SMART CONTRACT AUDIT REPORT

for

SiloLeverageStrategy (Factor)

Prepared By: Xiaomi Huang

PeckShield

March 17, 2024

Document Properties

Client	Factor Studio
Title	Smart Contract Audit Report
Target	SiloLeverageStrategy
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	March 17, 2024	Xuxian Jiang	Final Release
1.0-rc	March 10, 2024	Xuxian Jiang	Release Candidate

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 `SiloLeverageStrategy` in `Factor`, 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 Factor

`Factor` provides a middleware infrastructure to aggregate core `DeFi` primitives, including the creation, management, and discovery of powerful financial instruments with innovative vaults, yield pools, lending pools, liquidity pools, and tokenized baskets. The audited `SiloLeverageStrategy` contract adds the much-desired support of `Silo`, which aims to maximize yield, switch assets, and manage debts smartly. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SiloLeverageStrategy

Item	Description
Name	Factor Studio
Website	https://factor.fi/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 17, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the repository has a number of contracts and this audit covers the following contract: `SiloLeverageStrategy.sol`.

- https://github.com/FactorDAO/factor-monorepo.git (852f5c5)

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

- https://github.com/FactorDAO/factor-monorepo.git (9ebd028)

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 the 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 Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	DeltaPrimeLabs 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 checklist of 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 `SiloLeverageStrategy` contract in `Factor`. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	1	■
Informational	0	
Total	2	

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 1 medium-severity vulnerability and 1 low-severity vulnerability.

Table 2.1: Key SiloLeverageStrategy Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited feeAmount Logic in _flSwitchDebt()	Business Logic	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Revisited feeAmount Logic in _flSwitchDebt()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SiloLeverageStrategy
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The examined `SiloLeverageStrategy` contract is designed to create and manage user positions on `Silo`. While examining the logic to switch a position's debt, we notice the implementation can be improved.

To elaborate, we show below the related `_flSwitchDebt()` routine. This routine implements the actual logic to repay all current debt, next borrow in new debt, and then swap new debt to repay flashloan. However, it comes to our attention that the `feeAmount` is supposed in `oldDebtToken` (line 368), but used to borrow from `newDebtToken` (line 374).

```
354     function _flSwitchDebt(bytes calldata params, uint256 feeAmount) internal {
355         // decode params
356         (address newDebtToken, uint256 repayAmount, uint256 newDebt, address poolAddress
            , bytes memory data) = abi
357             .decode(params, (address, uint256, uint256, address, bytes));
358         address oldDebtToken = debtToken();
359
360         // repay all debt
361         IERC20(debtToken()).approve(poolAddress, repayAmount);
362         ISiloStrategy(poolAddress).repay(debtToken(), repayAmount);
363
364         _debtToken = IERC20(newDebtToken);
365         _debtPool = IERC20(IFactorLeverageVault(vaultManager()).debts(newDebtToken));
366
367         // borrow
368         ISiloStrategy(poolAddress).borrow(debtToken(), newDebt + feeAmount);
```

```
        // swap new debt to flashloan
        this.swapBySelf(debtToken(), oldDebtToken, IERC20(_debtToken).balanceOf(address(
            this)), data);

        // repay Flashloan
        IERC20(oldDebtToken).safeTransfer(balancerVault, repayAmount + feeAmount);

        emit DebtSwitched(newDebtToken, assetBalance());
    }
```

Listing 3.1: `SiloLeverageStrategy::_flSwitchDebt()`

Recommendation Revise the above routine to calculate the correct fee amount to repay flashloan.

Status The issue has been fixed by removing the related debt-switching logic.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
```

```
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.2: USDT Token Contract

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()/transferFrom()` as well, i.e., `safeTransfer()/safeTransferFrom()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.3: `SafeERC20::safeApprove()`

In current implementation, if we examine the `SiloReward::borrow()/repay()` routines that are designed to borrow and repay current debt. To accommodate the specific idiosyncrasy, there is a need to make use of `safeTransfer()` and `safeApprove()`.

```
    function borrow(address debtToken, uint256 amount, address poolAddress) public
        returns (uint256) {
```

```
        ISiloStrategy(poolAddress).borrow(debtToken, amount);
        IERC20(debtToken).transfer(msg.sender, amount);
        return amount;
    }

    function repay(address debtToken, uint256 amount, address poolAddress) public
        returns (uint256) {
        IERC20(debtToken).transferFrom(msg.sender, address(this), amount);

        IERC20(debtToken).approve(poolAddress, amount);
        ISiloStrategy(poolAddress).repay(debtToken, amount);

        return amount;
    }
```

Listing 3.4: `SiloReward::borrow()/repay()`

Note other routines, such as `SiloLeverageStrategy::addLeverage()/_flAddLeverage()/_flRemoveLeverage()/_flSwitchAsset()/_flSwitchDebt()/_flCloseLeverage()/claimRewardsSupply()/claimRewardsRepay()/supply()/repay()/withdraw()` share the same issue.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note this issue affects all current leveraged strategies.

Status The issue has been fixed by this commit: `9ebd028`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `SiloLeverageStrategy` in `Factor`, which provides a middleware infrastructure to aggregate core `DeFi` primitives, including the creation, management, and discovery of powerful financial instruments. The audited `SiloLeverageStrategy` contract adds the much-desired support of `Silo`, which aims to maximize yield, switch assets, and manage debts smartly. The current code base is well structured and neatly organized. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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