SMART CONTRACT AUDIT REPORT

for

IDEX Ikon Protocol

Prepared By: Xiaomi Huang

PeckShield

August 15, 2023

Document Properties

Client	IDEX
Title	Smart Contract Audit Report
Target	Ikon Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 15, 2023	Xuxian Jiang	Final Release
1.0-rc	May 13, 2023	Luck Hu	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the IDEX Ikon protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. 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 Ikon Protocol

The IDEX Ikon's key innovation is the introduction of perpetual futures, enabling high-performance, leveraged trading backed by smart contract fund custody. This Ikon release includes updated contracts as well as off-chain infrastructure and discontinues the use of the earlier version (Silverton)'s hybrid liquidity. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Ikon Protocol

Item	Description
Name	IDEX
Website	https://linktr.ee/idexofficial
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 15, 2023

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

- https://github.com/idexio/idex-contracts-ikon (94726de9)

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

- https://github.com/idexio/idex-contracts-ikon (fbd50eb)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

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

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

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

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Ikon Protocol` implementations. 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	5	■■■■■
Low	1	■
Informational	0	
Total	6	

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. 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 5 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Deny-of-Service in _validateInsuranceFundCannotLiquidateWallet()	Coding Practices	Fixed
PVE-002	Medium	Transfer to Exited Wallet in transfer_delegatecall()	Business Logic	Fixed
PVE-003	Low	Enhanced Caller Validation for sgReceive()	Coding Practices	Fixed
PVE-004	Medium	Properly Update of exchange in finalizeExchangeUpgrade()	Business Logic	Fixed
PVE-005	Medium	Improved Signature Hashes in Hashing	Coding Practices	Fixed
PVE-006	Medium	Trust Issue on Admin Keys	Security Features	Confirmed

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 Potential Deny-of-Service in _validateInsuranceFundCannotLiquidateWallet()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple libraries`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

`Ikon` introduces a process called automatic deleveraging (`ADL`), which is used by `Ikon` to close open positions directly against the select counterparty positions. Because of this, `ADL` provides a backstop of system solvency when liquidation is not an option. As one precondition of the `Wallet Exited` deleveraging, it requires the insurance fund (`IF`) wallet cannot liquidate the wallet in maintenance via the standard `Wallet Exited` liquidation. While examining the logic to check whether the `IF` wallet can liquidate a wallet in maintenance, we notice the existence of a possible `deny-of-service` issue.

In the following, we show the code snippet of the `_validateInsuranceFundCannotLiquidateWallet()` routine from the `WalletExitAcquisitionDeleveraging` library. As the name indicates, this routine is used to ensure the `IF` wallet cannot liquidate the underwater wallet via the standard `Wallet Exited` liquidation. It builds an union of base asset symbols for all the markets where the liquidating or `IF` wallets have open positions (line 312). Then it loops the union markets and calls the `_validateExitQuoteQuantity()` routine (line 338) to validate the quote quantities that are used to liquidate the liquidating wallet. In particular, if the liquidating wallet doesn't have open position in one of the union markets, the retrieved `balanceStruct` is null (`balanceStruct.balance = 0`).

```
299  function _validateInsuranceFundCannotLiquidateWallet(
300    AcquisitionDeleverageArguments memory arguments,
301    WalletExitAcquisitionDeleveragePriceStrategy deleveragePriceStrategy,
302    address insuranceFundWallet,
```

```
303     int64 liquidatingWalletTotalAccountValue,
304     uint64 liquidatingWalletTotalMaintenanceMarginRequirement,
305     BalanceTracking.Storage storage balanceTracking,
306     mapping(address => string[]) storage baseAssetSymbolsWithOpenPositionsByWallet,
307     mapping(string => mapping(address => MarketOverrides)) storage
        marketOverridesByBaseAssetSymbolAndWallet,
308     mapping(string => Market) storage marketsByBaseAssetSymbol
309   ) private {
310     // Build array of union of open position base asset symbols for both liquidating and
         IF wallets. Result of merge
311     // will already be de-duped and sorted
312     string[] memory baseAssetSymbolsForInsuranceFundAndLiquidatingWallet =
        baseAssetSymbolsWithOpenPositionsByWallet[
313       insuranceFundWallet
314     ].merge(baseAssetSymbolsWithOpenPositionsByWallet[arguments.liquidatingWallet]);
315     ...
316     Balance memory balanceStruct;
317     Market memory market;
318     // Loop through open position union and populate argument struct fields
319     for (uint8 i = 0; i < baseAssetSymbolsForInsuranceFundAndLiquidatingWallet.length; i
        ++) {
320       // Load market
321       market = marketsByBaseAssetSymbol[
          baseAssetSymbolsForInsuranceFundAndLiquidatingWallet[i]];
322       validateInsuranceFundCannotLiquidateWalletArguments.markets[i] = market;

324       balanceStruct = balanceTracking.loadBalanceStructAndMigrateIfNeeded(
325         arguments.liquidatingWallet,
326         market.baseAssetSymbol
327       );

329       validateExitQuoteQuantityArguments.indexPrice = market.lastIndexPrice;
330       validateExitQuoteQuantityArguments.liquidationBaseQuantity = balanceStruct.balance
          ;
331       validateExitQuoteQuantityArguments.liquidationQuoteQuantity = arguments
332         .validateInsuranceFundCannotLiquidateWalletQuoteQuantities[i];
333       validateExitQuoteQuantityArguments.maintenanceMarginFraction = market
334         .loadMarketWithOverridesForWallet(arguments.liquidatingWallet,
            marketOverridesByBaseAssetSymbolAndWallet)
335         .overridableFields
336         .maintenanceMarginFraction;

338       _validateExitQuoteQuantity(balanceStruct, validateExitQuoteQuantityArguments);
339     }
340     ...
341   }
```

Listing 3.1: _validateInsuranceFundCannotLiquidateWallet()

Within the `_validateExitQuoteQuantity()` routine, it validates the quote quantity per the given strategies. For the `ExitPrice` strategy, it first calculates the cost basis of the base quantity being liquidated by calling the `Math.multiplyPipsByFraction()` routine (line 277). However, it comes to our attention that if `balanceStruct.balance = 0` (line 281), the transaction will revert because of division by zero (line 59). For the `BankruptcyPrice` strategy, it calls the `LiquidationValidations.validateQuoteQuantityAtBankruptcyPrice()` routine (line 288) to validate the quote quantity. In particular, if `balanceStruct.balance = 0`, `liquidationBaseQuantity = 0` (line 290), and the calculated `expectedLiquidationQuoteQuantity = 0` (line 165), the transaction will also revert because of arithmetic underflow (line 175). Based on this, we suggest to validate the quote quantities only for the markets where the liquidating wallet has open positions.

Note the same issue is also applicable to the `_validateInsuranceFundCannotLiquidateWallet()` routine from the `WalletInMaintenanceAcquisitionDeleveraging` library.

```
270     function _validateExitQuoteQuantity(
271         Balance memory balanceStruct,
272         ValidateExitQuoteQuantityArguments memory arguments
273       ) private pure {
274         if (arguments.deleveragePriceStrategy ==
              WalletExitAcquisitionDeleveragePriceStrategy.ExitPrice) {
275           LiquidationValidations.validateQuoteQuantityAtExitPrice(
276             // Calculate the cost basis of the base quantity being liquidated while
                  observing signedness
277             Math.multiplyPipsByFraction(
278               balanceStruct.costBasis,
279               arguments.liquidationBaseQuantity,
280               // Position size implicitly validated non-zero by `Validations.
                    loadAndValidateActiveMarket`
281               int64(Math.abs(balanceStruct.balance))
282             ),
283             arguments.indexPrice,
284             arguments.liquidationBaseQuantity,
285             arguments.liquidationQuoteQuantity
286           );
287         } else {
288           LiquidationValidations.validateQuoteQuantityAtBankruptcyPrice(
289             arguments.indexPrice,
290             arguments.liquidationBaseQuantity,
291             arguments.liquidationQuoteQuantity,
292             arguments.maintenanceMarginFraction,
293             arguments.totalAccountValue,
294             arguments.totalMaintenanceMarginRequirement
295           );
296         }
297       }
```

Listing 3.2: _validateExitQuoteQuantity()

```
54  function multiplyPipsByFraction(
55      int64 multiplicand,
56      int64 fractionDividend,
57      int64 fractionDivisor
58    ) internal pure returns (int64) {
59      int256 result = (int256(multiplicand) * fractionDividend) / fractionDivisor;
```

```
    require(result <= type(int64).max, "Pip quantity overflows int64");
    require(result >= type(int64).min, "Pip quantity underflows int64");

    return int64(result);
  }
```

Listing 3.3: Math:multiplyPipsByFraction()

```
function validateQuoteQuantityAtBankruptcyPrice(
    uint64 indexPrice,
    int64 liquidationBaseQuantity,
    uint64 liquidationQuoteQuantity,
    uint64 maintenanceMarginFraction,
    int64 totalAccountValue,
    uint64 totalMaintenanceMarginRequirement
  ) internal pure {
    uint64 expectedLiquidationQuoteQuantity = calculateQuoteQuantityAtBankruptcyPrice(
      indexPrice,
      maintenanceMarginFraction,
      liquidationBaseQuantity,
      totalAccountValue,
      totalMaintenanceMarginRequirement
    );

    // Allow additional pip buffers for integer rounding
    require(
      expectedLiquidationQuoteQuantity - 1 <= liquidationQuoteQuantity &&
        expectedLiquidationQuoteQuantity + 1 >= liquidationQuoteQuantity,
      "Invalid quote quantity"
    );
  }
```

Listing 3.4: LiquidationValidations :validateQuoteQuantityAtBankruptcyPrice()

Recommendation Revisit the above `_validateInsuranceFundCannotLiquidateWallet()` routine and validate the quote quantities only for the markets where the liquidating wallet has open positions.

Status The issue has been fixed by this commit: `7fa1e7c`.

3.2 Transfer to Exited Wallet in transfer_delegatecall()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Transferring`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

`Ikon` provides the ability for wallets to transfer quote funds directly to other wallets within the `Exchange`. Transfers are subject to the same constraints as withdrawals for the source wallets, including margin requirements and gas fees. Meanwhile, transfers shall also be subject to the same constraints as deposits for the destination wallets. While reviewing the constraints for transfers, we notice users can transfer quote funds to exited wallets which is not allowed in deposits.

To elaborate, we show below the code snippets of the `Transferring::transfer_delegatecall()/Depositing::deposit_delegatecall()` routines. As the name indicates, the `Transferring::transfer_delegatecall()` routine is used to transfer quote assets from the source wallet to the destination wallet. At the beginning of the routine, it validates the source and destination wallets. Specifically, the destination wallet shall be a valid wallet and cannot be the exit fund (`EF`) wallet (lines 42 – 43). However, we notice there is a lack of validation for the destination wallet to ensure it is not an exited wallet. As a result, a user may transfer quote assets to an exited wallet, which is not allowed in deposit (line 35).

Based on this, we suggest to add a validation in the `Transferring::transfer_delegatecall()` routine to ensure the destination wallet is not an exited wallet, i.e., `require(!walletExits[arguments.transfer.destinationWallet].exists)`.

```
26    function transfer_delegatecall(
27      Arguments memory arguments,
28      BalanceTracking.Storage storage balanceTracking,
29      mapping(address => string[]) storage baseAssetSymbolsWithOpenPositionsByWallet,
30      mapping(bytes32 => bool) storage completedTransferHashes,
31      mapping(string => FundingMultiplierQuartet[]) storage
          fundingMultipliersByBaseAssetSymbol,
32      mapping(string => uint64) storage
          lastFundingRatePublishTimestampInMsByBaseAssetSymbol,
33      mapping(string => mapping(address => MarketOverrides)) storage
          marketOverridesByBaseAssetSymbolAndWallet,
34      mapping(string => Market) storage marketsByBaseAssetSymbol,
35      mapping(address => WalletExit) storage walletExits
36    ) public returns (int64 newSourceWalletExchangeBalance) {
37      require(!WalletExits.isWalletExitFinalized(arguments.transfer.sourceWallet,
          walletExits), "Wallet exited");
```

```
    require(arguments.transfer.sourceWallet != arguments.exitFundWallet, "Cannot transfer from EF");
    require(arguments.transfer.sourceWallet != arguments.insuranceFundWallet, "Cannot transfer from IF");

    require(arguments.transfer.destinationWallet != address(0x0), "Invalid destination wallet");
    require(arguments.transfer.destinationWallet != arguments.exitFundWallet, "Cannot transfer to EF");

    require(
      Validations.isFeeQuantityValid(arguments.transfer.gasFee, arguments.transfer.grossQuantity),
      "Excessive transfer fee"
    );
    ...
  }
```

Listing 3.5: Transferring :: transfer_delegatecall ()

```
    function deposit_delegatecall(
        ICustodian custodian,
        uint64 depositIndex,
        address destinationWallet,
        address exitFundWallet,
        uint256 quantityInAssetUnits,
        address quoteTokenAddress,
        address sourceWallet,
        BalanceTracking.Storage storage balanceTracking,
        mapping(address => WalletExit) storage walletExits
      ) public returns (uint64 quantity, int64 newExchangeBalance) {
        // Deposits are disabled until `setDepositIndex` is called successfully
        require(depositIndex != Constants.DEPOSIT_INDEX_NOT_SET, "Deposits disabled");
        require(destinationWallet != exitFundWallet, "Cannot deposit to EF");

        // Calling exitWallet disables deposits immediately on mining, in contrast to withdrawals and trades which respect
        // the Chain Propagation Period given by `effectiveBlockNumber` via `_isWalletExitFinalized`
        require(!walletExits[sourceWallet].exists, "Source wallet exited");
        require(!walletExits[destinationWallet].exists, "Destination wallet exited");

        ...
      }
```

Listing 3.6: Depositing :: deposit_delegatecall ()

Recommendation Revisit the validations in the `Transferring::transfer_delegatecall()` routine and ensure the destination wallet is not an exited wallet.

Status The issue has been fixed by this commit: `0fe3d4a`.

3.3 Enhanced Caller Validation for sgReceive()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ExchangeStargateAdapter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In order to support seamless cross-chain deposits and withdrawals, `Ikon` includes an extensible set of adapter contracts for `Stargate` bridge protocol integration. For deposits, adapters receive bridged funds and call `Exchange`'s deposit function with the provided destination wallet address. While reviewing the logic in the `sgReceive()` routine which is used as a callback function for the `Stargate` router to notify the adapters about receiving new bridged funds, we notice the possibility of funds loss if the `sgReceive()` can be called by anybody.

To elaborate, we show below the related code snippet of the `ExchangeStargateAdapter::sgReceive()` routine. Normally, when the `Stargate` bridge receives new bridged funds, the `Stargate` router transfers the funds to the related adapter, and next calls the `sgReceive()` of the adapter with a specified gas. In case the external call to the `sgReceive()` may fail due to some reasons like out-of-gas, the funds are transferred to the adapter but the call to the `sgReceive()` is not completed. However, the router caches the call to the `sgReceive()` and allows anybody to clear the cached calls with no specific gas limit. In this way, it can finally notify each message of the bridged funds to the related adapter.

However, it comes to our attention that the `sgReceive()` doesn't properly validate the caller. As a result, if the bridged funds are locked in the adapter, before anybody tries to clear the cached messages in the router, anybody can call the `sgReceive()` with a faked payload (destination address) to deposit the funds to the `Exchange` for the faked wallet.

Based on this, we suggest to add a proper validation for the caller in the `sgReceive()` routine and only allow the `Stargate` router to call this function.

```
  function sgReceive(
      uint16 /* chainId */,
      bytes calldata /* srcAddress */,
      uint256 /* nonce */,
      address token,
      uint256 amountLD,
      bytes memory payload
    ) public override {
      require(isDepositEnabled, "Deposits disabled");

      require(token == address(quoteAsset), "Invalid token");
```

```
        address destinationWallet = abi.decode(payload, (address));
        require(destinationWallet != address(0x0), "Invalid destination wallet");

        IExchange(custodian.exchange()).deposit(amountLD, destinationWallet);
    }
```

Listing 3.7: ExchangeStargateAdapter::sgReceive()

Recommendation Properly validate the caller of the `sgReceive()` routine and only allow the `Stargate` router to call it.

Status The issue has been fixed by this commit: `b805bb0`.

3.4 Properly Update of exchange in finalizeExchangeUpgrade()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Governance`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Ikon` protocol, the `Governance` contract implements the contract upgrade logic while enforcing governance constraints. It allows the whitelisted `admin` to upgrade the `Exchange` contract and configure the new `Exchange`. While reviewing the logic to upgrade the `Exchange`, we notice it doesn't update the `exchange` state variable in the `Governance` contract.

In the following, we show the related code snippet of the `Governance::finalizeExchangeUpgrade()` routine. As the name indicates, it is the last step to finalize an `Exchange` upgrade. It simply updates the `Exchange` for the `Custodian` which holds user funds (line 234). However, we notice it doesn't update the local `exchange` state variable, which is used by `admin` to configure the `Exchange`. If the `exchange` state is not updated to the new `Exchange`, all the configurations to the `exchange` are applied to the old `Exchange`, and the new `Exchange` misses the configurations.

Based on this, we suggest to update the `exchange` state to the new `Exchange` also in the `Governance::finalizeExchangeUpgrade()` routine.

```
228     function finalizeExchangeUpgrade(address newExchange) public onlyAdmin {
229         require(currentExchangeUpgrade.exists, "No Exchange upgrade in progress");
230         require(currentExchangeUpgrade.newContract == newExchange, "Address mismatch");
231         require(block.number >= currentExchangeUpgrade.blockThreshold, "Block threshold
            not yet reached");
232
```

```
        address oldExchange = address(exchange);
        custodian.setExchange(newExchange);
        delete currentExchangeUpgrade;

        emit ExchangeUpgradeFinalized(oldExchange, newExchange);
    }
```

Listing 3.8: `Governance::finalizeExchangeUpgrade()`

Recommendation Revisit the `Governance::finalizeExchangeUpgrade()` routine and update the `exchange` state to the new `Exchange`.

Status The issue has been fixed by this commit: `3851d49`.

3.5 Improved Signature Hashes in Hashing

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Hashing`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Ikon` protocol has a core library, i.e., `Hashing`, that aims to be a helper for building hashes and verifying wallet signatures. While reviewing the functions to build hashes, we notice the lack of protection to prevent the signatures from possible replay.

To elaborate, we show below the code snippet of the `Hashing::getTransferHash()` routine. As the name indicates, this routine is used to build a hash for the input transfer request. The hash is built simply by encoding the content of the transfer request. However, it comes to our attention that current implementation is missing a proper domain separator, hence making the signature validation susceptible to replay attacks across different chains/contracts. The domain separator usually contains the `block.chainid` and the contract address (`address(this)`), which can limit the signature to be valid only in the current chain and contract.

Moreover, another better practice is to add a function separator to the signature hash, which is used to identify the functionality the signature is signed for. For example, in the `ERC20Permit` contact of the `OpenZeppelin` lib, it defines the `_PERMIT_TYPEHASH`, i.e., `bytes32 private constant _PERMIT_TYPEHASH =keccak256("Permit(...)")`, for the `IERC20Permit-permit` function. The function separator can prevent a signature for one function to be replayed in other functions.

Based on this, we suggest to add a proper domain separator and a function separator into each of the hashes.

```
function getTransferHash(Transfer memory transfer) internal pure returns (bytes32) {
    require(transfer.signatureHashVersion == Constants.SIGNATURE_HASH_VERSION, "
        Signature hash version invalid");

    return
      keccak256(
        abi.encodePacked(
          transfer.signatureHashVersion,
          transfer.nonce,
          transfer.sourceWallet,
          transfer.destinationWallet,
          _pipToDecimal(transfer.grossQuantity)
        )
      );
  }
```

Listing 3.9: `Hashing::getTransferHash()`

Note this issue is applicable to all the hash building routines in the `Hashing` library.

Recommendation Revisit the hash building routines in the `Hashing` library and add proper domain separator and function separators into the hashes.

Status The issue has been fixed in these commits: `77b7080`, `dd4be27`, `ed7ed02`, `c795e4c`, `4b484ce`, and `9f6c27f`.

3.6 Trust Issue on Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Ikon` protocol, there are certain privileged accounts, i.e., `owner/admin/dispatcher`, that play critical roles in regulating the protocol-wide operations (e.g., add market, publish index price, upgrade exchange). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `Exchange` contract as an example and show the representative functions potentially affected by the privileges of the privileged accounts.

Specifically, the privileged functions in `Exchange` allow for the `admin` to set the `custodian` which custodies user funds, set the fee wallet which receives protocol fees, change or remove the dispatcher wallet, add market and provide price feed for the market, etc.

What's more, the `dispatcher` wallet is authorized to call operator-only contract functions: publish index prices, publish funding multiplier, execute trade, activate market, deactivate market, etc.

```
    function setCustodian(ICustodian newCustodian, IBridgeAdapter[] memory newBridgeAdapters) public onlyAdmin {
        require(custodian == ICustodian(payable(address(0x0))), "Custodian can only be set once");
        require(Address.isContract(address(newCustodian)), "Invalid address");

        custodian = newCustodian;

        for (uint8 i = 0; i < newBridgeAdapters.length; i++) {
          require(Address.isContract(address(newBridgeAdapters[i])), "Invalid adapter address");
        }

        bridgeAdapters = newBridgeAdapters;
    }

    function setFeeWallet(address newFeeWallet) public onlyAdmin {
        require(newFeeWallet != address(0x0), "Invalid fee wallet address");
        require(newFeeWallet != feeWallet, "Must be different from current");

        address oldFeeWallet = feeWallet;
        feeWallet = newFeeWallet;

        emit FeeWalletChanged(oldFeeWallet, newFeeWallet);
    }

    function setDispatcher(address newDispatcherWallet) public onlyAdmin {
        require(newDispatcherWallet != address(0x0), "Invalid wallet address");
        require(newDispatcherWallet != dispatcherWallet, "Must be different from current");
        dispatcherWallet = newDispatcherWallet;
    }

    function removeDispatcher() public onlyAdmin {
        dispatcherWallet = address(0x0);
    }

    function addMarket(Market memory newMarket) public onlyAdmin {
        MarketAdmin.addMarket_delegatecall(
          newMarket,
          fundingMultipliersByBaseAssetSymbol,
          lastFundingRatePublishTimestampInMsByBaseAssetSymbol,
          marketsByBaseAssetSymbol
        );
    }

    function publishIndexPrices(IndexPrice[] memory indexPrices) public onlyDispatcher {
        MarketAdmin.publishIndexPrices_delegatecall(indexPrices, indexPriceServiceWallets, marketsByBaseAssetSymbol);
```

```
    }

    function publishFundingMultiplier(
        string memory baseAssetSymbol,
        int64 fundingRate
      ) public onlyDispatcherWhenExitFundHasNoPositions {
        Funding.publishFundingMultiplier_delegatecall(
          baseAssetSymbol,
          fundingRate,
          fundingMultipliersByBaseAssetSymbol,
          lastFundingRatePublishTimestampInMsByBaseAssetSymbol,
          marketsByBaseAssetSymbol
        );

        emit FundingRatePublished(baseAssetSymbol, fundingRate);
    }

    function activateMarket(string memory baseAssetSymbol) public
        onlyDispatcherWhenExitFundHasNoPositions {
        MarketAdmin.activateMarket_delegatecall(baseAssetSymbol,
            marketsByBaseAssetSymbol);
    }

    function deactivateMarket(string memory baseAssetSymbol) public
        onlyDispatcherWhenExitFundHasNoPositions {
        MarketAdmin.deactivateMarket_delegatecall(baseAssetSymbol,
            marketsByBaseAssetSymbol);
    }
```

Listing 3.10: Example Privileged Operations in `Exchange`

We understand the need of the privileged functions for proper operations, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the `Ikon` users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged accounts to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `IDEX Ikon` protocol. `Ikon`'s key innovation is the introduction of perpetual futures, enabling high-performance, leveraged trading backed by smart contract fund custody. The `Ikon` release includes updated contracts as well as off-chain infrastructure and discontinues the use of `Silverton`'s hybrid liquidity. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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