SMART CONTRACT AUDIT REPORT

for

Sake Finance (Soneium)

Prepared By: Xiaomi Huang

PeckShield

February 17, 2025

Document Properties

Client	Sake Finance
Title	Smart Contract Audit Report
Target	Sake Finance
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	February 17, 2025	Xuxian Jiang	Final Release
1.0-rc1	February 15, 2025	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Sake Finance (Soneium) 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 Sake Finance (Soneium)

Sake Finance is an innovative platform that introduces the concept of on-chain AI agents to the world of decentralized finance and beyond. At its core, Sake Finance is designed to empower users with autonomous, intelligent agents capable of executing a broad spectrum of tasks directly on the blockchain. These tasks range from complex financial transactions to dynamic roles within interactive gaming environments, all performed with a level of sophistication and adaptability previously unseen in traditional crypto bots. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Sake Finance Protocol

Item	Description
Name	Sake Finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 17, 2025

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

- https://github.com/AgentFi/agentfi-contracts-soneium.git (509bfcf)

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

- https://github.com/AgentFi/agentfi-contracts-soneium.git (TBD)

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

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

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

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 1.4: 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 `Sake Finance (Soneium)` 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	1	■
Low	1	■
Informational	1	■
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 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational suggestion.

Table 2.1: Key Sake Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack of Caller Validation in UniswapV3 Swap Callback	Security Features	Resolved
PVE-002	Informational	Improved Logic Consistency Between _convertFreeAssetsToSupplyToken() And _convertFreeAssetsToSupplyAToken()	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of 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 Lack of Caller Validation in UniswapV3 Swap Callback

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DexAggregator`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

To facilitate the interaction with external `DEX` engines, `Sake Finance` has a built-in `DexAggregator` to handle token swaps across different protocols. While examining the interaction with the external `UniswapV3 DEX` engine, we notice a possible issue that needs to be addressed.

To elaborate, we show below the implementation of the related routine, i.e., `uniswapV3SwapCallback()`. As the name indicates, this routine is a callback that will be invoked to execute the desired swaps and pay the required tokens. However, each callback must be validated to verify that the call is originated from a genuine `UniswapV3` pool. Otherwise, the pool contract would be vulnerable to attack via an `EOA` manipulating the callback function.

```
    function uniswapV3SwapCallback(
        int256 amount0Delta,
        int256 amount1Delta,
        bytes calldata _data
    ) external override(IUniswapV3SwapCallback, IDexAggregator) {
        if(!(amount0Delta > 0  amount1Delta > 0)) revert Errors.AmountZero();

        (, address tokenIn, ) = abi.decode(_data, (address, address, address));

        uint256 amountToPay = amount0Delta > 0 ? uint256(amount0Delta) : uint256(
            amount1Delta);

        SafeERC20.safeTransfer(IERC20(tokenIn), msg.sender, amountToPay);
    }
```

Listing 3.1: `DexAggregator::uniswapV3SwapCallback()`

Recommendation Revise the above routine to properly validate the caller to ensure it is a genuine UniswapV3 pool.

Status This issue has been resolved in the following commit: 32.

3.2 Improved Logic Consistency Between _convertFreeAssetsToSupplyToken() And _convertFreeAssetsToSupplyAToken()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LooperModuleA/B
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The Sake Finance protocol supports a number of modules that can be readily extended and integrated. While examining a specific module for the Looper strategy, we notice the use of two internal routines for token swaps can be improved for consistency.

To elaborate, we show below the implementations of these two helper routines in LooperModuleA. These two routines are designed to convert unused assets to SupplyToken and SupplyAToken, respectively. It comes to our attention that the first one does not have the step of converting native currency Ether to WETH, while the second one does (line 844). Note this inconsistency is also present in another contract, i.e., LooperModuleB.

```
836     function _convertFreeAssetsToSupplyToken() internal {
837         _convertBorrowToken(true);
838         _convertSupplyATokenToSupplyToken();
839     }
840
841     /// @notice Converts all free assets to the supplyAToken.
842     /// @dev Assumes no borrow debt.
843     function _convertFreeAssetsToSupplyAToken() internal {
844         _convertEthToWeth();
845         _convertBorrowToken(false);
846         _convertSupplyTokenToSupplyAToken();
847     }
```

Listing 3.2: LooperModuleA::_convertFreeAssetsToSupplyToken()/_convertFreeAssetsToSupplyAToken()

Recommendation Revise the above-mentioned routines for improved consistency.

Status This issue has been resolved in the following commit: 32.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Sake Finance` contract, there is a privileged account (`owner`) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, manage factory/roles, and lock contract accounts). In the following, we show the representative functions potentially affected by the privilege of the privileged account.

```
    function setRoles(SetRolesParam[] calldata params) external payable override {
        _strategyManagerPrecheck();
        AccessControlLibrary.AccessControlLibraryStorage storage acls =
            AccessControlLibrary.accessControlLibraryStorage();
        for(uint256 i = 0; i < params.length; ++i) {
            bytes32 role = params[i].role;
            address account = params[i].account;
            bool grantAccess = params[i].grantAccess;
            acls.assignedRoles[role][account] = grantAccess;
            emit RoleAccessChanged(role, account, grantAccess);
        }
    }
    ...
    function executeByStrategyManager(ExecuteByStrategyManagerParam calldata params)
        external payable virtual override returns (bytes memory result) {
        _strategyManagerPrecheck();
        result = LibExecutor._execute(params.to, 0, params.data, LibExecutor.OP_CALL);
    }
    ...
    function executePayableByStrategyManager(ExecutePayableByStrategyManagerParam
        calldata params) external payable virtual override returns (bytes memory result)
        {
        _strategyManagerPrecheck();
        result = LibExecutor._execute(params.to, params.value, params.data, LibExecutor.
            OP_CALL);
    }
    ...
    function executeBatchByStrategyManager(ExecuteByStrategyManagerParam[] calldata
        params) external payable virtual override returns (bytes[] memory results) {
        _strategyManagerPrecheck();
        results = new bytes[](params.length);
        for(uint256 i = 0; i < params.length; ++i) {
            results[i] = LibExecutor._execute(params[i].to, 0, params[i].data,
                LibExecutor.OP_CALL);
```

```
        }
    }
```

Listing 3.3: Example Privileged Operations in `StrategyAgentAccount`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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 and will be mitigated with the use of a multi-sig to manage the privileged account.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Sake Finance` protocol, which is an innovative platform that introduces the concept of on-chain `AI` agents to the world of decentralized finance and beyond. At its core, `Sake Finance` is designed to empower users with autonomous, intelligent agents capable of executing a broad spectrum of tasks directly on the blockchain. These tasks range from complex financial transactions to dynamic roles within interactive gaming environments, all performed with a level of sophistication and adaptability previously unseen in traditional crypto bots. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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.

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