SMART CONTRACT AUDIT REPORT

for

STARKWARE INDUSTRIES LTD.

Prepared By: Shuxiao Wang

Hangzhou, China
Feb. 26, 2020

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	Feb. 26, 2020	Chiachih Wu	Final Release
1.0-rc2	Feb. 26, 2020	Chiachih Wu	Minor Revise
1.0-rc1	Feb. 17, 2020	Chiachih Wu	Status Update
0.4	Jan. 21, 2020	Chiachih Wu	Add More Findings
0.3	Jan. 14, 2020	Chiachih Wu	Add More Findings
0.2	Jan. 7, 2020	Chiachih Wu	Add More Findings
0.1	Dec. 31, 2019	Chiachih Wu	Initial Draft

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 **StarkEx** design document and related smart contract source code, 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

StarkEx is a `zkSTARK`-powered scalability engine, which makes essential use of cryptographic proofs to attest to the validity of a batch of ramp and trade transactions. The attestation allows for ensuring the state consistency between the scalable off-chain, transaction-processing exchange service and the on-chain DEX with transaction commitment (or finality). With that, StarkEx enables next-generation exchanges that provide non-custodial trading at an unprecedented scale with high liquidity and low costs.

The basic information of StarkEx is as follows:

Table 1.1: Basic Information of StarkEx

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

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

- https://github.com/starkware-industries/starkex (7a8d0da)
- https://github.com/starkware-libs/starkex-contracts (d6bde00)

1.2 About PeckShield

PeckShield Inc. [24] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [19]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [18], 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 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 an 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 implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	3	
Informational	8	
Total	12	

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 need to be brought up and paid more attention to, which are categorized in the above table. All of the issues have be resolved, except the two inconsequential ones. 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, 3 low-severity vulnerabilities, and 8 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Info.	Implicit Assumption of FullWithdrawalRequests	Arg.s and Parameters	Resolved
PVE-002	Info.	Inconsistent Uses of SafeMath	Coding Practices	Confirmed
PVE-003	Info.	Potential Integer Overflow in ApprovalChain	Numeric Errors	Resolved
PVE-004	Low	Possible Denial-of-Service in Registration	Business Logic Errors	Resolved
PVE-005	Info.	Misleading Comments about MVerifiers	Coding Practices	Resolved
PVE-006	Low	Business Logic Inconsistency in Committee	Behavioral Issues	Resolved
PVE-007	Info.	starkKey, vaultId, tokenId Ordering	Coding Practices	Resolved
PVE-008	Info.	Redundant Timestamp Checks	Coding Practices	Confirmed
PVE-009	Low	Upgrades Depend on States of Old Versions in Proxy	Data Integrity Issues	Resolved
PVE-010	Info.	Optimization Suggestions to Proxy	Coding Practices	Resolved
PVE-011	Info.	Optimization Suggestions to DexStatementVerifier	Coding Practices	Resolved
PVE-012	Medium	Possible Integer Overflow in DexStatementVerifier	Numeric Errors	Resolved

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Implicit Assumption of FullWithdrawalRequests

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `interactions/UpdateState.sol`
- Category: Arg.s and Parameters [17]
- CWE subcategory: CWE-628 [10]

Description

The StarkEx contract keeps track of the execution state of the off-chain exchange service by storing Merkle roots of the vault state (off-chain account state) and the order state (including fully executed and partially fulfilled orders). It is achieved by the operator responsibly initiating a series of state-updating operations, i.e., `updateState`. However, the implementation has an implicit assumption that is not documented or evident from the codebase. This assumption, if non-present, could lead to to unauthorized removal of legitimate full withdraw requests.

Specifically, when an operator performs `updateState`, this operation requires two parameters as its input: `publicInput` and `applicationData`. Note the first parameter is properly verified by both `Integrity Verifiers` and `Availability Verifiers`. But the second parameter is blindly trusted and may be misused for unintended uses, if the above-mentioned assumption is not present.

```
    function performUpdateState(
        uint256[] memory publicInput,
        uint256[] memory applicationData
    )
        internal
    {
        rootUpdate(
            publicInput[OFFSET_VAULT_INITIAL_ROOT],
            publicInput[OFFSET_VAULT_FINAL_ROOT],
            publicInput[OFFSET_ORDER_INITIAL_ROOT],
            publicInput[OFFSET_ORDER_FINAL_ROOT],
            publicInput[OFFSET_VAULT_TREE_HEIGHT],
```

```
            publicInput[OFFSET_ORDER_TREE_HEIGHT]
        );
        sendModifications(publicInput, applicationData);
    }
```

Listing 3.1: interactions/UpdateState.sol

In particular, the second parameter `applicationData` is directly passed to the `performUpdateState()` routine, and then further dribbled down to `sendModifications()`. In the following, we list the related code snippets.

```
124         for (uint256 i = 0; i < nModifications; i++) {
125             uint256 modificationOffset = OFFSET_MODIFICATION_DATA + i *
                    N_WORDS_PER_MODIFICATION;
126             uint256 starkKey = publicInput[modificationOffset];
127             uint256 requestingKey = applicationData[i + 1];
128             uint256 tokenId = publicInput[modificationOffset + 1];
129 
130             require(starkKey < K_MODULUS, "Stark key >= PRIME");
131             require(requestingKey < K_MODULUS, "Requesting key >= PRIME");
132             require(tokenId < K_MODULUS, "Token id >= PRIME");
133 
134             uint256 actionParams = publicInput[modificationOffset + 2];
135             uint256 amountBefore = (actionParams >> 192) & ((1 << 63) - 1);
136             uint256 amountAfter = (actionParams >> 128) & ((1 << 63) - 1);
137             uint256 vaultId = (actionParams >> 96) & ((1 << 31) - 1);
138 
139             if (requestingKey != 0) {
140                 // This is a false full withdrawal.
141                 require(
142                     starkKey != requestingKey,
143                     "False full withdrawal requesting_key should differ from the vault
                        owner key.");
144                 require(amountBefore == amountAfter, "Amounts differ in false full
                    withdrawal.");
145                 clearFullWithdrawalRequest(requestingKey, vaultId);
146                 continue;
147             }
```

Listing 3.2: interactions/UpdateState.sol

Notice that `requestingKey` is directly derived from `applicationData` (line 127.): `requestingKey = applicationData[i + 1]`. Its validity needs to be properly checked before the sensitive function (line 145), i.e., `clearFullWithdrawalRequest(requestingKey, vaultId)`, is invoked. However, current implementation does not enforce strict verification. If a false full withdrawal might be crafted to bear the same `vaultId` with a legitimate full withdrawal request, the legitimate request can then be cleared. To block this, the system has an implicit assumption that at any given time there is a unique mapping from `vaultId` to `starkKey` and that <`vaultId`, `starkKey`> pairs in the public input always correspond to the real mapping.

We highlight this particular assumption is essential and needs to be be explicitly documented. If without this assumption, assuming the operator may be fully compromised, legitimate full withdrawal requests can always be cleared. Here is a possible attack scenario:

1. A normal user `Alice` submits a legitimate `fullWithdrawalRequest`, say R_{alice}, with $vaultId_{Alice}$ as its `vaultId` argument. Internally, the contract records the request in the following data structure: $fullWithdrawalRequests[starkKey_{Alice}][vaultId_{Alice}] = now$.

2. The operator `Bob` observes R_{alice} and her goal here is to clear R_{alice}. Note that the contract is designed to prevent any unauthorized removal, even for a rogue operator. Otherwise, no normal user is able to perform `fullWithdrawalRequest`, and then `freezeRequest` (after `FREEZE_GRACE_PERIOD` without fulfillment from the operator) to freeze the contract. In the context of our scenario, by the intended design, `Bob` should not be able to clear `Alice`'s legitimate, non-false, `fullWithdrawalRequest`: R_{alice}. However, based on current implementation, we show `Bob` is able to clear R_{alice}, i.e., $fullWithdrawalRequests[starkKey_{Alice}][vaultId_{Alice}] = 0$.

   To achieve that, `Bob` asks an accomplice `Malice` to submit a false `fullWithdrawalRequest`, say R_{malice}, with $vaultId_{malice}$ as its `vaultId` argument. Similarly, the contract internally records the request in $fullWithdrawalRequests[starkKey_{Malice}][vaultId_{Malice}] = now$. We emphasize R_{malice} is a false full withdrawal request, but is intentionally crafted with the same `vaultId` argument, i.e., $vaultId_{malice} = vaultId_{alice}$. Note R_{alice} and R_{malice} are two different requests regarding two different vaults! These two vaults have different `starkKey` values, but share the same `vaultId` number. (Note this is impossible in reality because of the implicit assumption.)

3. Before R_{alice}'s `FREEZE_GRACE_PERIOD` (7 days) expires, `Bob` prepares a malicious state update. For the prepared `updateState`, it recognizes R_{malice} as the false full withdrawal, but crafted in the `applicationData` argument with $requestingKey_{craft}$ occupying the corresponding modification slot that belongs to R_{malice}. As the modification slot belongs to R_{malice}, it naturally satisfies the requirement of `require(amountBefore == amountAfter)`. Moreover, as there is no validity check regarding `requestingKey`, $requestingKey_{craft}$ can be arbitrarily chosen by the operator. In this scenario, `Bob` chooses `Alice`'s `starkKey`, i.e., $requestingKey_{craft} = starkKey_{Alice}$.

4. After the preparation, `Bob` submits `updateState`. When encountering the R_{malice}'s modification slot, since $requestingKey_{craft}! = 0$, the contract further verifies two specific requirements: $require(starkKey! = requestingKey_{craft})$ (Condition I) and $require(amountBefore == amountAfter)$ (Condition II). Condition I is satisfied because $starkKey = starkKey_{malice}$, which is different from $requestingKey_{craft} = starkKey_{Alice}$. Condition II is also satisfied because this slot belongs to R_{malice} and it is a false full withdrawal request. Consequently, the contract is tricked to execute the sensitive function – `clearFullWithdrawalRequest(requestingKey, vaultId)`. Notice the arguments here: $requestingKey = requestingKey_{craft} = starkKey_{Alice}$ and $vaultId =$

$vaultId_{malice} = vaultId_{alice}$. As a result, $fullWithdrawalRequests[starkKey_{Alice}][vaultId_{Alice}] = 0$, which means the operator `Bob` successfully clears the legitimate full withdrawal request R_{alice} from `Alice`.

Recommendation Make the implicit assumption explicit. This had been addressed in the patched `UpdateState.sol` by adding comments saying that the verified publicInput implies that the vaultId is currently owned by the starkKey.

3.2 Inconsistent Uses of SafeMath

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `components/Tokens.sol`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1076 [4]

Description

Throughout the StarkEx codebase, many of the arithmetic operations follow the best practice of utilizing the `SafeMath`. However, there're some functions which do not follow the coding style but detect the overflow scenarios in their own ways, which makes the codebase slightly less consistent.

Case I Line 90-92 of `Deposits::deposit()`.

```
90          // Update the balance.
91          pendingDeposits[starkKey][tokenId][vaultId] += quantizedAmount;
92          require(pendingDeposits[starkKey][tokenId][vaultId] >= quantizedAmount,
                DEPOSIT_OVERFLOW);
```

Listing 3.3: interactions/Deposits.sol

Case II Line 109-111 of `Withdrawals::allowWithdrawal()`.

```
109         // Add accepted quantized amount.
110         withdrawal += quantizedAmount;
111         require(withdrawal >= quantizedAmount, WITHDRAWAL_OVERFLOW);
```

Listing 3.4: interactions/Withdrawals.sol

Case III Line 104-106 of `FullWithdrawals::freezeRequest()`.

```
104         // Verify timer on escape request.
105         uint256 freezeTime = requestTime + FREEZE_GRACE_PERIOD;
106         assert(freezeTime >= FREEZE_GRACE_PERIOD);
```

Listing 3.5: interactions/FullWithdrawals.sol

Recommendation Make consistent uses of `SafeMath` to detect and block various overflow scenarios.

3.3 Potential Integer Overflow in ApprovalChain

- ID: PVE-003
- Severity: Informational
- Likelihood: None
- Impact: Medium
- Target: `components/ApprovalChain.sol`
- Category: Numeric Errors [16]
- CWE subcategory: CWE-190 [7]

Description

The ApprovalChain uses a `unlockedForRemovalTime[]` array to store the removal time as well as the intention to remove an entry. However, while announcing the removal intention, the `announceRemovalIntent()` fails to check if the third parameter, `removalDelay`, makes the calculation overflow in line 58. If a caller of `announceRemovalIntent()` happens to pass in a large `removalDelay` that makes `now + removalDelay` overflow, the `removeEntry()` could not function properly. Fortunately, all the current callers throughout the StarkEx codebase invoke `announceRemovalIntent()` with a constant `removalDelay`. We suggest the `announceRemovalIntent()` itself checks the overflow instead of ensuring the correct functionality by the callers.

```
50     function announceRemovalIntent(
51         ApprovalChainData storage chain, address entry, uint256 removalDelay)
52         internal
53         onlyGovernance()
54         notFrozen()
55     {
56         safeFindEntry(chain.list, entry);
57         // solium-disable-next-line security/no-block-members
58         chain.unlockedForRemovalTime[entry] = now + removalDelay;
59     }
```

Listing 3.6: components/ApprovalChain.sol

Recommendation Ensure `now + removalDelay` would not overflow. This had been addressed in the patched `components/ApprovalChain.sol`.

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

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

Listing 3.7: components/ApprovalChain.sol

3.4 Possible Denial-of-Service in Registration

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `components/Users.sol`
- Category: Business Logic Errors[14]
- CWE subcategory: CWE-754 [11]

Description

Since users of Stark Exchange are identified within the exchange by their Stark Key, each user needs to invoke `register()` with a `starkKey` generated off-chain before any other user operation can take place. As shown in the following code snippets, when an user `register()` himself with a `starkKey`, the availability of the `starkKey` is checked in line 70 followed by the sanity checks which validate the `starkKey`. It means if Alice can somehow get $starkKey_{Bob}$ before Bob `register()` himself, she can occupy `etherKeys`[$starkKey_{Bob}$] in line 76 and make Bob's registration fail in line 70. This could be done by front-running.

```
60      function register(
61          uint256 starkKey
62      )
63          external
64      {
65          // Validate keys and availability.
66          address etherKey = msg.sender;
67          require(etherKey != ZERO_ADDRESS, INVALID_ETHER_KEY);
68          require(starkKey != 0, INVALID_STARK_KEY);
69          require(starkKeys[etherKey] == 0, ETHER_KEY_UNAVAILABLE);
70          require(etherKeys[starkKey] == ZERO_ADDRESS, STARK_KEY_UNAVAILABLE);
71          require(starkKey < K_MODULUS, INVALID_STARK_KEY);
72          require(isOnCurve(starkKey), INVALID_STARK_KEY);

74          // Update state.
75          starkKeys[etherKey] = starkKey;
76          etherKeys[starkKey] = etherKey;

78          // Log new user.
79          emit LogUserRegistered(etherKey, starkKey);
```

```
    }
```

Listing 3.8: components/Users.sol

Since an Ethereum transaction would stay in the mempool for a while before it is included in a block, Alice can always get Bob's valid starkKey before Bob's `register()` operation being included in a block and somehow `register()` in front of Bob (e.g., by assigning a higher gas price). In an extreme case, Alice can monitor the mempool and `register()` every `starkKey` she identifies, leading to denial-of-service attacks.

Recommendation Looks like there's no efficient way to solve this issue. One possible solution is raising the price of launching the attack by burning some gas in each `register()` operation. The patched `register()` has two input parameters, `starkKey` and `signature`, while the latter is used to validate the 3-tuple `(starkKey,etherKey,signature)` can't be fabricated. Besides, the permissions of the `signer` derived from the input `(starkKey,etherKey,signature)` is limited by the `userAdmins` mapping which can only be set by the `Governor`. This essentially removes the attack surface to trigger the DoS attack against the old implementation.

3.5 Misleading Comments about MVerifiers

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AvailabilityVerifiers`, `Verifiers`
- Category: Coding Practice [12]
- CWE subcategory: CWE-1116 [6]

Description

In StarkEx codebase, the convention of declaring the exported interfaces of a contract `Xyz` is implementing another contract named `MXyz`. For example, the `MApprovalChain` contract defines the interfaces such as `addEntry()`, `findEntry()`, `etc` and the `ApprovalChain` contract implements the real function logic. Based on the above convention, we identified an abnormal case — `MVerifiers`.

```
/*
  Implements MVerifiers.
*/
contract AvailabilityVerifiers is MainStorage, MApprovalChain, LibConstants {
```

Listing 3.9: AvailabilityVerifiers . sol

```
/*
  Implements MVerifiers.
*/
```

```
contract Verifiers is MainStorage, MApprovalChain, LibConstants {
```

Listing 3.10: Verifiers.sol

As shown in the above code snippets, `AvailabilityVerifiers` and `Verifiers` seem to implement `MVerifiers`. However, there's no `MVerifiers.sol` in the `interfaces` directory. Moreover, the usage of `AvailabilityVerifiers` and `Verifiers` are implemented as directly inherit those two contracts as follows.

```
contract StarkExchange is
    LibErrors,
    IVerifierActions,
    MainGovernance,
    ApprovalChain,
    AvailabilityVerifiers,
    Operator,
    Freezable,
    Tokens,
    Users,
    StateRoot,
    Deposits,
    Verifiers,
    Withdrawals,
    FullWithdrawals,
    Escapes,
    UpdateState
{
```

Listing 3.11: StarkExchange.sol

It seems `MVerifiers` is obsolete throughout the StarkEx codebase.

Recommendation Refine the comments in `AvailabilityVerifiers` and `Verifiers` contracts. This had been addressed in the patches.

3.6 Business Logic Inconsistency in Committee

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Committee`
- Category: Behavioral Issues [13]
- CWE subcategory: CWE-440 [8]

Description

The `Commitee` contract is constructed with a list of `committeeMembers` and `numSignaturesRequired` which is less than or equal to the number of `committeeMembers`. As stated in the function header comments of

`Committee::verifyAvailabilityProof()`, there should be at least `numSignaturesRequired` of valid signatures signed by `committeeMembers` to verify a specific `claimHash`. However, the `verifyAvailabilityProof()` is not implemented as how it is designed/documented.

```
    /// @dev Verifies the availability proof. Reverts if invalid.
    /// An availability proof should have a form of a concatenation of ec-signatures by
        signatories.
    /// Signatures should be sorted by signatory address ascendingly.
    /// Signatures should be 65 bytes long. r(32) + s(32) + v(1).
    /// There should be at least the number of required signatures as defined in this
        contract.
    ///
```

Listing 3.12: Committee.sol

Specifically, the sanity check in line 45-47 makes a `availabilityProofs.length` greater than `signaturesRequired * SIGNATURE_LENGTH` always fail. For example, if there are 10 committee members who construct a committee with `numSignaturesRequired=3`, the case of 5 committee members verifying a `claimHash` would fail.

```
39      function verifyAvailabilityProof(
40          bytes32 claimHash,
41          bytes calldata availabilityProofs
42      )
43          external
44      {
45          require(
46              availabilityProofs.length == signaturesRequired * SIGNATURE_LENGTH,
47              "INVALID_AVAILABILITY_PROOF_LENGTH");
```

Listing 3.13: Committee.sol

In addition, the for-loop in line 51-66 requires all signatures are signed by one of the committee members. This violates the at least the number of required signatures design. Specifically, the `require()` call in line 63 rejects all non-committee signers.

```
49          uint256 offset = 0;
50          address prevRecoveredAddress = address(0);
51          for (uint256 proofIdx = 0; proofIdx < signaturesRequired; proofIdx++) {
52              bytes32 r = bytesToBytes32(availabilityProofs, offset);
53              bytes32 s = bytesToBytes32(availabilityProofs, offset + 32);
54              uint8 v = uint8(availabilityProofs[offset + 64]);
55              offset += SIGNATURE_LENGTH;
56              address recovered = ecrecover(
57                  claimHash,
58                  v,
59                  r,
60                  s
61              );
62              // Signatures should be sorted off-chain before submitting to enable cheap
                    uniqueness check on-chain.
```

```
63              require(isMember[recovered], "AVAILABILITY_PROVER_NOT_IN_COMMITTEE");
```

Listing 3.14: Committee.sol

Recommendation Ensure how `verifyAvailabilityProof()` should be implemented. If it should verify at least the number of required signatures, the check against `availabilityProofs.length` should be modified. Also, the `require()` in line 63 should be removed. Instead, the number of valid signatures should be counted and checked in the end of the function.

```
39      function verifyAvailabilityProof(
40          bytes32 claimHash,
41          bytes calldata availabilityProofs
42      )
43          external
44      {
45          require(
46              availabilityProofs.length >= signaturesRequired * SIGNATURE_LENGTH,
47              "INVALID_AVAILABILITY_PROOF_LENGTH");

49          uint256 offset = 0;
50          uint256 validSignatures = 0;
51          address prevRecoveredAddress = address(0);
52          for (uint256 proofIdx = 0; proofIdx < (availabilityProofs.length/
                SIGNATURE_LENGTH); proofIdx++) {
53              bytes32 r = bytesToBytes32(availabilityProofs, offset);
54              bytes32 s = bytesToBytes32(availabilityProofs, offset + 32);
55              uint8 v = uint8(availabilityProofs[offset + 64]);
56              offset += SIGNATURE_LENGTH;
57              address recovered = ecrecover(
58                  claimHash,
59                  v,
60                  r,
61                  s
62              );
63              if (isMember[recovered]) {
64                  validSignatures += 1;
65              }
66              // Signatures should be sorted off-chain before submitting to enable cheap
                    uniqueness check on-chain.
67              require(recovered > prevRecoveredAddress, "NON_SORTED_SIGNATURES");
68              prevRecoveredAddress = recovered;
69          }
70          if ( validSignatures >= signaturesRequired ) {
71              verifiedFacts[claimHash] = true;
72          }
73      }
```

Listing 3.15: Committee.sol

If `verifyAvailabilityProof()` should verify exactly `signaturesRequired` signatures, the function header comments need to be fixed. This had been addressed in the patched `Committee.sol` by validating `availabilityProofs.length >= signaturesRequired * SIGNATURE_LENGTH` instead of `availabilityProofs.length == signaturesRequired * SIGNATURE_LENGTH`.

3.7 starkKey, vaultId, tokenId Ordering

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Targets: `Escapes.sol`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1099 [5]

Description

Throughout the StarkEx codebase, there are lots of use cases of the combination of `(starkKey, vaultId, tokenId)` or any two of them. In most of the cases, the 3-tuple is passed into a function as the first three parameters where `starkKey` is the first parameter followed by `vaultId` and/or `tokenId`. However, we identified one case that the ordering is not consistent to others.

```
    function escape(
        uint256 vaultId,
        uint256 starkKey,
        uint256 tokenId,
        uint256 quantizedAmount
    )
```

Listing 3.16: Escapes.sol

As shown in the above code snippets, the `escape()` function uses the ordering of `(vaultId, starkKey, tokenId)`. Since all three parameters are `uint256`, it would be better to set one of the ordering as a convention to avoid people from passing wrong ordering of parameters.

Recommendation Make the parameters ordering of `escape()` same as others. As an advanced recommendation, since the 3-tuple, `(starkKey, vaultId, tokenId)`, is used in many places, it would be good to pack them into a `struct` to simplify the code and avoid the wrong ordering in both maintenance and operations. This had been addressed in the patched `Escapes.sol` by making the parameters ordering same as others (i.e., `(starkKey, vaultId, tokenId)`).

3.8 Redundant Timestamp Checks

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Deposits.sol`, `FullWithdrawals.sol`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1041 [2]

Description

In solidity, the keyword `now` is used as an alias of `block.timestamp` which returns the current block timestamp as seconds since unix epoch. As an extreme case, the timestamp at `9999-12-31T23:59:59+00:00` would be 253402300799 or $0x3afff4417f$. It means that the day a block is packed with a timestamp which is approaching to the maximum value of `uint` (i.e., $2^{256} - 1$) is not likely to happen. Based on that, we believe the following assertion checks against timestamp overflows are redundant.

Specifically, in line 143 of `Deposits::depositReclaim()`, `requestTime` is retrieved from `cancellationRequests[starkKey][tokenId][vaultId]` which was set as `now` in `Deposits::depositCancel()`. Then, in line 145, `requestTime` is added by `DEPOSIT_CANCEL_DELAY` which is equivalent to 86400. After that, an assertion check takes place in line 146 to ensure the arithmetic operation in line 145 does not have an integer overflow. As we mentioned earlier, the `assert()` call in line 146 is a redundant assertion.

```
132     function depositReclaim(uint256 tokenId, uint256 vaultId)
133         external
134         // No modifiers: This function can always be used, even when frozen.
135     {
136         require(vaultId <= MAX_VAULT_ID, OUT_OF_RANGE_VAULT_ID);

138         // Fetch user and key.
139         address user = msg.sender;
140         uint256 starkKey = getStarkKey(user);

142         // Make sure enough time has passed.
143         uint256 requestTime = cancellationRequests[starkKey][tokenId][vaultId];
144         require(requestTime != 0, DEPOSIT_NOT_CANCELED);
145         uint256 freeTime = requestTime + DEPOSIT_CANCEL_DELAY;
146         assert(freeTime >= DEPOSIT_CANCEL_DELAY);
```

Listing 3.17: Deposits.sol

There is another case in line 94-100 of `FullWithdrawals.sol`. In particular, the `requestTime` added by `FREEZE_GRACE_PERIOD` (21 days) would not cause an integer overflow such that the `assert()` call in line 100 is redundant.

```
81     function freezeRequest(
82         uint256 vaultId
83     )
```

```
        external
        notFrozen()
    {
        // Fetch user and key.
        address user = msg.sender;
        uint256 starkKey = getStarkKey(user);

        // Verify vaultId in range.
        require(vaultId <= MAX_VAULT_ID, OUT_OF_RANGE_VAULT_ID);

        // Load request time.
        uint256 requestTime = fullWithdrawalRequests[starkKey][vaultId];
        require(requestTime != 0, FULL_WITHDRAWAL_UNREQUESTED);

        // Verify timer on escape request.
        uint256 freezeTime = requestTime + FREEZE_GRACE_PERIOD;
        assert(freezeTime >= FREEZE_GRACE_PERIOD);
```

Listing 3.18: FullWithdrawals.sol

Recommendation Remove redundant assertions with additional benefits of saving gas usage.

3.9 Upgrades Depend on States of Old Versions in Proxy

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Targets: `Proxy.sol`
- Category: Data Integrity Issues [15]
- CWE subcategory: CWE-494 [9]

Description

The `Proxy` contract delegates calls to the `implementation()` contract. Moreover, it manages the `implementation` with a two-step approach: `addImplementation()` and `upgradeTo()`. However, we identify that the `upgradeTo()` function depends on the state of the old version of implementation, which could be a deadlock if the old version has some problems which need to be fixed by a replacement.

```
    function upgradeTo(address newImplementation, bytes calldata data, bool finalize)
        external payable onlyGovernance notFinalized notFrozen {
        uint256 activation_time = enabledTime[newImplementation];

        require(activation_time > 0, "ADDRESS_NOT_UPGRADE_CANDIDATE");
        // solium-disable-next-line security/no-block-members
        require(activation_time <= now, "UPGRADE_NOT_ENABLED_YET");

        bytes32 init_vector_hash = initializationHash[newImplementation];
```

```
257         require(init_vector_hash == keccak256(abi.encode(data, finalize)), "
                CHANGED_INITIALIZER");
258         _setImplementation(newImplementation);
259         if (finalize == true) {
260             setFinalizedFlag();
261             emit FinalizedImplementation(newImplementation);
262         }

264         // solium-disable-next-line security/no-low-level-calls
265         (bool success, bytes memory returndata) = newImplementation.delegatecall(
266             abi.encodeWithSelector(this.initialize.selector, data));
267         require(success, string(returndata));

269         emit Upgraded(newImplementation);
270     }
```

Listing 3.19: Proxy.sol

Specifically, in line 249, the `upgradeTo()` can only be triggered by an effective governor (`onlyGovernance`) when the implementation is not finalized (`notFinalized` and the old implementation is not frozen `notFrozen`. Note that `notFinalized` checks a flag which is set in line 260. It means only a governor can make the call of finalizing an implementation. But, the `notFrozen` modifier is not the case.

```
    modifier notFrozen()
    {
        require(implementationIsFrozen() == false, "STATE_IS_FROZEN");
        _;
    }
```

Listing 3.20: Proxy.sol

```
    function implementationIsFrozen() private returns (bool) {
        address _implementation = implementation();

        // We can't call low level implementation before it's assigned. (i.e. ZERO).
        if (implementation() == ZERO_ADDRESS) {
            return false;
        }
        // solium-disable-next-line security/no-low-level-calls
        (bool success, bytes memory returndata) = _implementation.delegatecall(
            abi.encodeWithSignature("isFrozen()"));
        require(success, string(returndata));
        return abi.decode(returndata, (bool));
    }
```

Listing 3.21: Proxy.sol

As shown in the above code snippets, the `notFrozen` is decided by the results of `isFrozen()` of the current implementation. When `isFrozen()` is not implemented correctly, the Proxy contract cannot fix the problem by a upgrade.

Recommendation Remove the `notFrozen()` modifier on `upgradeTo()`. If we want to prevent a frozen implementation from being upgraded, a governor can update the `activation_time` by `addImplementation()`. In the patches, a `delegatecall` to the `isFrozen()` of the new implementation is invoked after the initialization call to it. This ensures the `isFrozen()` function is correctly implemented and the state is `notFrozen`, which resolves this issue.

3.10 Optimization Suggestions to Proxy

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Targets: `Proxy.sol`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1099 [5]

Description

In Proxy contract, the `_setImplementation()` is used to finalize the implementation by storing the `newImplementation` into the storage. Since a typical convention of using `_xyz()` is calling it in function `xyz()`, the existence of `_setImplementation()` looks a little bit weird here. As an example, in `ERC20.sol`, both `transfer()` and `transferFrom()` call `_transfer()` to do the real transferring tokens thing. We believe the naming of `_setImplementation()` is not compatible to others.

```
    function _setImplementation(address newImplementation) private {
        bytes32 slot = IMPLEMENTATION_SLOT;
        // solium-disable-next-line security/no-inline-assembly
        assembly {
            sstore(slot, newImplementation)
        }
    }
```

Listing 3.22: Proxy.sol

Recommendation Modify `_setImplementation()` to `setImplementation()`. This had been addressed in the patched `Proxy.sol`.

There's another suggestion related to `addImplementation()` and `removeImplementation()`. Since the `newImplementation` and related book-keeping data (i.e., `enabledTime` and `initializationHash`) which are set/clear by those two functions are only used inside `upgrade()`, those two functions are useless when the implementation is finalized. Specifically, the `notFinalized` modifier could be added to `addImplementation()` and `removeImplementation()` to reduce gas consumption.

```
    function addImplementation(address newImplementation, bytes calldata data, bool
        finalize)
        external onlyGovernance {
```

```
        require(isContract(newImplementation), "ADDRESS_NOT_CONTRACT");

        bytes32 init_hash = keccak256(abi.encode(data, finalize));
        initializationHash[newImplementation] = init_hash;

        // solium-disable-next-line security/no-block-members
        uint256 activation_time = now + UPGRADE_ACTIVATION_DELAY;

        // First implementation should not have time-lock.
        if (implementation() == ZERO_ADDRESS) {
            // solium-disable-next-line security/no-block-members
            activation_time = now;
        }

        enabledTime[newImplementation] = activation_time;
        emit ImplementationAdded(newImplementation, data, finalize);
    }
```

Listing 3.23: Proxy.sol

```
    function removeImplementation(address newImplementation)
        external onlyGovernance {

        // If we have initializer, we set the hash of it.
        uint256 activation_time = enabledTime[newImplementation];

        require(activation_time > 0, "ADDRESS_NOT_UPGRADE_CANDIDATE");

        enabledTime[newImplementation] = 0;

        initializationHash[newImplementation] = 0;
        emit ImplementationRemoved(newImplementation);
    }
```

Listing 3.24: Proxy.sol

Recommendation Add `notFinalized` to `addImplementation()` and `removeImplementation()`.

3.11 Optimization Suggestions to DexStatementVerifier

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FriStatementContract.sol`, `MerkleStatementContract.sol`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1068 [3]

Description

In `FriStatementContract::verifyFRI()`, the `friQueue` is used to store the input triplets of `(query_index, FRI_value, FRI_inverse_point)` such that the length of `friQueue` is checked in line 35 to ensure that it has `3*friQueries + 1` elements. However, the case `friQueries == 0` is meaningless in `verifyFRI()`. We could simply optimize the function by requiring `friQueue.length >= 4` and filter out the no query to process case.

```
22     function verifyFRI(
23         uint256[] memory proof,
24         uint256[] memory friQueue,
25         uint256 evaluationPoint,
26         uint256 friStepSize,
27         uint256 expectedRoot) public {
28
29         require(friStepSize <= FRI_MAX_FRI_STEP, "FRI step size too large");
30         /*
31           The friQueue should have of 3*nQueries + 1 elements, beginning with nQueries
               triplets
32           of the form (query_index, FRI_value, FRI_inverse_point), and ending with a
               single buffer
33           cell set to 0, which is accessed and read during the computation of the FRI
               layer.
34         */
35         require(
36             friQueue.length % 3 == 1,
37             "FRI Queue must be composed of triplets plus one delimiter cell");
```

Listing 3.25: FriStatementContract.sol

Recommendation Check the length of `friQueue` and ensure that there's at least one triplet to process. This had been addressed in the patched `FriStatementContract.sol`.

```
22     function verifyFRI(
23         uint256[] memory proof,
24         uint256[] memory friQueue,
25         uint256 evaluationPoint,
26         uint256 friStepSize,
27         uint256 expectedRoot) public {
28
29         require(friStepSize <= FRI_MAX_FRI_STEP, "FRI step size too large");
30         /*
31           The friQueue should have of 3*nQueries + 1 elements, beginning with nQueries
               triplets
32           of the form (query_index, FRI_value, FRI_inverse_point), and ending with a
               single buffer
33           cell set to 0, which is accessed and read during the computation of the FRI
               layer.
34         */
35         require(friQueue.length >= 4, "No query to process");
36         require(
```

```
            friQueue.length % 3 == 1,
            "FRI Queue must be composed of triplets plus one delimiter cell");
```

Listing 3.26: FriStatementContract.sol

Besides, there's a typo in line 34 of `verifyMerkle()` where the word `function` is misspelled as `functin`.

```
13     function verifyMerkle(
14         uint256[] memory merkleView,
15         uint256[] memory initialMerkleQueue,
16         uint256 height,
17         uint256 expectedRoot
18         )
19         public
20     {
21         require(height < 200, "Height must be < 200.");
22
23         uint256 merkleQueuePtr;
24         uint256 channelPtr;
25         uint256 nQueries;
26         uint256 dataToHashPtr;
27         uint256 badInput = 0;
28
29         assembly {
30             // Skip 0x20 bytes length at the beginning of the merkleView.
31             let merkleViewPtr := add(merkleView, 0x20)
32             // Let channelPtr point to a free space.
33             channelPtr := mload(0x40) // freePtr.
34             // channelPtr will point to the merkleViewPtr because the functin 'verify'
                   expects
```

Listing 3.27: MerkleStatementContract.sol

Recommendation Modify `functin` to `function`. This had been addressed in the patched `MerkleStatementContract.sol`.

3.12 Possible Integer Overflow in MerkleVerifier

- ID: PVE-012
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MerkleVerifier.sol`
- Category: Numeric Errors [16]
- CWE subcategory: CWE-190 [7]

Description

In `MerkleVerifier::verify()`, `n` slots of leaf indices and leaf values are iterated through `queuePtr` for hash calculation. However, there's a possible integer overflow throughout this process such that a malicious batch of queries could has the same verification result as a legit batch of queries.

```
22     function verify(
23         uint256 channelPtr,
24         uint256 queuePtr,
25         bytes32 root,
26         uint256 n)
27         internal view
28         returns (bytes32 hash)
29     {
30         uint256 lhashMask = getHashMask();
31
32         assembly {
33             // queuePtr + i * 0x40 gives the i'th index in the queue.
34             // hashesPtr + i * 0x40 gives the i'th hash in the queue.
35             let hashesPtr := add(queuePtr, 0x20)
36             let queueSize := mul(n, 0x40)
37             let slotSize := 0x40
```

Listing 3.28: MerkleVerifier.sol

Specifically, in line 35, `queueSize` is set as `n*0x40`. When `n` is larger or equal to `0x0400...0000`, `n*0x40` would overflow. Say if a triplet of `(queuePtr, root, 1)` is verified. An attacker could use that fact to verify `(queuePtr, root, 0x0400...0001)` with bunch of malicious queries appended right after the first legit query due to the fact that `0x0400...0001*0x40 = 0x40`. Only the first legit query would be hashed. Fortunately, `verify()` is an internal function which is invoked by `verifyMerkle()` so that the bad actors cannot exploit this vulnerability directly.

```
    function verifyMerkle(
        uint256[] memory merkleView,
        uint256[] memory initialMerkleQueue,
        uint256 height,
        uint256 expectedRoot
        )
        public
    {
        require(height < 200, "Height must be < 200.");

        uint256 merkleQueuePtr;
        uint256 channelPtr;
        uint256 nQueries;
        uint256 dataToHashPtr;
        uint256 badInput = 0;

        assembly {
            // Skip 0x20 bytes length at the beginning of the merkleView.
            let merkleViewPtr := add(merkleView, 0x20)
```

```
32            // Let channelPtr point to a free space.
33            channelPtr := mload(0x40) // freePtr.
34            // channelPtr will point to the merkleViewPtr because the function 'verify'
                 expects
35            // a pointer to the proofPtr.
36            mstore(channelPtr, merkleViewPtr)
37            // Skip 0x20 bytes length at the beginning of the initialMerkleQueue.
38            merkleQueuePtr := add(initialMerkleQueue, 0x20)
39            // Get number of queries.
40            nQueries := div(mload(initialMerkleQueue), 0x2)
```

Listing 3.29: MerkleStatementContract.sol

However, as we look into the public function `verifyMerkle()`, the `nQueries` is derived from the user controllable data `initialMerkleQueue` in line 40. The attacker could craft the `initialMerkleQueue` to make `nQueries >= 0x0400...0000`, which leads to the overflow mentioned above. To sum up, an attacker can trick the verifier by appending arbitrary nodes into an already verified merkle tree represented by `initialMerkleQueue`.

Recommendation Validate the number of queries. In the patches, this issue had be resolved by validating the number of queries with `MAX_N_MERKLE_VERIFIER_QUERIES` as shown in the following code snippets:

```
    function verify(
        uint256 channelPtr,
        uint256 queuePtr,
        bytes32 root,
        uint256 n)
        internal view
        returns (bytes32 hash)
    {
        uint256 lhashMask = getHashMask();
        require(n <= MAX_N_MERKLE_VERIFIER_QUERIES, "TOO_MANY_MERKLE_QUERIES");
```

Listing 3.30: MerkleVerifier.sol

```
    function verifyMerkle(
        uint256[] memory merkleView,
        uint256[] memory initialMerkleQueue,
        uint256 height,
        uint256 expectedRoot
        )
        public
    {
        require(height < 200, "Height must be < 200.");
        require(
            initialMerkleQueue.length <= MAX_N_MERKLE_VERIFIER_QUERIES * 2,
            "TOO_MANY_MERKLE_QUERIES");
```

Listing 3.31: MerkleStatementContract.sol

Some other places throughout the `evm-verifier` codebase have the potential integer overflow issues. All possible overflow cases had been guarded by sanity checks in the patches. We list them in the following:

`n` in line 23.

```
15  function verify(uint256 /*channelPtr*/, uint256 queuePtr, bytes32 root, uint256 n)
        internal view
16      returns(bytes32) {
17      bytes32 statement;
18
19      assembly {
20          let dataToHashPtrStart := mload(0x40) // freePtr.
21          let dataToHashPtrCur := dataToHashPtrStart
22
23          let queEndPtr := add(queuePtr, mul(n, 0x40))
```

Listing 3.32: MerkleStatementVerifier.sol

`nCoefs` in line 22.

```
14  function hornerEval(uint256 coefsStart, uint256 point, uint256 nCoefs)
15      internal pure
16      returns (uint256) {
17      uint256 result = 0;
18      uint256 prime = PrimeFieldElement0.K_MODULUS;
19
20      require(nCoefs % 8 == 0, "Number of polynomial coefficients must be divisible by
            8");
21      assembly {
22          let coefsPtr := add(coefsStart, mul(nCoefs, 0x20))
```

Listing 3.33: HornerEvaluator.sol

`nElements` in line 51.

```
41  function sendFieldElements(uint256 channelPtr, uint256 nElements, uint256 targetPtr)
42      internal pure
43  {
44      assembly {
45          let PRIME := 0
            x800000000000011000000000000000000000000000000000000000000000001
46          let PRIME_MON_R_INV := 0
            x40000000000001100000000000012100000000000000000000000000000000
47          let PRIME_MASK := 0
            x0fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
48          let digestPtr := add(channelPtr, 0x20)
49          let counterPtr := add(channelPtr, 0x40)
50
51          let endPtr := add(targetPtr, mul(nElements, 0x20))
```

Listing 3.34: VerifierChannel.sol

`nColumns` in line 268.

```
    function readQuriesResponsesAndDecommit(
        uint256[] memory ctx, uint256 nColumns, uint256 proofDataPtr, bytes32 merkleRoot
        )
         internal view {
        uint256 nUniqueQueries = ctx[MM_N_UNIQUE_QUERIES];
        uint256 channelPtr = getPtr(ctx, MM_CHANNEL);
        uint256 friQueue = getPtr(ctx, MM_FRI_QUEUE);
        uint256 friQueueEnd = friQueue + nUniqueQueries * 0x60;
        uint256 merkleQueuePtr = getPtr(ctx, MM_MERKLE_QUEUE);
        uint256 rowSize = 0x20 * nColumns;
        uint256 lhashMask = getHashMask();
```

Listing 3.35: StarkVerifier.sol

`offset` in line 13.

```
6     function getPtr(uint256[] memory ctx, uint256 offset)
7         internal pure
8         returns (uint256) {
9         uint256 ctxPtr;
10        assembly {
11            ctxPtr := add(ctx, 0x20)
12        }
13        return ctxPtr + offset * 0x20;
14    }
```

Listing 3.36: MemoryAccessUtils.sol

`nModifications` in line 214.

```
193   function computeBoundaryPeriodicColumn(
194       uint256 modificationsPtr, uint256 nModifications, uint256 nTransactions, uint256
            point,
195       uint256 prime, uint256 gen, uint256 resultArrayPtr)
196       internal view {
197       bool sorted = true;
198       assembly {
199           function expmod(base, exponent, modulus) -> res {
200               let p := mload(0x40)
201               mstore(p, 0x20)                  // Length of Base.
202               mstore(add(p, 0x20), 0x20)       // Length of Exponent.
203               mstore(add(p, 0x40), 0x20)       // Length of Modulus.
204               mstore(add(p, 0x60), base)       // Base.
205               mstore(add(p, 0x80), exponent)   // Exponent.
206               mstore(add(p, 0xa0), modulus)    // Modulus.
207               // Call modexp precompile.
208               if iszero(staticcall(not(0), 0x05, p, 0xc0, p, 0x20)) {
209                   revert(0, 0)
210               }
211               res := mload(p)
212           }
213
214           let lastOffset := mul(nModifications, 0x20)
```

Listing 3.37: DexVerifier.sol

3.13 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.5.2;` instead of `pragma solidity ^0.5.2;`.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries. In case there is an absolute need of leveraging experimental features or integrating external libraries, make necessary contingency plans.

4 Conclusion

In this audit, we thoroughly analyzed the StarkEx documentation and implementation. The audited system does involve various intricacies in both design and implementation. 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 [20, 21, 22, 23, 25].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [26] 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] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[3] MITRE. CWE-1068: Inconsistency Between Implementation and Documented Design. https://cwe.mitre.org/data/definitions/1068.html.

[4] MITRE. CWE-1076: Insufficient Adherence to Expected Conventions. https://cwe.mitre.org/data/definitions/1076.html.

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

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

[7] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[8] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[9] MITRE. CWE-494: Download of Code Without Integrity Check. https://cwe.mitre.org/data/definitions/494.html.

[10] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[11] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

[13] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[14] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[15] MITRE. CWE CATEGORY: Data Integrity Issues. https://cwe.mitre.org/data/definitions/1214.html.

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

[17] MITRE. CWE CATEGORY: Often Misused: Arguments and Parameters. https://cwe.mitre.org/data/definitions/559.html.

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

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

[20] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

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

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

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

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

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

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