SMART CONTRACT AUDIT REPORT

for

Spherium Protocol

Prepared By: Shuxiao Wang

PeckShield
May 25, 2021

Document Properties

Client	Spherium Finance
Title	Smart Contract Audit Report
Target	Spherium Vesting
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 25, 2021	Yiqun Chen	Final Release
0.1	May 20, 2021	Yiqun Chen	First Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Spherium protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, 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 Spherium

Spherium is a financial ecosystem that unifies the current scattered DeFi (Decentralized Finance) landscape. Spherium utilises the principles of decentralized finance to provide a single platform for multi-asset, cross-chain swaps, crypto financing solutions, and cross-chain operability. The Spherium team aims to provide a transparent, decentralized, non-custodial, and user-friendly one-stop platform for all segments of the Financial system, empowering average users to avail the best products and services in the DeFi space to maximize their investment/loans returns with minimal efforts.

The basic information of the Spherium protocol is as follows:

Table 1.1: Basic Information of The Spherium Protocol

Item	Description
Issuer	Spherium Finance
Website	https://spherium.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 25, 2021

In the following, we show the address of the contract used in this audit:

- https://ropsten.etherscan.io/address/0xc476195aafc17fea2b94b67ee53450390a21026a#code

And this is the final revised file `SphrVestingStatic.sol` and its MD5/SHA checksum value after all fixes have been checked in :

- `MD5: 74d41d1627c35c2ed64ecab030ad7a10`
- `SHA256: e9fd022dbd8e9bdd0544199d1d7016f1686b9c42b45429ad8f8cd3115993370e`

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Spherium Vesting 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 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	1	■
Low	1	■
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, and 1 low-severity vulnerability.

Table 2.1: Key Spherium Vesting Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Two-Step Transfer Of Privileged Account Ownership	Coding Practices	Confirmed
PVE-002	Low	Suggested Use Of Safemath For claim()	Coding Practices	Fixed
PVE-003	High	Partial Funds Can Never Be Withdrawn	Coding Practices	Fixed

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 Two-Step Transfer Of Privileged Account Ownership

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Ownable
- Category: Coding Practices [3]
- CWE subcategory: CWE-1109 [4]

Description

The Spherium protocol implements a rather basic access control mechanism that allows a privileged account, i.e., owner, to be granted exclusive access to a typically sensitive function (i.g., addVestingSchedule()). Because of the privileged access and the implications of this sensitive function, the owner account is essential for the protocol-level safety and operation. In the following, we elaborate with the owner account.

```
    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
```

Listing 3.1: Ownable::transferOwnership()

The current implementation provides a specific function, i.e., transferOwnership(), to allow for possible owner updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the newOwner parameter is always correctly provided. However, in the unlikely situation, when an incorrect newOwner is provided, the contract owner may be forever lost, which might be devastating for Spherium operation and maintenance.

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract owner to an uncontrolled address. In other words, this two-step procedure ensures that a owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

Recommendation Implement a two-step approach for owner update (or transfer): `transferOwnership()` and `acceptOwnership()`.

Status This issue has been confirmed.

3.2 Suggested Use Of Safemath For claim()

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `SphrVestingStatic`
- Category: Coding Practices [3]
- CWE subcategory: CWE-190 [2]

Description

The `Spherium` protocol allows the `owner` to set a schedule for the user, and the schedule includes the available time and the amount of tokens for the user to withdraw from the `SphrVestingStatic` contract.

In the following, we list below the `claim()` and the `vestedAmount()` functions. Our analysis exposes an underflow issue in both these two functions.

```
    function claim() public returns (bool) {
        VestingSchedule[] memory vestingSchedules_ = _vestingSchedules[msg.sender];

        uint256 vestedAmount_;
        for(uint256 i=0; i<vestingSchedules_.length; i++) {
            if (vestingSchedules_[i].schedule < block.timestamp) {
                vestedAmount_ += vestingSchedules_[i].amount;
                delete vestingSchedules_[i];
            }
        }
        vestedAmount_ -= _releaseAmount[msg.sender];
        require(vestedAmount_ > 0, "SphrVestingStatic: vested amount must be greater
            then 0");
        _token.safeTransfer(msg.sender, vestedAmount_);
        _releaseAmount[msg.sender] += vestedAmount_;
```

```
        return true;
    }
```

Listing 3.2: SphrVestingStatic :: claim()

```
    function vestedAmount() public view virtual returns (uint256) {
        VestingSchedule[] memory vestingSchedules_ = _vestingSchedules[msg.sender];

        uint256 vestedAmount_;
        for(uint256 i=0; i<vestingSchedules_.length; i++) {
            if (vestingSchedules_[i].schedule < block.timestamp) {
                vestedAmount_ += vestingSchedules_[i].amount;
            }

        }
        vestedAmount_ -= _releaseAmount[msg.sender];
        return vestedAmount_;
    }
```

Listing 3.3: SphrVestingStatic :: vestedAmount()

The problem is when the `vestedAmount_` is smaller than the `_releaseAmount[msg.sender]`, the underflow occurs, and the `vestedAmount_` will be extremely large. For the `claim()` function, although the `safeTransfer()` will revert if the contract does not have enough tokens, we still have to consider the worst condition. If the contract has enough tokens, the user can drain the founds from it. For the `vestedAmount()` function, it will return an extremely large `vestedAmount` when the user tries to check his/her own `vestedAmount`.

Recommendation Use `Safemath` for all the calculations in the `claim()` function and the `vestedAmount()` function.

Status This issue has been fixed.

3.3 Partial Funds Can Never Be Withdrawn

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `SphrVestingStatic`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

The `Spherium` protocol allows the user to withdraw a certain amount of tokens after the scheduled time. The user could have many schedules with different time and amount set by the `owner`.

```
706     function claim() public returns (bool) {
707         VestingSchedule[] memory vestingSchedules_ = _vestingSchedules[msg.sender];
708 
709         uint256 vestedAmount_;
710         for(uint256 i=0; i<vestingSchedules_.length; i++) {
711             if (vestingSchedules_[i].schedule < block.timestamp) {
712                 vestedAmount_ += vestingSchedules_[i].amount;
713                 delete vestingSchedules_[i];
714             }
715         }
716         vestedAmount_ -= _releaseAmount[msg.sender];
717         require(vestedAmount_ > 0, "SphrVestingStatic: vested amount must be greater
                then 0");
718         _token.safeTransfer(msg.sender, vestedAmount_);
719         _releaseAmount[msg.sender] += vestedAmount_;
720         return true;
721     }
```

Listing 3.4: SphrVestingStatic::claim()

To elaborate, we show above the related `claim()` function. An problem may occur when the user tries to call `claim()` multiple times to collect tokens separately. In the following, we illustrate this problem by a specific example. Assume the user has two schedules, one is 100 tokens (`amount`) and 5 days (`time`), and another one is 50 tokens and 10 days. After 5 days, the user calls `claim()`, and receives 100 tokens, so the `_releaseAmount[msg.sender]` increases to 100, then after another 5 days, when the user calls `claim()` again, the `vestedAmount` now equals to 50 this time, and the result of subtraction between the `vestedAmount` and the `_releaseAmount[msg.sender]` will be extremely large because of the underflow. If the contract does not have enough tokens, it will finally revert. Even we use the `Safemath` as suggested in Section 3.2, it will still revert. As a result, the user can never withdraw this part of funds.

Recommendation Remove the statement of `vestedAmount_ -= _releaseAmount[msg.sender]` (line 716) in the `claim()` function.

Status This issue has been fixed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Spherium` protocol. The audited contract allows the user to collect a certain amount of tokens after the available time set by the contract `owner`. 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-1041: Partial Funds Can Never Be Withdrawn. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE CATEGORY: Two-Step Transfer Of Privileged Account Ownership. https://cwe.mitre.org/data/definitions/1109.html.

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

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

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