STARKWARE

Smart Contract Audit Report

Conducted by PeckShield

As part of our due process, we retained PeckShield to review the design document and related source code of the StarkEx v4.5 contracts. 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 conducted their audit over a period of several weeks. Their audit has revealed three minor issues, and the relevant issue was 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	0
Informational	3
Total	3

Key Findings

ID	Severity	Title	Status	Status
PVE-001	Informational	Consistent Handling of ERC1155 Token Transfer Ins And Outs	Coding Practices	Resolved
PVE-002	Informational	Removal Of Unused State/Code	Coding Practices	Resolved
PVE-003	Informational	Proper Placement of globalConfigCode	Coding Practices	Resolved

SMART CONTRACT AUDIT REPORT

for

StarkEx v4.5.0

Prepared By: Patrick Lou

PeckShield
May 15, 2022

Document Properties

Client	StarkWare Industries Ltd.
Title	Smart Contract Audit Report
Target	StarkEx v4.5.0
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 15, 2022	Xuxian Jiang	Final Release
1.0-rc1	April 8, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Patrick Lou
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 **StarkEx v4.5.0** contracts, 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. This document outlines our audit results.

1.1 About StarkEx v4.5.0

`StarkEx` is an `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 the version 4.5.0 of `StarkEx` with numerous improvements on `ERC1155` as well as certain composite actions. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of StarkEx v4.5.0

Item	Description
Issuer	StarkWare Industries Ltd.
Website	https://starkware.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 15, 2022

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

- https://github.com/starkware-libs/starkex2.0-contracts.git (1ab2620, 90dd00d)

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/starkex2.0-contracts.git (f4ed79b)

1.2 About PeckShield

PeckShield Inc. [6] 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 [5]:

- 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.

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

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) [4], 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 StarkEx v4.5.0 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	0	
Informational	3	■■■
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 3 informational recommendations.

Table 2.1: Key Audit Findings of StarkEx v4.5.0 Protocol

ID	Severity	Title	Category	Status
PVE-001	Informational	Consistent Handling of ERC1155 Token Transfer Ins And Outs	Coding Practices	Resolved
PVE-002	Informational	Removal Of Unused State/Code	Coding Practices	Resolved
PVE-003	Informational	Proper Placement of globalConfigCode	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 Consistent Handling of ERC1155 Token Transfer Ins And Outs

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TokenTransfers`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1099 [1]

Description

The `StarkEx` is an `STARK`-powered scalability engine for crypto exchanges. The audited version enhances the earlier versions with numerous features. While examining a specific feature on the support of `ERC1155` tokens, we notice the current implementation can be improved for consistency.

To elaborate, we show below the related functions `transferInWithTokenId()` and `transferOutWithTokenId()`. As the names indicate, they are designed to transfer in or out `ERC1155` or `ERC721` tokens. It comes to our attenion that there exists certain (minor) inconsistency. In particular, the first function makes an early exist when the given `quantizedAmount` is 0, which is missing in the second function. For consistency, we can arrange the same early exit in the second function.

```
    function transferInWithTokenId(
        uint256 assetType,
        uint256 tokenId,
        uint256 quantizedAmount
    ) internal override {
        require(isAssetTypeWithTokenId(assetType), "FUNGIBLE_ASSET_TYPE");
        if (quantizedAmount == 0) return;

        if (isERC721(assetType)) {
            require(quantizedAmount == 1, "ILLEGAL_NFT_BALANCE");
            transferInNft(assetType, tokenId);
        } else {
            transferInSft(assetType, tokenId, quantizedAmount);
```

```
        }
    }
```

Listing 3.1: `transferInWithTokenId()`

```
    function transferOutWithTokenId(
        address recipient,
        uint256 assetType,
        uint256 tokenId,
        uint256 quantizedAmount
    ) internal override {
        require(isAssetTypeWithTokenId(assetType), "FUNGIBLE_ASSET_TYPE");
        if (isERC721(assetType)) {
            require(quantizedAmount == 1, "ILLEGAL_NFT_BALANCE");
            transferOutNft(recipient, assetType, tokenId);
        } else {
            transferOutSft(recipient, assetType, tokenId, quantizedAmount);
        }
    }
```

Listing 3.2: `transferInWithTokenId()`

Similarly, when analyzing two other related functions `depositWithTokenIdReclaim()` and `depositNftReclaim()`, we notice both functions take four arguments, i.e., `starkKey`, `assetType`, `tokenId`, and `vaultId`. However, the first function takes the four arguments in the order of `starkKey`, `assetType`, `tokenId`, and `vaultId` while the second function takes the order of `starkKey`, `assetType`, `vaultId`, and `tokenId`. For improved readability and maintenance, we suggest to make the same ordering of these arguments.

Recommendation Resolve the unnecessary inconsistency in the afore-mentioned functions.

Status The issue has been resolved in the following commit: `f91c804`.

3.2 Removal Of Unused State/Code

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GenericGovernance`, `NamedStorage`
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [2]

Description

The `StarkEx` protocol makes good use of a number of reference contracts, such as `ERC20` and `ERC721`, to facilitate its code implementation and organization. For example, the `Deposits` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `NamedStorage` library contract, there are a number of functions that are defined, but not used. This library provides a clean abstraction in accessing basic low-level basic storage, in storage locations out of the low linear address space. However, this entire library contract is not used in current code base and is therefore suggested for removal.

```
library NamedStorage {
    function bytes32ToUint256Mapping(string memory tag_)
        internal
        pure
        returns (mapping(bytes32 => uint256) storage randomVariable)
    {
        bytes32 location = keccak256(abi.encodePacked(tag_));
        assembly {
            randomVariable_slot := location
        }
    }
    ...
}
```

Listing 3.3: The `NamedStorage` Contract

Similarly, there exists another contract `GenericGovernance`, which is not used in the current protocol and can be safely removed.

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The team clarifies that the above-mentioned two contract files are in use by other components that are not included in this audit (`StarkNet` etc.)

3.3 Proper Placement of globalConfigCode

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MainStorage`
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [2]

Description

The audited `StarkEx` version introduces a new global state `globalConfigCode`. Our analysis shows this new global state is only used in `StarkExchange`. As mentioned earlier, there are two versions of `StarkEx`: One for spot trading (`StarkExchange`) and one for derivative trading (`StarkPerpetual`). The use of globalConfigCode only in StarkExchange indicates that it should be included inside StarkExchange, instead of the current MainStorage (that also serves StarkPerpetual).

```
    // Mapping for timelocked actions.
    // A actionKey => activation time.
    mapping(bytes32 => uint256) actionsTimeLock;

    // Append only list of requested forced action hashes.
    bytes32[] actionHashList;
    // ---- END OF MAIN STORAGE AS DEPLOYED IN STARKEX3.0 ----
    // ---- END OF MAIN STORAGE AS DEPLOYED IN STARKEX4.0 ----

    // Rollup Vaults Tree Root & Height.
    uint256 rollupVaultRoot; // NOLINT: constable-states uninitialized-state.
    uint256 rollupTreeHeight; // NOLINT: constable-states uninitialized-state.

    uint256 globalConfigCode; // NOLINT: constable-states uninitialized-state.
```

Listing 3.4: The MainStorage Contract

Recommendation Relocate the new state globalConfigCode to the StarkExState data structure, instead of the current MainStorage.

Status This issue has been confirmed. The team indicates that while this globalConfigCode is currently used only on StarkExchange, there is a plan to use this item in future versions of StarkPerpetual.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `StarkEx v4.5.0` 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.

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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