SMART CONTRACT AUDIT REPORT

for

Creditum & Singularity Swap

Prepared By: Yiqun Chen

PeckShield

November 29, 2021

Document Properties

Client	Creditum & Singularity
Title	Smart Contract Audit Report
Target	Creditum & Singularity Swap
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang, Patrick Liu
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	November 29, 2021	Xuxian Jiang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Creditum` protocol as well as `Singularity Swap`, 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 Creditum & Singularity Swap

`Creditum` is a decentralized, fixed-interest lending/borrowing protocol that allows users to mint stablecoins (`fUSD`) and synthetic tokens through over-collateralized positions. The core system/controller is operated and maintained through governance modules, allowing the system to be paused as well as customizing the stability fees, collateralization ratios, mint limits, and liquidation auction details for supported collaterals. `Singularity Swap` is a decentralized exchange (DEX) that utilizes the popularized automated market maker (AMM) model to create liquidity around paired tokens. By relying on on-chain oracles for token prices, the AMM is able to dynamically shift the concentration of liquidity as the market moves, allowing for more capital efficient liquidity pools and low slippage trades compared to other AMM curves. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocols

Item	Description
Name	Creditum & Singularity
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 29, 2021

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

- https://github.com/xam-darnold/singularity-main.git (cc70657)
- https://github.com/xam-darnold/creditum-fusd.git (9ba9cfa)

And these are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/xam-darnold/singularity-main.git (TBD)
- https://github.com/xam-darnold/creditum-fusd.git (TBD)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 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) [9], 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

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

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.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Creditum` protocol as well as `Singularity Swap`. 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	3	■ ■ ■
Low	4	■ ■ ■ ■
Informational	0	
Total	8	

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, 3 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of Creditum & Singularity Swap Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Fee Computation in SingularityRouter::getAmountOut()	Business Logic	
PVE-002	Medium	Improved Logic of Accumulated Fees in SingularityPairs	Business Logic	
PVE-003	High	Proper WRAPPED_NATIVE Enforcement in SingularityRouter	Business Logic	
PVE-004	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	
PVE-005	Low	Implicit Decimal Assumption in SingularityPair::initialize()	Numeric Errors	
PVE-006	Low	Fork-Compliant Domain Separator in AlpacaStablecoin	Business Logic	
PVE-007	Medium	Proper Liquidity Return in Controller::getPositionData()	Business Logic	
PVE-008	Low	FlashMint Amount Limit in ERC20FlashMint::flashLoan()	Coding Practices	

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.

Description

`Creditum` is a decentralized, fixed-interest lending/borrowing protocol that allows users to mint stablecoins and synthetic tokens through over-collateralized positions. To facilitate its management, the governance modules are designed to customize the stability fees, collateralization ratios, mint limits, as well as liquidation auction details for supported collaterals. In the process of analyzing these protocol-wide risk parameters, we notice one helper function `getPositionData()` can be improved.

To elaborate, we show below the `getPositionData()` routine in the `Controller` contract. This routine computes the user's position data, including current collateral value, debt value, liquidity, shortfall, as well as health factor. One protocol-wide risk parameter `maxDebtRatio` defines the maximum debt ratio, which needs to be honored for the liquidity computation. However, our analysis shows that in the case of `debtValue>maxDebt`, the returned liquidity needs to be 0, instead of current `liquidity = debtValue - maxDebt` (line 428).

```
399     /// @notice Calculates user's position data
400     /// @param user The address of the user
401     /// @param collateral The collateral
402     /// @return (error code, collateral value, debt value, liquidity, shortfall, health
            factor)
403     function getPositionData(address user, address collateral) public view returns (uint
            , uint, uint, uint, uint, uint) {
404         (uint error, uint collateralValue) = getCollateralValue(user, collateral);
405         if (error != uint(Error.NO_ERROR)) {
406             return (error, 0, 0, 0, 0, 0);
407         }
408
409         uint debtValue = getDebtValue(user, collateral);
410         uint maxDebtRatio = collateralData[collateral].maxDebtRatio;
411         uint liquidationThreshold = collateralData[collateral].liquidationThreshold;
412         uint maxDebt = collateralValue * maxDebtRatio / MULTIPLIER;
413
414         uint healthFactor;
415         if (debtValue == 0) {
416             healthFactor = type(uint).max;
417         } else {
418             healthFactor = collateralValue * liquidationThreshold / debtValue;
419         }
420
421         uint liquidity;
422         uint shortfall;
423         if (healthFactor >= MULTIPLIER) {
424             // Use max debt-to-collateral ratio to calculate available borrow
425             if (debtValue <= maxDebt) {
426                 liquidity = maxDebt - debtValue;
427             } else {
428                 liquidity = debtValue - maxDebt;
429             }
430             shortfall = 0;
```

```
        } else {
            // Use liquidation threshold to calculate shortfall
            liquidity = 0;
            shortfall = debtValue − collateralValue ∗ liquidationThreshold / MULTIPLIER;
        }

        return (uint(Error.NO_ERROR), collateralValue, debtValue, liquidity, shortfall,
            healthFactor);
    }
```

Listing 3.8: Controller::getPositionData()

Recommendation Revise the above routine to return the proper liquidity when current debt value is larger than the maximum allowed debt, i.e., `debtValue>maxDebt`.

Status

3.8 FlashMint Amount Limit in ERC20FlashMint::flashLoan()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `ERC20FlashMint`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.7, `Creditum` is a decentralized, fixed-interest lending/borrowing protocol that allows users to mint stablecoins `fUSD` and synthetic tokens through over-collateralized positions. This `fUSD` token contract supports a `flash mint` feature that allows anyone to mint with the one condition that they pay it all back in the same transaction. Our analysis on the `flash mint` feature shows the current implementation can be improved to set a limit on the minted amount.

In the following, we show the `flashLoan()` routine. With it, new tokens are minted and sent to the given receiver, who is expected to own `amount + fee` tokens and have them approved back to the token contract itself so they can be burned. However, there is no limit on the amount that can be minted! It is suggested to have the governance module to set a upper threshold or impose the limitation that the total supply is still valid after the flash mint!

```
    function flashLoan(
        IERC3156FlashBorrower receiver,
        address token,
        uint256 amount,
        bytes calldata data
    ) public virtual override returns (bool) {
```

```
        uint256 fee = flashFee(token, amount);
        _mint(address(receiver), amount);
        require(
            receiver.onFlashLoan(msg.sender, token, amount, fee, data) == _RETURN_VALUE,
            "ERC20FlashMint: invalid return value"
        );
        uint256 currentAllowance = allowance(address(receiver), address(this));
        require(currentAllowance >= amount + fee, "ERC20FlashMint: allowance does not
            allow refund");
        _approve(address(receiver), address(this), currentAllowance - amount - fee);
        _burn(address(receiver), amount + fee);
        return true;
    }
```

Listing 3.9: `ERC20FlashMint::flashLoan()`

Recommendation Revise the above `flashLoan()` function to set a limit on the flash mint amount.

Status

4 Conclusion

In this audit, we have analyzed the design and implementation of `Creditum` protocol as well as `Singularity Swap`. `Creditum` is a decentralized, fixed-interest lending/borrowing protocol that allows users to mint stablecoins (`fUSD`) and synthetic tokens through over-collateralized positions. `Singularity Swap` is a decentralized exchange (DEX) that utilizes the popularized automated market maker (AMM) model to create liquidity around paired tokens. The current code base is well organized and 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.