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 our partnership with dYdX, a StarkEx scalability engine for perpetual trading, 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				
High	**Critical**	**High**	**Medium**	
Medium	**High**	**Medium**	**Low**	
Low	**Medium**	**Low**	**Low**	**Informational**
	High	Medium	Low	
		Likelihood		

Summary

Severity	# of Findings
Critical	**0**
High	**0**
Medium	**1**
Low	**3**
Informational	**3**
Total	**7**

Key Findings

ID	Severity	Title	Status
PVE-001	Low	Suggested Solidity Compiler Version Upgrade	FIXED
PVE-002	Low	Improved assetType Enforcement	FIXED
PVE-003	Medium	Lack Of Replay Protection Against Legitimate Freezes	FIXED
PVE-004	Low	Improved Sanity Checks in Users::registerUser()	FIXED
PVE-005	Info.	Improved Sanity Checks in MainDispatcherBase::initialize()	FIXED
PVE-006	Info.	Redundant Check Removals in UpdateState::rootUpdate()	FIXED
PVE-007	Info.	Explicit Block of Logic Contract Initialization	FIXED

SMART CONTRACT AUDIT REPORT

for

STARKWARE INDUSTRIES LTD.

Prepared By: Shuxiao Wang

PeckShield

February 21, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	February 21, 2021	Xuxian Jiang	Final Release
1.0-rc2	February 4, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	February 3, 2021	Xuxian Jiang	Release Candidate #1
0.3	January 19, 2021	Xuxian Jiang	Additional Findings #2
0.2	January 12, 2021	Xuxian Jiang	Additional Findings #1
0.1	January 5, 2021	Xuxian Jiang	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 design document and related source code of the **StarkPerpetual** 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 StarkPerpetual

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

The basic information of StarkPerpetual is as follows:

Table 1.1: Basic Information of StarkPerpetual

Item	Description
Issuer	StarkWare Industries Ltd.
Website	https://starkware.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 21, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this repository contains a number of sub-directories (e.g., `scalable-dex`, `evm-verifier`, and `common-contracts`) and this audit focuses on the `perpetual` sub-directory and related surrounding contracts. However, the cryptographic proofs as well as the associated `assembly` implementations are not part of this audit. Also, both versions of `StarkEx` use a proprietary tool to generate the dispatch-related mapping of supported functions (in so-called `subcontracts`), and the mapping generation falls out of the audit scope.

- https://github.com/starkware-libs/starkex2.0-contracts.git (2a60039)

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

- https://github.com/starkware-libs/starkex2.0-contracts.git (b2cc6f9)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full List of Check Items

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

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the StarkPerpetual 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	3	■ ■ ■
Informational	3	■ ■ ■
Total	7	

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, 3 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Audit Findings of StarkPerpetual Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Solidity Compiler Version Upgrade	Coding Practices	Fixed
PVE-002	Low	Improved assetType Enforcement	Business Logic	Fixed
PVE-003	Medium	Lack Of Replay Protection Against Legitimate Freezes	Business Logic	Fixed
PVE-004	Low	Improved Sanity Checks in Users::registerUser()	Business Logic	Fixed
PVE-005	Informational	Improved Sanity Checks in MainDispatcherBase::initialize()	Business Logic	Fixed
PVE-006	Informational	Redundant Check Removal in UpdateState::rootUpdate()	Coding Practices	Fixed
PVE-007	Informational	Explicit Block Of Logic Contract Initialization	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Suggested Solidity Compiler Version Upgrade

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1116 [1]

Description

In StarkPerpetual, the current implementation chooses the following version pragma: `pragma solidity ^0.5.2`. The intention is to allow for the compilation with the `Solidity` compiler from version 0.5.2 (inclusive) up to 0.6.0 (exclusive). As a rule of thumb, we strongly suggest the deployment of contracts with the same compiler version and flags that they have been tested the most with. With that, locking the pragma is helpful in ensuring that contracts do not accidentally get deployed using untested compiler versions.

In the meantime, it is important to keep in mind that smart contracts as a whole are still in an early, but exciting stage of development. Often times, we observe breaking upgrades in the `Solidity` compiler or the discovery of unintended bugs in recent compilers.[1] 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`.

However, it comes to our attention that the default compiler version 0.5.2 has a number of bugs, i.e., `privateCanBeOverridden`, `SignedArrayStorageCopy`, `ABIEncoderV2StorageArrayWithMultiSlotElement`, `DynamicConstructorArgumentsClippedABIV2`, `UninitializedFunctionPointerInConstructor`, `IncorrectEventSignatureInLibraries`, and `ABIEncoderV2PackedStorage`. For each specific bug, we outline below a brief summary. A full list of current known bugs of `Solidity` compiler can be found at

[1]Note that a new released compiler may have higher risks of undiscovered bugs.

the following link: https://github.com/ethereum/solidity/blob/develop/docs/bugs.json.

```
    {
        "name": "privateCanBeOverridden",
        "summary": "Private methods can be overridden by inheriting contracts.",
        "description": "While private methods of base contracts are not visible and
            cannot be called directly from the derived contract, it is still possible to
             declare a function of the same name and type and thus change the behaviour
            of the base contract's function.",
        "introduced": "0.3.0",
        "fixed": "0.5.17",
        "severity": "low"
    },

    {
        "name": "SignedArrayStorageCopy",
        "summary": "Assigning an array of signed integers to a storage array of
            different type can lead to data corruption in that array.",
        "description": "In two's complement, negative integers have their higher order
            bits set. In order to fit into a shared storage slot, these have to be set
            to zero. When a conversion is done at the same time, the bits to set to zero
             were incorrectly determined from the source and not the target type. This
            means that such copy operations can lead to incorrect values being stored.",
        "link": "https://blog.ethereum.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.7",
        "fixed": "0.5.10",
        "severity": "low/medium"
    },

    {
        "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
        "summary": "Storage arrays containing structs or other statically-sized arrays
            are not read properly when directly encoded in external function calls or in
             abi.encode*.",
        "description": "When storage arrays whose elements occupy more than a single
            storage slot are directly encoded in external function calls or using abi.
            encode*, their elements are read in an overlapping manner, i.e. the element
            pointer is not properly advanced between reads. This is not a problem when
            the storage data is first copied to a memory variable or if the storage
            array only contains value types or dynamically-sized arrays.",
        "link": "https://blog.ethereum.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.16",
        "fixed": "0.5.10",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },

    {
        "name": "DynamicConstructorArgumentsClippedABIV2",
        "summary": "A contract's constructor that takes structs or arrays that contain
            dynamically-sized arrays reverts or decodes to invalid data.",
```

```
        "description": "During construction of a contract, constructor parameters are copied from the code section to memory for decoding. The amount of bytes to copy was calculated incorrectly in case all parameters are statically-sized but contain dynamically-sized arrays as struct members or inner arrays. Such types are only available if ABIEncoderV2 is activated.",
        "introduced": "0.4.16",
        "fixed": "0.5.9",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },

    {
        "name": "UninitializedFunctionPointerInConstructor",
        "summary": "Calling uninitialized internal function pointers created in the constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special piece of code that causes a revert when called. Jump target positions are different during construction and after deployment, but the code for setting this special jump target only considered the situation after deployment.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },

    {
        "name": "IncorrectEventSignatureInLibraries",
        "summary": "Contract types used in events in libraries cause an incorrect event signature hash",
        "description": "Instead of using the type `address` in the hashed signature, the actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },

    {
        "name": "ABIEncoderV2PackedStorage",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can cause data corruption if encoded directly from storage using the experimental ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes are not properly decoded from storage when encoded directly (i.e. not via a memory type) using ABIEncoderV2. This can cause corruption in the values themselves but can also overwrite other parts of the encoded data.",
        "link": "https://blog.ethereum.org/2019/03/26/solidity-optimizer-and-abiencoderv2-bug/",
        "introduced": "0.5.0",
        "fixed": "0.5.7",
        "severity": "low",
        "conditions": {
```

```
            "ABIEncoderV2": true
        }
    }
```

Listing 3.1: A List Of Known Bugs in **Solidity** Compiler Version 0.5.2

Recommendation Upgrade the `solidity` compiler version to `0.6.12`.

Status This issue has been addressed by upgrading the compiler version to `^0.6.11`.

3.2 Improved assetType Enforcement

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The StarkPerpetual protocol defines its standard in encapsulating and processing supported assets with the notion of `assetType` and `quantization`. In particular, there are in total five self-explanatory `assetType`: `ETH_SELECTOR`, `ERC20_SELECTOR`, `ERC721_SELECTOR`, `MINTABLE_ERC20_SELECTOR`, and `MINTABLE_ERC721_SELECTOR`.

To elaborate, we show below the `deposit()` routine that handles the user deposits of assets with two `assetType`: `ETH_SELECTOR` and `ERC20_SELECTOR`. However, per line 154, the routine is programmed to only validate they are not `isMintableAssetType()`, i.e., `MINTABLE_ERC20_SELECTOR`, and `MINTABLE_ERC721_SELECTOR`. In other words, the validation is insufficient in permitting `ERC721_SELECTOR` assets.

```
143     function deposit(
144         uint256 starkKey,
145         uint256 assetType,
146         uint256 vaultId,
147         uint256 quantizedAmount
148     ) public notFrozen()
149     {
150         // No need to verify amount > 0, a deposit with amount = 0 can be used to undo
                cancellation.
151         require(vaultId <= STARKEX_MAX_VAULT_ID, "OUT_OF_RANGE_VAULT_ID");
152         // starkKey must be registered.
153         require(ethKeys[starkKey] != ZERO_ADDRESS, "INVALID_STARK_KEY");
154         require(!isMintableAssetType(assetType), "MINTABLE_ASSET_TYPE");
155         uint256 assetId = assetType;
```

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

        // Disable the cancellationRequest timeout when users deposit into their own
            account.
        if (isMsgSenderStarkKeyOwner(starkKey) &&
                cancellationRequests[starkKey][assetId][vaultId] != 0) {
            delete cancellationRequests[starkKey][assetId][vaultId];
        }

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

        // Log event.
        emit LogDeposit(
            msg.sender,
            starkKey,
            vaultId,
            assetType,
            fromQuantized(assetType, quantizedAmount),
            quantizedAmount
        );
    }
```

Listing 3.2: Deposits :: deposit()

For those assets of the `ERC721_SELECTOR` type, the protocol defines its own handler – `depositNft()`. However, it is also hardcoded to validate `require(!isMintableAssetType(assetType), "MINTABLE_ASSET_TYPE")` (line 116).

```
106    function depositNft(
107        uint256 starkKey,
108        uint256 assetType,
109        uint256 vaultId,
110        uint256 tokenId
111    ) external notFrozen()
112    {
113        require(vaultId <= STARKEX_MAX_VAULT_ID, "OUT_OF_RANGE_VAULT_ID");
114        // starkKey must be registered.
115        require(ethKeys[starkKey] != ZERO_ADDRESS, "INVALID_STARK_KEY");
116        require(!isMintableAssetType(assetType), "MINTABLE_ASSET_TYPE");
117        uint256 assetId = calculateNftAssetId(assetType, tokenId);

119        // Update the balance.
120        pendingDeposits[starkKey][assetId][vaultId] = 1;

122        // Disable the cancellationRequest timeout when users deposit into their own
               account.
```

```
        if (isMsgSenderStarkKeyOwner(starkKey) &&
                cancellationRequests[starkKey][assetId][vaultId] != 0) {
            delete cancellationRequests[starkKey][assetId][vaultId];
        }

        // Transfer the tokens to the Deposit contract.
        transferInNft(assetType, tokenId);

        // Log event.
        emit LogNftDeposit(msg.sender, starkKey, vaultId, assetType, tokenId, assetId);
    }
```

Listing 3.3: Deposits::depositNft()

Note two other routines `withdraw()` and `withdrawTo()` can be further enforced to accepts tokens of `ETH_SELECTOR` or `ERC20_SELECTOR`. And others `withdrawNft()` and `withdrawNftTo()` only take assets with the `assetType` of `ERC721_SELECTOR`.

Recommendation Validate the given `assetType` so that only the intended types are accepted.

Status The issue has been fixed by the following commit: `6d7d399`.

3.3 Lack Of Replay Protection Against Legitimate Freezes

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The StarkPerpetual protocol has a unique feature in allowing any user to opt to perform a forced action on a given off-chain vault. Failure of the off-chain exchange to service a forced action request within a given time frame gives the user the option to freeze the exchange, hence disabling the ability to update its state.

In the following, we use the `forcedTradeRequest()` function as an example. For elaboration, we show its code snippet below. It implements a rather straightforward logic in firstly validating the given input arguments, next recording the forced trade request (via `setForcedTradeRequest()` - line 55), and finally validating the trading peer's signature (via `validatePartyBSignature()` - line 68).

```
65      // NOLINTNEXTLINE: uninitialized-state.
66      function forcedTradeRequest(
67          uint256 starkKeyA,
68          uint256 starkKeyB,
```

```
        uint256 vaultIdA,
        uint256 vaultIdB,
        uint256 collateralAssetId,
        uint256 syntheticAssetId,
        uint256 amountCollateral,
        uint256 amountSynthetic,
        bool aIsBuyingSynthetic,
        uint256 nonce,
        bytes calldata signature
    ) external notFrozen() isSenderStarkKey(starkKeyA) {
        require(vaultIdA < PERPETUAL_POSITION_ID_UPPER_BOUND, "OUT_OF_RANGE_POSITION_ID"
            );
        require(vaultIdB < PERPETUAL_POSITION_ID_UPPER_BOUND, "OUT_OF_RANGE_POSITION_ID"
            );

        require(collateralAssetId == systemAssetType, "SYSTEM_ASSET_NOT_IN_TRADE");
        require(collateralAssetId != uint256(0x0), "SYSTEM_ASSET_NOT_SET");
        require(collateralAssetId != syntheticAssetId, "IDENTICAL_ASSETS");
        require(configurationHash[syntheticAssetId] != bytes32(0x0), "UNKNOWN_ASSET");
        require(amountCollateral < PERPETUAL_AMOUNT_UPPER_BOUND, "ILLEGAL_AMOUNT");
        require(amountSynthetic < PERPETUAL_AMOUNT_UPPER_BOUND, "ILLEGAL_AMOUNT");
        require(nonce < K_MODULUS, "INVALID_NONCE_VALUE");

        // Start timer on escape request.
        setForcedTradeRequest(
            starkKeyA,
            starkKeyB,
            vaultIdA,
            vaultIdB,
            collateralAssetId,
            syntheticAssetId,
            amountCollateral,
            amountSynthetic,
            aIsBuyingSynthetic,
            nonce
        );

        validatePartyBSignature(
            starkKeyA,
            starkKeyB,
            vaultIdA,
            vaultIdB,
            collateralAssetId,
            syntheticAssetId,
            amountCollateral,
            amountSynthetic,
            aIsBuyingSynthetic,
            nonce,
            signature
        );

        // Log request.
```

```
        emit LogForcedTradeRequest(
            starkKeyA,
            starkKeyB,
            vaultIdA,
            vaultIdB,
            collateralAssetId,
            syntheticAssetId,
            amountCollateral,
            amountSynthetic,
            aIsBuyingSynthetic,
            nonce
        );

        // Burn gas to prevent denial of service (too many requests per block).
        for (uint256 i = 0; i < 22231; i++) {}
        // solium-disable-previous-line no-empty-blocks
    }
```

Listing 3.4: ForcedTrades::forcedTradeRequest()

After the request timeout (`FREEZE_GRACE_PERIOD`), if the forced trade is still not fulfilled, any one may call `freezeRequest()` to freeze the exchange operations.

```
    function freezeRequest(
        uint256 starkKeyA,
        uint256 starkKeyB,
        uint256 vaultIdA,
        uint256 vaultIdB,
        uint256 collateralAssetId,
        uint256 syntheticAssetId,
        uint256 amountCollateral,
        uint256 amountSynthetic,
        bool aIsBuyingSynthetic,
        uint256 nonce
    ) external notFrozen() {
        // Verify vaultId in range.
        require(vaultIdA < PERPETUAL_POSITION_ID_UPPER_BOUND, "OUT_OF_RANGE_POSITION_ID"
            );
        require(vaultIdB < PERPETUAL_POSITION_ID_UPPER_BOUND, "OUT_OF_RANGE_POSITION_ID"
            );

        // Load request time.
        uint256 requestTime = getForcedTradeRequest(
            starkKeyA,
            starkKeyB,
            vaultIdA,
            vaultIdB,
            collateralAssetId,
            syntheticAssetId,
            amountCollateral,
            amountSynthetic,
            aIsBuyingSynthetic,
            nonce
```

```
        );

        require(requestTime != 0, "FORCED_TRADE_UNREQUESTED");

        // Verify timer on escape request.
        uint256 freezeTime = requestTime + FREEZE_GRACE_PERIOD;
        assert(freezeTime >= FREEZE_GRACE_PERIOD);
        // solium-disable-next-line security/no-block-members
        require(block.timestamp >= freezeTime, "FORCED_TRADE_PENDING"); // NOLINT:
            timestamp.

        freeze();
    }
```

Listing 3.5: ForcedTrades::freezeRequest()

When examining the above logic, it comes to our attention that a valid freezed request can always be replayed. In other words, if an exchange has been freezed before, any one can freeze it again.

Note the handling logic of `TradeRequest` is similar to another forced request `ForcedWithdrawalRequest` . And the related handling logic shares the same issue.

Recommendation Enhance the freeze logic to validate current freeze request in order to properly prevent them from being replayed again.

Status The issue has been fixed by the following commit: `6d7d399`.

3.4 Improved Sanity Checks in Users::registerUser()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Users`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In StarkPerpetual, users are identified within the exchange by their `Stark Keys` which are public keys defined over a `Stark-friendly elliptic curve` that is different from the standard `Ethereum elliptic curve`. The use of `Stark Keys` makes it necessary to associate exchange users with their `Ethereum` account addresses. A user registration process is implemented to achieve that.

In the following, we show the `registerUser()` routine that is designed to accomplish the above user registration. While examining its logic, we notice that the signature validation (lines 88 – 89) uses the built-in precompile, i.e., `ecrecover()`. A failed verification from `ecrecover()` will return 0, which may be interpreted as `address(0)`.

```
    function registerUser(address ethKey, uint256 starkKey, bytes calldata signature)
        external {
        // Validate keys and availability.
        require(starkKey != 0, "INVALID_STARK_KEY");
        require(starkKey < K_MODULUS, "INVALID_STARK_KEY");
        require(ethKey != ZERO_ADDRESS, "INVALID_ETH_ADDRESS");
        require(ethKeys[starkKey] == ZERO_ADDRESS, "STARK_KEY_UNAVAILABLE");
        require(isOnCurve(starkKey), "INVALID_STARK_KEY");
        require(signature.length == 65, "INVALID_SIGNATURE");

        bytes32 signedData = keccak256(abi.encodePacked("UserRegistration:", ethKey,
            starkKey));

        bytes memory sig = signature;
        uint8 v = uint8(sig[64]);
        bytes32 r;
        bytes32 s;

        // solium-disable-next-line security/no-inline-assembly
        assembly {
            r := mload(add(sig, 32))
            s := mload(add(sig, 64))
        }

        address signer = ecrecover(signedData, v, r, s);
        require(isUserAdmin(signer), "INVALID_SIGNATURE");

        // Update state.
        ethKeys[starkKey] = ethKey;

        // Log new user.
        emit LogUserRegistered(ethKey, starkKey, msg.sender);
    }
```

Listing 3.6: Users::registerUser()

With that, if `address(0)` is registered by mistake as the user admin that has the privileged to register new users, any one can abuse the current logic in `registerUser()` to claim the ownership of a new, previously unclaimed `Start Key`.

```
    function registerUser(address ethKey, uint256 starkKey, bytes calldata signature)
        external {
        // Validate keys and availability.
        require(starkKey != 0, "INVALID_STARK_KEY");
        require(starkKey < K_MODULUS, "INVALID_STARK_KEY");
        require(ethKey != ZERO_ADDRESS, "INVALID_ETH_ADDRESS");
        require(ethKeys[starkKey] == ZERO_ADDRESS, "STARK_KEY_UNAVAILABLE");
        require(isOnCurve(starkKey), "INVALID_STARK_KEY");
        require(signature.length == 65, "INVALID_SIGNATURE");

        bytes32 signedData = keccak256(abi.encodePacked("UserRegistration:", ethKey,
            starkKey));
```

```
        bytes memory sig = signature;
        uint8 v = uint8(sig[64]);
        bytes32 r;
        bytes32 s;

        // solium-disable-next-line security/no-inline-assembly
        assembly {
            r := mload(add(sig, 32))
            s := mload(add(sig, 64))
        }

        address signer = ecrecover(signedData, v, r, s);
        require(signer != address(0) && isUserAdmin(signer), "INVALID_SIGNATURE");

        // Update state.
        ethKeys[starkKey] = ethKey;

        // Log new user.
        emit LogUserRegistered(ethKey, starkKey, msg.sender);
    }
```

Listing 3.7: Revised Users::registerUser()

Recommendation Explicitly validate that the `signer` after `ecrecover()` is not `address(0)`. An example revision to the above routine is shown below:

Status The issue has been fixed by the following commit: `6d7d399`.

3.5 Improved Sanity Checks in MainDispatcherBase::initialize()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MainDispatcherBase`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The StarkPerpetual implementation takes a proxy-based approach where the proxy contract is deployed at the front-end while the logic contract contains the actual business logic implementation. This approach has the flexible support in terms of upgradeability. However, the upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts.

Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

To elaborate, we show below the `initialize()` routine in the logic contract implementation, i.e., `MainDispatcherBase`. This routine is designed to take the following three key steps: It firstly extracts and assigns the given subcontracts addresses, then determines the so-called `external initializing contract` or `EIC` address, and next calls the `EIC`, if present. If `EIC` is not present, it loops through the subcontracts and for each one it extracts the initializing data and passes it to the subcontract's initialize function.

```
    // NOLINTNEXTLINE: external-function.
    function initialize(bytes memory data) public {
        // Number of sub-contracts.
        uint256 nSubContracts = getNumSubcontracts();

        // We support currently 4 bits per contract, i.e. 16, reserving 00 leads to 15.
        require(nSubContracts <= 15, "TOO_MANY_SUB_CONTRACTS");

        // Init data MUST include addresses for all sub-contracts.
        require(data.length >= 32 * nSubContracts, "SUB_CONTRACTS_NOT_PROVIDED");

        // Ensure implementation is a valid contract.
        require(implementation().isContract(), "INVALID_IMPLEMENTATION");

        // Size of passed data, excluding sub-contract addresses.
        uint256 additionalDataSize = data.length - 32 * (nSubContracts + 1);

        // Sum of subcontract initializers. Aggregated for verification near the end.
        uint256 totalInitSizes = 0;

        // Offset (within data) of sub-contract initializer vector.
        // Just past the sub-contract addresses.
        uint256 initDataContractsOffset = 32 * (nSubContracts + 1);

        // Extract & update contract addresses.
        for (uint256 nContract = 1; nContract <= nSubContracts; nContract++) {
            address contractAddress;

            // Extract sub-contract address.
            // solium-disable-next-line security/no-inline-assembly
            assembly {
                contractAddress := mload(add(data, mul(32, nContract)))
            }

            validateSubContractIndex(nContract, contractAddress);

            // Contracts are indexed from 1 and 0 is not in use here.
            setSubContractAddress(nContract, contractAddress);
        }
```

```
142         // Check if we have an external initializer contract.
143         address externalInitializerAddr;

145         // 2. Extract sub-contract address, again. It's cheaper than reading from
                storage.
146         // solium-disable-next-line security/no-inline-assembly
147         assembly {
148             externalInitializerAddr := mload(add(data, mul(32, add(nSubContracts, 1))))
149         }

151         // 3(a). Yield to EIC initialization.
152         if (externalInitializerAddr != address(0x0)) {
153             callExternalInitializer(data, externalInitializerAddr, additionalDataSize);
154             return;
155         }

157         // 3(b). Subcontracts initialization.
158         // I. If no init data passed besides sub-contracts, return.
159         if (additionalDataSize == 0) {
160             return;
161         }

163         // Just to be on the safe side.
164         assert(externalInitializerAddr == address(0x0));

166         // II. Gate further initialization.
167         initializationSentinel();
168         ...
169 }
```

Listing 3.8: MainDispatcherBase:: initialize ()

Before kicking off the first step, the `initialize` routine validates the given input data. There is a specific requirement: i.e., `require(data.length >= 32 * nSubContracts, "SUB_CONTRACTS_NOT_PROVIDED")` (line 111). This specific requirement does not take into account the 4 bytes occupied by the `external initializer contract`. Therefore, the above requirement can be revised as follows: `require(data.length >= 32 * (nSubContracts+1), "SUB_CONTRACTS_NOT_PROVIDED")`.

Recommendation Revise the sanity checks in the above `initialize()` routine to get the `EIC` address accounted for.

Status The issue has been fixed by the following commit: `3ebee9b`.

3.6 Redundant Check Removal in UpdateState::rootUpdate()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: UpdateState
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The StarkPerpetual protocol has a dedicated `operator` that is authorized to submit state updates for a batch of exchange transactions that aim to keep track of the state of the off-chain exchange service. The state is in essence saved by recording the Merkle roots of the vault state (off-chain account state) and the order state (including fully executed and partially fulfilled orders).

There is a designated function, i.e., `updateState()`, that handles the submitted state updates. Note the processing of the submitted state updates only occurs when the contract is not in the `frozen` state. And the state updates include the `publicInput` of a `STARK` proof, and additional data that includes information not attested to by the proof.

To elaborate, we show below an internal helper named `rootUpdate()` that actually records the new roots of the vault state and the order state.

```
    function rootUpdate(
        uint256 oldVaultRoot,
        uint256 newVaultRoot,
        uint256 oldOrderRoot,
        uint256 newOrderRoot,
        uint256 vaultTreeHeightSent,
        uint256 orderTreeHeightSent,
        uint256 batchId
    )
        internal
        notFrozen()
    {
        // Assert that the old state is correct.
        require(oldVaultRoot == vaultRoot, "VAULT_ROOT_INCORRECT");
        require(oldOrderRoot == orderRoot, "ORDER_ROOT_INCORRECT");

        // Assert that heights are correct.
        require(vaultTreeHeight == vaultTreeHeightSent, "VAULT_HEIGHT_INCORRECT");
        require(orderTreeHeight == orderTreeHeightSent, "ORDER_HEIGHT_INCORRECT");

        // Update state.
        vaultRoot = newVaultRoot;
        orderRoot = newOrderRoot;
        sequenceNumber = sequenceNumber + 1;
        lastBatchId = batchId;
```

```
        // Log update.
        emit LogRootUpdate(sequenceNumber, batchId, vaultRoot, orderRoot);
    }
```

Listing 3.9: UpdateState::rootUpdate()

We notice that this `rootUpdate()` helper has a `notFrozen()` modifier, which seems redundant as the upper-level caller, i.e., `updateState()`, performs the same check on `notFrozen()`.

Recommendation Remove the redundant check on `notFrozen()` in the `rootUpdate()` helper.

Status The issue has been fixed by the following commit: `3ebee9b`.

3.7 Explicit Block Of Logic Contract Initialization

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MainDispatcherBase`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.5, the StarkPerpetual implementation takes a proxy-based approach where the proxy contract is deployed at the front-end while the logic contract contains the actual business logic implementation. Specifically, it takes a `delegatecall`-based proxy pattern so that each component is split into two contracts: a back-end logic contract (that holds the implementation) and a front-end proxy (that contains the data and uses delegatecall to interact with the logic contract). From the user's perspective, they interact with the proxy while the code is executed on the logic contract.

In this section, we focus on the `initialize()` routine in the logic contract implementation, i.e., `MainDispatcherBase`. Our analysis aims to validate whether it suffers from a possible denial-of-service issue that may affect the protocol-wide normal operations. To illustrate, we show below the `initialize()` routine.

```
    // NOLINTNEXTLINE: external-function.
    function initialize(bytes memory data) public {
        // Number of sub-contracts.
        uint256 nSubContracts = getNumSubcontracts();

        // We support currently 4 bits per contract, i.e. 16, reserving 00 leads to 15.
        require(nSubContracts <= 15, "TOO_MANY_SUB_CONTRACTS");
```

```
        // Init data MUST include addresses for all sub-contracts.
        require(data.length >= 32 * nSubContracts, "SUB_CONTRACTS_NOT_PROVIDED");

        // Ensure implementation is a valid contract.
        require(implementation().isContract(), "INVALID_IMPLEMENTATION");

        // Size of passed data, excluding sub-contract addresses.
        uint256 additionalDataSize = data.length - 32 * (nSubContracts + 1);

        // Sum of subcontract initializers. Aggregated for verification near the end.
        uint256 totalInitSizes = 0;

        // Offset (within data) of sub-contract initializer vector.
        // Just past the sub-contract addresses.
        uint256 initDataContractsOffset = 32 * (nSubContracts + 1);

        // Extract & update contract addresses.
        for (uint256 nContract = 1; nContract <= nSubContracts; nContract++) {
            address contractAddress;

            // Extract sub-contract address.
            // solium-disable-next-line security/no-inline-assembly
            assembly {
                contractAddress := mload(add(data, mul(32, nContract)))
            }

            validateSubContractIndex(nContract, contractAddress);

            // Contracts are indexed from 1 and 0 is not in use here.
            setSubContractAddress(nContract, contractAddress);
        }

        // Check if we have an external initializer contract.
        address externalInitializerAddr;

        // 2. Extract sub-contract address, again. It's cheaper than reading from
            storage.
        // solium-disable-next-line security/no-inline-assembly
        assembly {
            externalInitializerAddr := mload(add(data, mul(32, add(nSubContracts, 1))))
        }

        // 3(a). Yield to EIC initialization.
        if (externalInitializerAddr != address(0x0)) {
            callExternalInitializer(data, externalInitializerAddr, additionalDataSize);
            return;
        }

        // 3(b). Subcontracts initialization.
        // I. If no init data passed besides sub-contracts, return.
        if (additionalDataSize == 0) {
            return;
```

```
161         }

163         // Just to be on the safe side.
164         assert(externalInitializerAddr == address(0x0));

166         // II. Gate further initialization.
167         initializationSentinel();
168         ...
169 }
```

Listing 3.10: MainDispatcherBase:: initialize ()

Our focus is related to the support of so-called `external initializing contract` or `EIC`. When there is a need for a custom initialization, a specific EIC is deployed to perform the intended initialization. In particular, after the extraction and assignment of sub-contracts addresses, the `initialize` routine in `MainDispatcherBase` yields its execution to the EIC, skipping the rest of its initialization code.

Moreover, it comes to our attention that the functions on the logic contract, i.e., `MainDispatcherBase` , can be invoked directly, including the `initialize()` routine. This is typically expected as the states are stored in the proxy contract, not the logic contract. However, a malicious actor may call directly the `initialize()` routine (without go through the front-end proxy) and further provide a crafted `EIC`. By design, the `EIC` is executed in the context of `MainDispatcherBase`. If the crafted `EIC` simply performs `selfdestruct`, the logic implementation will be essentially removed. Meanwhile, the front-end proxy continues to delegate the calls to the logic contract without any code. Note that a `delegatecall` to a contract without code would return success without executing any code. We emphasize that the front-end proxy will return success, even though no code was executed!

Fortunately, there is a validation check at line 114 that ensures the current implementation is a contract. The direct call of `initialize()` on the logic contract fails the check, hence subverting the attempt to directly initialize the logic contract! From another perspective, this `initialize()` routine is crucial and there is no restriction on the intended caller. It is helpful to add necessary authorization on possible callers.

Recommendation Safeguard the `initialize()` routine to explicitly prevent it from being called by arbitrary users.

Status The issue has been fixed by the following commit: `3ebee9b`.

4 Conclusion

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

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

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

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

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

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

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

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