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 2.0, the next version of our scalability engine, on Ethereum Mainnet.

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	**1**
Low	**1**
Informational	**8**
Total	**10**

Key Findings

ID	Severity	Title	Status
PVE-001	Low	Incompatibility with Deflationary/Rebasing Tokens	CONFIRMED
PVE-002	Info.	Unnecessary Zero Amount Transfers	CONFIRMED
PVE-003	Info.	Improved Gas Consumption by Removing Unused Storage	FIXED
PVE-004	Info.	Unused Internal Functions	FIXED
PVE-005	Info.	Unused Interfaces	FIXED
PVE-006	Info.	Missed Sanity Checks While Calling Token Contracts	FIXED
PVE-007	Info.	Redundant Sanity Checks	FIXED
PVE-008	Info.	Redundant Timestamp Checks	FIXED
PVE-009	Info.	Improved Ether Transfers	FIXED
PVE-010	Medium	Denial-of-Service Risks in depositCancel()	FIXED
PVE-011	Info.	Typos in Comments	FIXED

SMART CONTRACT AUDIT REPORT

for

STARKWARE INDUSTRIES LTD.

Prepared By: Shuxiao Wang

Hangzhou, China
Oct. 26, 2020

Document Properties

Client	StarkWare Industries Ltd.
Title	Smart Contract Audit Report
Target	StarkEx V2
Version	1.0
Author	Chiachih Wu
Auditors	Chiachih Wu, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Oct. 26, 2020	Chiachih Wu	Final Release
1.0-rc	Oct. 19, 2020	Chiachih Wu	Release Candidate

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related source code of the **StarkEx V2** 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 V2

StarkEx is StarkWare's Layer-2 scalability engine. StarkEx leverages STARK technology to power scalable self-custodial transactions (trading & payments) for applications such as DeFi and gaming. StarkEx allows an application to significantly scale and improve its offering and to bring in new business. StarkEx V2 runs over Cairo, StarkWare's Turing-complete framework for STARKs, and includes new features such as Fast Withdrawals: withdraw funds from L2 to any L1 address in blockchain-time, ERC-721 support and more.

The basic information of StarkEx V2 is as follows:

Table 1.1: Basic Information of StarkEx V2

Item	Description
Issuer	StarkWare Industries Ltd.
Website	https://starkware.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Oct. 26, 2020

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

- https://github.com/starkware-libs/starkex-contracts (8d596dd)
- https://github.com/starkware-libs/starkex-contracts (e4c1faa)
- https://github.com/starkware-libs/starkex-contracts (2799231)

1.2 About PeckShield

PeckShield Inc. [13] 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 [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.

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

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) [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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 V2 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	1	■
Low	1	■
Informational	8	■■■■■■■■
Total	10	

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, 8 informational recommendations.

Table 2.1: Key Audit Findings of StarkEx V2 Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logics	Fixed
PVE-002	Info.	Unnecessary Zero Amount Transfers	Business Logics	Confirmed
PVE-003	Info.	Improved Gas Consumption by Removing Unused Storage	Business Logics	Fixed
PVE-004	Info.	Unused Internal Functions	Coding Practices	Fixed
PVE-005	Info.	Unused Interfaces	Coding Practices	Fixed
PVE-006	Info.	Missed Sanity Checks While Calling Token Contracts	Business Logics	Fixed
PVE-007	Info.	Redundant Sanity Checks	Business Logics	Fixed
PVE-008	Info.	Improved Ether Transfers	Business Logics	Fixed
PVE-009	Medium	Denial-of-Service Risks in depositCancel()	Business Logics	Fixed
PVE-010	Info.	Typos in Comments	Coding Practices	Fixed

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 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LendingPool
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

In StarkEx V2, the `Deposits` and `Withdrawals` contracts are designed to be the main entries for interacting with users. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets. Naturally, the `Tokens` contract implements a number of low-level helper routines to transfer assets into the `Deposits` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
function deposit(
    uint256 starkKey,
    uint256 assetType,
    uint256 vaultId,
    uint256 quantizedAmount
) public notFrozen()
{
    // No need to verify amount > 0, a deposit with amount = 0 can be used to undo
        cancellation.
    require(vaultId <= MAX_VAULT_ID, "OUT_OF_RANGE_VAULT_ID");
    // starkKey must be registered.
    require(ethKeys[starkKey] != ZERO_ADDRESS, "INVALID_STARK_KEY");
    require(!isMintableAssetType(assetType), "MINTABLE_ASSET_TYPE");
    uint256 assetId = assetType;

    // Update the balance.
    pendingDeposits[starkKey][assetId][vaultId] += quantizedAmount;
    require(
```

```
        pendingDeposits[starkKey][assetId][vaultId] >= quantizedAmount,
        "DEPOSIT_OVERFLOW"
    );

    // Disable the timeout.
    delete cancellationRequests[starkKey][assetId][vaultId];

    // Transfer the tokens to the Deposit contract.
    transferIn(assetType, quantizedAmount);
```

Listing 3.1: interactions/Deposits.sol

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of StarkEx V2 and affects protocol-wide operation and maintenance.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the `Deposits` before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into StarkEx V2. In StarkEx V2, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

We emphasize that the current deployment of `Deposits` is safe as it uses whitelisted `assetType` for deposits and withdrawals. However, the current code implementation is generic in supporting various tokens and there is a need to highlight the possible pitfall from the audit perspective.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transferIn()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been addressed by checking the balance before and after within the `TransferIn()` function in commit 2799231.

3.2 Unnecessary Zero Amount Transfers

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Withdrawals.sol`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

In the `Withdrawals` contract, the `withdrawTo()` function allows users to withdraw assets by claiming the `pendingWithdrawals[starkKey][assetId]` with a valid `startKey` associated with the `msg.sender`. While reviewing the implementation, we identify that certain corner cases may lead to zero amount transfers with `LogWithdrawalPerformed()` events emitted, which is not necessary.

```
93 function withdrawTo(uint256 starkKey, uint256 assetType, address payable recipient)
94     public
95     isSenderStarkKey(starkKey)
96 // No notFrozen modifier: This function can always be used, even when frozen.
97 {
98     require(!isMintableAssetType(assetType), "MINTABLE_ASSET_TYPE");
99     uint256 assetId = assetType;
100     // Fetch and clear quantized amount.
101     uint256 quantizedAmount = pendingWithdrawals[starkKey][assetId];
102     pendingWithdrawals[starkKey][assetId] = 0;

104     // Transfer funds.
105     transferOut(recipient, assetType, quantizedAmount);
106     emit LogWithdrawalPerformed(
107         starkKey,
108         assetType,
109         fromQuantized(assetType, quantizedAmount),
110         quantizedAmount,
111         recipient
112     );
113 }
```

Listing 3.2: interactions/Withdrawals.sol

Specifically, when `pendingWithdrawals[starkKey][assetId] == 0`, the `transferOut()` call in line 105 results in a zero transfer since `quantizedAmount` is 0. In addition, line 106 emits the `LogWithdrawalPerformed` event with the zero `quantizedAmount`, which is a waste of gas.

Recommendation Add `pendingWithdrawals[starkKey][assetId] > 0` sanity check into `withdrawTo()`.

```
function withdrawTo(uint256 starkKey, uint256 assetType, address payable recipient)
    public
```

```
    isSenderStarkKey(starkKey)
// No notFrozen modifier: This function can always be used, even when frozen.
{
    require(!isMintableAssetType(assetType), "MINTABLE_ASSET_TYPE");
    uint256 assetId = assetType;
    require(pendingWithdrawals[starkKey][assetId] > 0, "ZERO_AMOUNT_WITHDRAW");
    // Fetch and clear quantized amount.
    uint256 quantizedAmount = pendingWithdrawals[starkKey][assetId];
    pendingWithdrawals[starkKey][assetId] = 0;

    // Transfer funds.
    transferOut(recipient, assetType, quantizedAmount);
    emit LogWithdrawalPerformed(
        starkKey,
        assetType,
        fromQuantized(assetType, quantizedAmount),
        quantizedAmount,
        recipient
    );
}
```

Listing 3.3: interactions/Withdrawals.sol

Status This issue has been confirmed. Considering this is an unlikely case, the team decides to leave it as is for the time being.

3.3 Improved Gas Consumption by Removing Unused Storage

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ApprovalChain`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

In the `ApprovalChain` contract, when a `governor` tends to remove an entry from the `ApprovalChain`, she needs to invoke the `announceRemovalIntent()` first to set the `chain.unlockedForRemovalTime[entry]`. It means the `governor` cannot literally remove the entry until `now` reaches `now + removalDelay`.

```
function announceRemovalIntent(
    StarkExTypes.ApprovalChainData storage chain, address entry, uint256 removalDelay)
    internal
    onlyGovernance()
    notFrozen()
{
    safeFindEntry(chain.list, entry);
```

```
    require(now + removalDelay > now, "INVALID_REMOVAL_DELAY"); // NOLINT: timestamp.
    // solium-disable-next-line security/no-block-members
    chain.unlockedForRemovalTime[entry] = now + removalDelay;
}
```

Listing 3.4: components/ApprovalChain.sol

To achieve that, the `removeEntry()` function checks the `chain.unlockedForRemovalTime[entry]` when the `governor` removes the entry for real. However, in the case that the `governor` successfully remove the specific entry from `chain.list`, the no longer needed `chain.unlockedForRemovalTime[entry]` is not cleared. Fortunately, the uncleared removal time could not be re-used since `safeFindEntry()` would revert in the beginning of `removeEntry()`. But it's worth to remove the unused storage to refund some gas.

```
function removeEntry(StarkExTypes.ApprovalChainData storage chain, address entry)
    internal
    onlyGovernance()
    notFrozen()
{
    address[] storage list = chain.list;
    // Make sure entry exists.
    uint256 idx = safeFindEntry(list, entry);
    uint256 unlockedForRemovalTime = chain.unlockedForRemovalTime[entry];

    // solium-disable-next-line security/no-block-members
    require(unlockedForRemovalTime > 0, "REMOVAL_NOT_ANNOUNCED");
    // solium-disable-next-line security/no-block-members
    require(now >= unlockedForRemovalTime, "REMOVAL_NOT_ENABLED_YET"); // NOLINT:
        timestamp.

    uint256 n_entries = list.length;

    // Removal of last entry is forbidden.
    require(n_entries > 1, "LAST_ENTRY_MAY_NOT_BE_REMOVED");

    if (idx != n_entries - 1) {
        list[idx] = list[n_entries - 1];
    }
    list.pop();
}
```

Listing 3.5: components/ApprovalChain.sol

Recommendation Remove `chain.unlockedForRemovalTime[entry]` in `removeEntry()`.

Status This issue has been addressed by deleting `chain.unlockedForRemovalTime[entry]` in commit 2799231.

3.4 Unused Internal Functions

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Deposits
- Category: Coding Practices [5]
- CWE subcategory: CWE-1116 [3]

Description

In StarkEx V2, after users deposit assets into the on-chain deposit area, the UpdateState contract is in charge to transfer funds to the off-chain deposit area by invoking the acceptDeposit() function which is implemented in the AcceptModifications contract. However, we notice that there is another acceptDeposit() internal function in the Deposits contract which is not used anywhere.

```
function acceptDeposit(
    uint256 starkKey,
    uint256 vaultId,
    uint256 assetId,
    uint256 quantizedAmount
)
    internal
{
    // Fetch deposit.
    require(
        pendingDeposits[starkKey][assetId][vaultId] >= quantizedAmount,
        "DEPOSIT_INSUFFICIENT"
    );

    // Subtract accepted quantized amount.
    pendingDeposits[starkKey][assetId][vaultId] -= quantizedAmount;
}
```

Listing 3.6: interactions/Deposits.sol

Recommendation Remove the unused acceptDeposit() function from the Deposits contract.

Status This issue has been addressed by removing acceptDeposit() function from the Deposits contract in commit 2799231.

3.5 Unused Interfaces

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: MWithdrawal
- Category: Coding Practices [5]
- CWE subcategory: CWE-1116 [3]

Description

In StarkEx V2, after users issue the withdrawal requests, the `UpdateState` contract is in charge to transfer funds from the off-chain area to the on-chain withdrawal area by invoking the `acceptWithdrawal()` function which is implemented in the `AcceptModifications` contract. The underlying function of `acceptWithdrawal()` is the internal function `allowWithdrawal()` implemented in `AcceptModifications` contract as well. However, we notice that there is another `allowWithdrawal()` interface declared in the `MWithdrawal` contract which is not used anywhere. In addition, the `MWithdrawal` contract seems to be a obsolete contract left in the code base as no other contract imports it.

```
function allowWithdrawal(
    uint256 starkKey,
    uint256 assetId,
    uint256 quantizedAmount
)
    internal;
```

Listing 3.7: interfaces /MWithdrawal.sol

Recommendation Remove the unused `MWithdrawal` contract.

Status This issue has been addressed by removing the `MWithdrawal` contract in commit 2799231.

3.6 Missed Sanity Checks While Calling Token Contracts

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Tokens
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

In StarkEx V2, assets are transfer in/out with helper functions such as `transferIn()` and `transferOut()`. While dealing with ERC20s, the `safeTokenContractCall()` helper function is used to deal with non-standard ERC20 implementations. However, the `safeTokenContractCall()` function fails to check if the `tokenAddress` is a contract or not. When the `tokenAddress` happens to be an EOA address, the `call()` returns 0 as well. This leads to a buggy implementation of token transfers.

```
function safeTokenContractCall(address tokenAddress, bytes memory callData) internal {
    // solium-disable-next-line security/no-low-level-calls
    // NOLINTNEXTLINE: low-level-calls.
    (bool success, bytes memory returndata) = address(tokenAddress).call(callData);
    require(success, string(returndata));

    if (returndata.length > 0) {
        require(abi.decode(returndata, (bool)), "TOKEN_OPERATION_FAILED");
    }
}
```

Listing 3.8: components/Tokens.sol

Fortunately, the `tokenAddress` is checked while an asset is registered into the system. There's no plausible way to exploit this issue.

Recommendation Ensure `tokenAddress` is a contract before `call()` it.

Status This issue has been fixed by checking `tokenAddress` with `isContract()` in commit 2799231.

3.7 Redundant Sanity Checks

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `gps/GpsStatementVerifier.sol`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

While reviewing the EVM Verifier of StarkEx V2, we identify a redundant sanity check in the `GpsStatementVerifie` contract. Specifically, in the `verifyProofAndRegister()` function, the `offset` is set as `OFFSET_PUBLIC_MEMORY` in line 72. However, in line 77, the `require()` call checks the `offset` immediately, which is redundant. The `offset == OFFSET_PUBLIC_MEMORY` here should always be true.

```
72          uint256 offset = OFFSET_PUBLIC_MEMORY;

74          // Write public memory, which is a list of pairs (address, value).
75          {
76          // Program segment.
77          require(offset == OFFSET_PUBLIC_MEMORY, "Wrong value of offset.");
```

```
        uint256[PROGRAM_SIZE] memory bootloaderProgram =
            bootloaderProgramContractAddress.getCompiledProgram();
        for (uint256 i = 0; i < bootloaderProgram.length; i++) {
            cairoPublicInput[offset] = i;
            cairoPublicInput[offset + 1] = bootloaderProgram[i];
            offset += 2;
        }
        }
```

Listing 3.9: GpsStatementVerifier :: verifyProofAndRegister ()

Recommendation Remove the redundant sanity check against `offset`.

Status This issue has been fixed by refactoring the `verifyProofAndRegister()` function in commit e4c1faa.

3.8 Improved Ether Transfers

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TransferRegistry`, `Tokens`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

As described in Section 3.6, assets are transfer in/out with helper functions such as `transferIn()` and `transferOut()`. While dealing with ERC20s, the `safeTokenContractCall()` helper function is used to deal with non-standard ERC20 implementations. As for the case of transferring ether, the Solidity function, `transfer()`, is used (line 213 in the code snippet below). However, as described in [2], when the `recipient` happens to be a contract which implements a callback function containing EVM instructions such as `SLOAD`, the 2300 gas supplied with `transfer()` might be insufficient, leading to an out-of-gas error.

```
200     function transferOut(address payable recipient, uint256 assetType, uint256
            quantizedAmount)
201         internal {
202         bytes memory assetInfo = getAssetInfo(assetType);
203         uint256 amount = fromQuantized(assetType, quantizedAmount);

205         bytes4 tokenSelector = extractTokenSelector(assetInfo);
206         if (tokenSelector == ERC20_SELECTOR) {
207             address tokenAddress = extractContractAddress(assetInfo);
208             safeTokenContractCall(
209                 tokenAddress,
210                 abi.encodeWithSelector(IERC20(0).transfer.selector, recipient, amount)
```

```
            );
        } else if (tokenSelector == ETH_SELECTOR) {
            recipient.transfer(amount); // NOLINT: arbitrary-send.
        } else {
            revert("UNSUPPORTED_TOKEN_TYPE");
        }
    }
```

Listing 3.10: components/Tokens.sol

As suggested in [2], we suggest to stop using Solidity's `transfer()` as well. Note that the use of `call()` leads to side effects such as reentrancy attacks and gas token vulnerabilities.

Recommendation Replace `transfer()` with `call()`.

Status This issue has been fixed by introducing the `performEthTransfer()` helper function which uses `call()` for ether transfers in commit 2799231.

3.9 Denial-of-Service Risks in depositCancel()

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Deposits`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

As described in Section 3.1, `deposit()` is the entry routine which accepts user deposits. To handle the case that users tend to undo the `deposit()` operations, the `depositCancel()` allows users to issue the cancellation with the current timestamp (i.e., `now`) stored in the `cancellationRequests` (line 201 in the code snippet below).

```
188 function depositCancel(
189     uint256 starkKey,
190     uint256 assetId,
191     uint256 vaultId
192 )
193     external
194     isSenderStarkKey(starkKey)
195 // No notFrozen modifier: This function can always be used, even when frozen.
196 {
197     require(vaultId <= MAX_VAULT_ID, "OUT_OF_RANGE_VAULT_ID");
198 
199     // Start the timeout.
200     // solium-disable-next-line security/no-block-members
```

```
    cancellationRequests[starkKey][assetId][vaultId] = now;
```

Listing 3.11: interactions/Deposits.sol

The user cannot `depositReclaim()` the deposit until `DEPOSIT_CANCEL_DELAY` (i.e., 24 hours) after the previously stored timestamp. As shown in the code snippet below, the book-keeping record, `cancellationRequests[starkKey][assetId][vaultId]` is retrieved into `requestTime` in line 220. Later on, the `freetime` is derived by $requestTime + 24hours$ in line 222. If `now` is greater or equal to `freetime`, the `transferOut()` is invoked to transfer assets to `msg.sender` (line 233). So far, the business logic seems solid.

```
207 function depositReclaim(
208     uint256 starkKey,
209     uint256 assetType,
210     uint256 vaultId
211 )
212     external
213     isSenderStarkKey(starkKey)
214 // No notFrozen modifier: This function can always be used, even when frozen.
215 {
216     require(vaultId <= MAX_VAULT_ID, "OUT_OF_RANGE_VAULT_ID");
217     uint256 assetId = assetType;
218
219     // Make sure enough time has passed.
220     uint256 requestTime = cancellationRequests[starkKey][assetId][vaultId];
221     require(requestTime != 0, "DEPOSIT_NOT_CANCELED");
222     uint256 freeTime = requestTime + DEPOSIT_CANCEL_DELAY;
223     assert(freeTime >= DEPOSIT_CANCEL_DELAY);
224     // solium-disable-next-line security/no-block-members
225     require(now >= freeTime, "DEPOSIT_LOCKED"); // NOLINT: timestamp.
226
227     // Clear deposit.
228     uint256 quantizedAmount = pendingDeposits[starkKey][assetId][vaultId];
229     delete pendingDeposits[starkKey][assetId][vaultId];
230     delete cancellationRequests[starkKey][assetId][vaultId];
231
232     // Refund deposit.
233     transferOut(msg.sender, assetType, quantizedAmount);
```

Listing 3.12: interactions/Deposits.sol

But here comes the flawed business logic. In the `deposit()` function, we notice that the book-keeping record is cleared when the user re-deposit some assets. It is reasonable to cancel a previous cancellation if the user tend to deposit assets again. However, any user could `deposit()` to an arbitrary `(starkKey, assetType, vaultId)` tuple with zero amount as the comment suggested in line 148. Therefore, a bad actor could maliciously cancel a victim's deposit cancellation by front-running the `depositReclaim()` calls. This leads to a denial-of-service vulnerability targeting the `depositCancel()` and `depositReclaim()` mechanism.

```
function deposit(
    uint256 starkKey,
    uint256 assetType,
    uint256 vaultId,
    uint256 quantizedAmount
) public notFrozen()
{
    // No need to verify amount > 0, a deposit with amount = 0 can be used to undo
        cancellation.
    require(vaultId <= MAX_VAULT_ID, "OUT_OF_RANGE_VAULT_ID");
    // starkKey must be registered.
    require(ethKeys[starkKey] != ZERO_ADDRESS, "INVALID_STARK_KEY");
    require(!isMintableAssetType(assetType), "MINTABLE_ASSET_TYPE");
    uint256 assetId = assetType;

    // Update the balance.
    pendingDeposits[starkKey][assetId][vaultId] += quantizedAmount;
    require(
        pendingDeposits[starkKey][assetId][vaultId] >= quantizedAmount,
        "DEPOSIT_OVERFLOW"
    );

    // Disable the timeout.
    delete cancellationRequests[starkKey][assetId][vaultId];
}
```

Listing 3.13: interactions/Deposits.sol

Recommendation Ensure `isSenderStarkKey(starkKey)` before entering `Deposit()`. However, this breaks the business logic of depositing to an arbitrary `(starkKey, assetType, vaultId)` tuple.

Status This issue has been fixed by checking the `msg.sender` in `Deposit()` before removing the `cancellationRequests` entry in commit 2799231, which preserves the `deposit()` business logic mentioned above.

3.10 Typos in Comments

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target:
- Category: Coding Practices [5]
- CWE subcategory: CWE-1116 [3]

Description

While reviewing the StarkEx V2 codebase, we occasionally identify typos in the comments around the Solidity code. Here we list some cases only. The complete list of typos has been sent to the team in a separate file.

Case I The comments in line 36 of the `ApprovalChain` contract: "in te chain"

```
20 function addEntry(
21     StarkExTypes.ApprovalChainData storage chain,
22     address entry, uint256 maxLength, string memory identifier)
23     internal
24     onlyGovernance()
25     notFrozen()
26 {
27     address[] storage list = chain.list;
28     require(entry.isContract(), "ADDRESS_NOT_CONTRACT");
29     bytes32 hash_real = keccak256(abi.encodePacked(Identity(entry).identify()));
30     bytes32 hash_identifier = keccak256(abi.encodePacked(identifier));
31     require(hash_real == hash_identifier, "UNEXPECTED_CONTRACT_IDENTIFIER");
32     require(list.length < maxLength, "CHAIN_AT_MAX_CAPACITY");
33     require(findEntry(list, entry) == ENTRY_NOT_FOUND, "ENTRY_ALREADY_EXISTS");
34 
35     // Verifier must have at least one fact registered before adding to chain,
36     // unless it's the first verifier in te chain.
```

Listing 3.14: components/ApprovalChain.sol

Case II The comments in line 26 of the `FactRegistry` contract: "But the check is against the local fact registry,"

```
23 /*
24   This is an internal method to check if the fact is already registered.
25   In current implementation of FactRegistry it's identical to isValid().
26   But the check is against the local fact registry,
27   So for a derived referral fact registry, it's not the same.
28 */
29 function _factCheck(bytes32 fact)
30     internal view
31     returns(bool)
32 {
33     return verifiedFact[fact];
34 }
```

Listing 3.15: FactRegistry.sol

Case III The comments in line 7 – 8 of the `GpsFactRegistryAdapter` contract: "The GpsFactRegistryAdapter contract is used as an adpater between a Dapp contact and a GPS fact registry. An isValid(fact) query is answered by querying the GPS contract about"

```
/*
  The GpsFactRegistryAdapter contract is used as an adpater between a Dapp contact and a
      GPS fact
  registry. An isValid(fact) query is answered by querying the GPS contract about
  new_fact := keccak256(programHash, fact).

```

```
  The goal of this contract is to simplify the verifier upgradeability logic in the Dapp
      contract
  by making the upgrade flow the same regardless of whether the update is to the program
      hash or
  the gpsContractAddress.
*/
contract GpsFactRegistryAdapter is IQueryableFactRegistry, Identity {
```

Listing 3.16: GpsFactRegistryAdapter.sol

Recommendation Perform spell check on the comments.

Status This issue has been addressed in commit 2799231.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the StarkEx V2 protocol, which utilizes zkSTARK-based cryptographic proofs to scale up Ethereum on-chain transaction throughputs. 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.

5 | Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [9, 10, 11, 12, 14].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [15] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] Steve Marx. Stop Using Solidity's transfer() Now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/.

[3] MITRE. CWE-1116: Inaccurate Comments. https://cwe.mitre.org/data/definitions/1116.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 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. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[10] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[11] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[12] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[14] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[15] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.