STARKWARE

Smart Contract Audit Report

Conducted by PeckShield

As part of our due process, we retained PeckShield to audit our smart contracts prior to launching StarkEx, our scalability engine, on Ethereum Mainnet. We chose to work with PeckShield based on warm recommendations, their ongoing public analyses of vulnerabilities on Ethereum, and our interaction with them.

PeckShield has recently conducted their audit over a period of several weeks. Their audit has revealed some minor issues, and the relevant issues were resolved to their satisfaction.

We are happy to share the key findings below, followed by the full report.

Vulnerability Severity Classification

Impact \ Likelihood	High	Medium	Low	
High	Critical	High	Medium	
Medium	High	Medium	Low	
Low	Medium	Low	Low	Informational

Summary

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

STARKWARE

Key Findings

ID	Severity	Title	Status
PVE-001	Low	Broken Link to Actual FactRegistry Implementation	Resolved
PVE-002	Low	Improved Committee Initialization	Resolved

SMART CONTRACT AUDIT REPORT

for

STARKWARE INDUSTRIES LTD.

Prepared By: Yiqun Chen

PeckShield

July 2, 2021

Document Properties

Client	StarkWare Industries Ltd.
Title	Smart Contract Audit Report
Target	StarkEx v3.0
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 2, 2021	Xuxian Jiang	Final Release
1.0-rc1	June 14, 2021	Xuxian Jiang	Release Candidate #1
0.2	June 8, 2021	Xuxian Jiang	Additional Findings #1
0.1	June 1, 2021	Xuxian Jiang	Initial Draft

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

PeckShield

Key Findings

ID	Severity	Title	Status
PVE-001	Low	Broken Link to Actual FactRegistry Implementation	Resolved
PVE-002	Low	Improved Committee Initialization	Resolved

1 Introduction

Given the opportunity to review the design document and related source code of the **StarkEx v3.0** contracts, we in the report outline 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 StarkEx v3.0

`StarkEx` is a `STARK`-powered scalability engine for crypto exchanges. It uses cryptographic proofs to attest to the validity of a batch of transactions (such as trades and transfers) and updates a commitment to the state of the exchange on-chain. `StarkEx` allows an application to significantly scale and improve its offering and is an enabler for a variety of unique applications. There are two versions of `StarkEx`: One for spot trading (`StarkExchange`) and one for derivative trading (`StarkPerpetual`). The first version allows exchanges to provide non-custodial spot trading at scale with high liquidity and lower costs, while the second version expands the support to derivative trading. This audit covers version 3.0 of `StarkEx` with various code-level reorganizations and enhancements including the support of on-chain vaults.

The basic information of StarkEx v3.0 is as follows:

Table 1.1: Basic Information of StarkEx v3.0

Item	Description
Issuer	StarkWare Industries Ltd.
Website	https://starkware.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 2, 2021

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

- https://github.com/starkware-libs/starkex-contracts.git (17e6dd3)

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

- https://github.com/starkware-libs/starkex-contracts.git (e42fede)

1.2 About PeckShield

PeckShield Inc. [5] 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 [4]:

- 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) [3], 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 1.3: 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the StarkEx v3.0 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	0	
Medium	0	
Low	2	
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 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 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of StarkEx v3.0 Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Broken Link To Actual FactRegistry Implementation	Coding Practices	Fixed
PVE-002	Low	Improved Committee Initialization	Coding Practices	Fixed

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.

PeckShield

3 Detailed Results

3.1 Broken Link To Actual FactRegistry Implementation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FactRegistry`
- Category: Coding Practices [2]
- CWE subcategory: CWE-1116 [1]

Description

In StarkEx v3.0, the current implementation follows a so-called `Fact Registry` design pattern that separates cryptographic verification from the business logic of the contract flow. Note that a fact registry holds a hash table of verified "facts" which are represented by a hash of claims. This table may be queried by accessing the `isValid()` function of the registry with a given hash.

While examining the code organization of `Fact Registry`, there is a symbolic file, i.e., `FactRegistry.sol` under the `common-contracts/src/components` subdirectory. This symbolic file is linked to `../../../scalable-dex/contracts/src/components/FactRegistry.sol`, which is also a symbolic file to the final location, i.e., `../../../../../src/starkware/contracts/components/FactRegistry.sol`. The final location unfortunately does not exist in the current repository.

Recommendation Fix the broken link of `FactRegistry.sol`.

Status The issue has been fixed by the following commit: `e42fede`.

3.2 Improved Committee Initialization

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Committee`
- Category: Coding Practices [2]
- CWE subcategory: CWE-1116 [1]

Description

The StarkEx v3.0 protocol defines a standard `Committee` contract, which is preset with an initial set of committee members and the required number of signatures so that a given availability proof can be properly verified.

To elaborate, we show below the `constructor()` routine of the `Committee` contract. This `constructor()` takes an initial set of committee members and the required number of signatures as input and populates its internal states. It comes to our attention that the given committee members can be better evaluated to ensure that `address(0)` is not mistakenly given.

```
    /// @dev Contract constructor sets initial members and required number of signatures.
    /// @param committeeMembers List of committee members.
    /// @param numSignaturesRequired Number of required signatures.
    constructor (address[] memory committeeMembers, uint256 numSignaturesRequired)
        public
    {
        require(numSignaturesRequired <= committeeMembers.length, "TOO_MANY_REQUIRED_SIGNATURES");
        for (uint256 idx = 0; idx < committeeMembers.length; idx++) {
            require(!isMember[committeeMembers[idx]], "NON_UNIQUE_COMMITTEE_MEMBERS");
            isMember[committeeMembers[idx]] = true;
        }
        signaturesRequired = numSignaturesRequired;
    }
```

Listing 3.1: `Committee::constructor()`

We need to emphasize that the current implementation does not pose any security risks even not addressed.

Recommendation Validate the given `committeeMembers` so that only the intended non-`address(0)` members are accepted.

Status The issue has been fixed by the following commit: `e42fede`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `StarkEx v3.0` protocol, which utilizes zkSTARK-based cryptographic proofs to scale up Ethereum on-chain transaction throughputs to support both spot and derivative trading. The system presents a clean and consistent design that makes it distinctive and valuable when compared with current decentralized exchange protocols. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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.

PeckShield

References

[1] MITRE. CWE-1116: Inaccurate Comments. https://cwe.mitre.org/data/definitions/1116.html.

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

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

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

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