint256 since
           deposit end always > deposit start and types are 64 bits
74         uint256 shareAmount = userDeposit.amount * getMultiplier(uint256(userDeposit.end
           - userDeposit.start)) / 1e18;
75
76         // remove Deposit
77         depositsOf[_msgSender()][_depositId] = depositsOf[_msgSender()][depositsOf[
           _msgSender()].length - 1];
78         depositsOf[_msgSender()].pop();
79
80         // burn pool shares
81         _burn(_msgSender(), shareAmount);
82
83         // return tokens
84         depositToken.safeTransfer(_receiver, userDeposit.amount);
85         emit Withdrawn(_depositId, _receiver, _msgSender(), userDeposit.amount);
86     }

```

Listing 3.4: MultiRewardsTimeLockPool::withdraw()

**Recommendation** Be consistent in adopting the above reentrancy defense mechanism.

**Status** The issue has been fixed by this commit: [eeadb9f](#).

### 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 [2]

#### Description

In the Vault Tec protocol, there is a special admin account (with the `DEFAULT_ADMIN_ROLE`). This admin account plays a critical role in governing and regulating the system-wide operations (e.g., assign other roles, configure various settings, and recover funds). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

93     function adjustWeight(uint256 _poolId, uint256 _newWeight) external onlyGov {
94         require(_poolId < pools.length, "LiquidityMiningManager.adjustWeight: Pool does
           not exist");
95         distributeRewards();
96         Pool storage pool = pools[_poolId];

97
98         totalWeight -= pool.weight;
99         totalWeight += _newWeight;

101
102         pool.weight = _newWeight;

103         emit WeightAdjusted(_poolId, address(pool.poolContract), _newWeight);
104     }

106     function setRewardPerSecond(uint256 _rewardPerSecond) external onlyGov {
107         distributeRewards();
108         rewardPerSecond = _rewardPerSecond;

109
110         emit RewardsPerSecondSet(_rewardPerSecond);
111     }

```

Listing 3.5: Example Privileged Operations in `LiquidityMiningManager`

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.

**Recommendation** Promptly transfer the privileged account to the intended DAO-like governance contract. All changes 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** The issue has been confirmed by the team. The team clarifies that the admin key will be mitigated with a multisig with timelock contract.





## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Vault Tec` protocol. Note `Vault .inc` plays a symbiotic role with projects that the platform integrates. In particular, it has multiple liquidity provision strategies ranging from `Token` and `NFT` staking to protocol-owned-liquidity bonding. It also provides on-chain analysis and leaderboard for projects that are doing `IDO` on `Copper Launch`. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.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: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.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](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>.

