SMART CONTRACT AUDIT REPORT

for

Holdstation

Prepared By: Xiaomi Huang

PeckShield

January 2, 2023

Document Properties

Client	Holdstation
Title	Smart Contract Audit Report
Target	Holdstation
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 2, 2023	Xuxian Jiang	Final Release
1.0-rc	December 28, 2023	Xuxian Jiang	Release Candidate

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 `Holdstation` 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 Holdstation

`Holdstation` is a self-custodial smart wallet that uses `zkSync` native `Account Abstraction (ERC-4337)` to optimize `UX` and enhance security for users. `Holdstation` uses `DPF (Dynamic Price Feed)` and single `USDC` vault to create unlimited scalability, allowing users to trade multiple trading pairs in multiple markets supported by the protocol. This model has only one general-purpose trading pool (i.e. one single smart contract) implementing the pricing and funding fee logic. And `80%` of `Holdstation`'s trading is shared with the `USDC` vault and `$HOLD` stakers. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Holdstation

Item	Description
Target	Holdstation
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	January 2, 2023

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Note that the protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://gitlab.com/hspublic/contract-holdstation-dex.git (3c84b89)

And these are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://gitlab.com/hspublic/contract-holdstation-dex.git (647242b)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

Table 1.3: 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Holdstation` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	3	■ ■ ■
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 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 high-severity vulnerability, 2 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Holdstation Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Signature Validation in NativeMetaTransaction	Coding Practices	Confirmed
PVE-002	Medium	Incoherent handleGoldGovFees Convention in HSTradingCallbacks	Coding Practices	Resolved
PVE-003	Low	Incorrect Deposit Logic in HSTradingVault	Business Logic	Resolved
PVE-004	Low	Incompatibility Between Multicall And ContextMixin	Coding Practices	Resolved
PVE-005	High	Revisited Market-Closing Logic in HSTradingCallbacks	Business Logic	Confirmed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved Signature Validation in NativeMetaTransaction

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NativeMetaTransaction`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `Holdstation` protocol has the built-in support of meta transactions, which allow a third party `Relayer` send the transaction on behalf of the user and pay for the gas fees. The built-in support inevitably requires the proper authentication of user signature. While examining the signature verification, we notice the current implementation can be improved.

To elaborate, we show below the related routine `verify()`. This routine ensures that the given `signer` is indeed the one who signs the transaction request. Note that the internal implementation makes use of the `ecrecover()` precompile for validation. It comes to our attention that the precompile-based validation needs to properly ensure the uncovered `signer`, is not equal to `address(0)`.

```
function verify(
  address signer,
  MetaTransaction memory metaTx,
  bytes32 sigR,
  bytes32 sigS,
  uint8 sigV
) internal view returns (bool) {
  require(signer != address(0), "NativeMetaTransaction: INVALID_SIGNER");
  return signer == ecrecover(toTypedMessageHash(hashMetaTransaction(metaTx)), sigV,
      sigR, sigS);
}
```

Listing 3.1: `NativeMetaTransaction::verify()`

Recommendation Strengthen the `verify()` routine to ensure `signer` is not equal to `address(0)`.

Status The issue has been confirmed and will be updated in the next release.

3.2 Incoherent handleGoldGovFees Convention in HSTradingCallbacks

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HSTradingCallbacks`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `Holdstation` protocol has a key `HSTradingCallbacks` contract that is designed to perform actual trading functionalities, including the governance fee collection. While analyzing the fee collection logic, we notice it makes use of an inconsistent function call convention.

In the folowing, we show the implementation of the `updateSlCallback()` routine that calls `storageT.handleGoldGovFees()` to manage the governance fees. It comes to our attention that this `handleGoldGovFees()` function has five arguments and the fourth argument is an `address` type. However, in `updateSlCallback()`, the calls to `storageT.handleGoldGovFees()` (lines 489-490) has the fourth argument in `boolean`. Note this issue affects a number of other routines, including `openTradeMarketCallback()`, `closeTradeMarketCallback()`, `registerTrade()`, and `updateSlCallback()`.

```
473  function updateSlCallback(AggregatorAnswer memory a) external onlyPriceAggregator notDone {
474    AggregatorInterfaceV6 aggregator = storageT.priceAggregator();
475    AggregatorInterfaceV6.PendingSl memory o = aggregator.pendingSlOrders(a.orderId);
476
477    StorageInterfaceV5.Trade memory t = storageT.openTrades(o.trader, o.pairIndex, o.index);
478
479    if (t.leverage > 0) {
480      StorageInterfaceV5.TradeInfo memory i = storageT.openTradesInfo(o.trader, o.pairIndex, o.index);
481
482      Values memory v;
483
484      v.tokenPriceUsdc = aggregator.tokenPriceUsdc();
485      v.levPosUsdc = (t.initialPosToken * i.tokenPriceUsdc * t.leverage) / PRECISION / 2;
486
487      // Charge in USDC if collateral in storage or token if collateral in vault
488      v.reward1 = t.positionSizeUsdc > 0
489        ? storageT.handleGoldGovFees(t.pairIndex, v.levPosUsdc, 0, true, false)
```

```
      : (storageT.handleGoldGovFees(t.pairIndex, (v.levPosUsdc * PRECISION) / v.
          tokenPriceUsdc, 0, false, false) *
        v.tokenPriceUsdc) / PRECISION;
    ...
  }
}
```

Listing 3.2: `HSTradingCallbacks::updateSlCallback()`

```
function handleGoldGovFees(
  uint256 _pairIndex,
  uint256 _leveragedPositionSize,
  uint256 _referralFee,
  address _trader,
  bool _fullFee
) external onlyTrading returns (uint256 fee) {
  fee = (_leveragedPositionSize * priceAggregator.openFeeP(_pairIndex)) / PRECISION /
      100;
  if (!_fullFee) {
    fee /= 2;
  }
  uint256 goldFeePaid = (fee * goldFeeP) / PRECISION;
  goldFeesUsdc += goldFeePaid;
  uint256 agencyFee = 0;
  if (_referralFee == 0 && address(hsAgency) != address(0)) {
    agencyFee = hsAgency.distributeReward(2 * fee - goldFeePaid, _trader);
  }
  govFeesUsdc += (2 * fee - goldFeePaid - _referralFee - agencyFee);
  fee = fee * 2 - _referralFee;
}
```

Listing 3.3: `HSTradingStorage::handleGoldGovFees()`

Recommendation Revise the above routines to provide intended arguments to handle governance fees.

Status The issue has been addressed in the following commit: `647242b`.

3.3 Incorrect Deposit Logic in HSTradingVault

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HSTradingVault`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

To facilitate user participation, `Holdstation` has a key `HSTradingVault` contract. Users may deposit funds and provide liquidity into the protocol. While examining the deposit logic, we notice an issue that needs to be addressed.

To elaborate, we show below two affected routines, i.e., `deposit()` and `mint()`. We notice the user deposit will change the user balance, which makes it necessary to timely notify and update reward accounting (via `tokenCredit.notifyBalanceChange()` – line 503). Note the notification should be used to update the reward related to the user who has the balance change. In the `deposit()` routine, the user who receives the balance change is the `receiver`, not `_msgSender()` (line 503). In the `mint()` routine, the notification call is completely missing.

```
496   function deposit(uint256 assets, address receiver) public override checks(assets,
          false) returns (uint256) {
497     require(assets <= maxDeposit(receiver), "ERC4626: deposit more than max");
498     uint256 shares = previewDeposit(assets);
499     scaleVariables(shares, assets, true);
500
501     _deposit(_msgSender(), receiver, assets, shares);
502     if (address(tokenCredit) != address(0)) {
503       tokenCredit.notifyBalanceChange(_msgSender(), address(0));
504     }
505     return shares;
506   }
507
508   function mint(uint256 shares, address receiver) public override checks(shares, false)
          returns (uint256) {
509     require(shares <= maxMint(receiver), "ERC4626: mint more than max");
510
511     uint256 assets = previewMint(shares);
512     scaleVariables(shares, assets, true);
513
514     _deposit(_msgSender(), receiver, assets, shares);
515     return assets;
516   }
```

Listing 3.4: `HSTradingVault::deposit()/mint()`

Recommendation Revise the above-mentioned routines to properly notify and update user rewards or credits.

Status The issue has been confirmed. And the team clarifies that users are unable to directly interact as they are not exposed via UI. As part of design, if any user intentionally violates this, they may not receive any credit.

3.4 Incompatibility Between Multicall And ContextMixin

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HSToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

Description

As mentioned earlier, the `Holdstation` protocol has the built-in support of meta transactions, which allow a third party `Relayer` send the transaction on behalf of the user. In the meantime, it also supports multicall to facilitate the user interaction. Unfortunately, the simultaneous use of `Multicall` and meta transactions may come with a so-called address spoofing risk if the underlying implementation is not carefully engineered.

To elaborate, we show below the related code snippet of two related routines, i.e., `aggregate()` and `msgSender()`. While each routine is rather straightforward and achieves the intended functionality, the combined use allows for the complete spoofing of the received `msgSender()`. In particular, if a call is originated from a trusted forwarder, the actual caller's address is extracted from the last 20 bytes of the calldata. However, the `aggregate()` routine does not properly propagate the caller adjustment into each internal call. The detailed description of this issue can be found here: https://blog.openzeppelin.com/arbitrary-address-spoofing-vulnerability-erc2771context-multicall-public-disclosure.

```
function aggregate(Call[] calldata calls) public payable returns (uint256 blockNumber,
    bytes[] memory returnData) {
  blockNumber = block.number;
  uint256 length = calls.length;
  returnData = new bytes[](length);
  Call calldata call;
  for (uint256 i = 0; i < length; ) {
    bool success;
    call = calls[i];
    (success, returnData[i]) = call.target.call(call.callData);
    require(success, "Multicall3: call failed");
    unchecked {
      ++i;
```

```
      }
    }
  }
```

Listing 3.5: `Multicall3::aggregate()`

```
abstract contract ContextMixin {
  function msgSender() internal view returns (address sender) {
    if (msg.sender == address(this)) {
      bytes memory array = msg.data;
      uint256 index = msg.data.length;
      assembly {
        // Load the 32 bytes word from memory with the address on the lower 20 bytes, and mask those.
        sender := and(mload(add(array, index)), 0xffffffffffffffffffffffffffffffffffffffff)
      }
    } else {
      sender = msg.sender;
    }
    return sender;
  }
}
```

Listing 3.6: `ContextMixin::msgSender()`

Recommendation Revise the multicall functions to ensure the caller will not be spoofed.

Status The issue has been resolved as `HSToken` is the only affected contract, which is no longer used.

3.5 Revisited Market-Closing Logic in HSTradingCallbacks

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `HSTradingCallbacks`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

The `Holdstation` protocol provides a non-custodial derivative trading service with normal functions to open and close user positions. While examining the close-related logic, we notice the implementation may not use the latest price feed.

In the following, we examine an example routine `closeTradeMarketCallback()`. As the name indicates, this routine is designed to close a user position. We notice there is an internal variable `levPosUsdc` which represents current leveraged position (of the given user) denominated in

USDC. This variable is calculated as (t.initialPosToken * i.tokenPriceUsdc * t.leverage)/ PRECISION (line 263). Our analysis shows that this calculation uses the stale token price tokenPriceUsdc that was saved when the position is opened. Note the same issue affects another related routine, i.e., executeNftCloseOrderCallback().

```
246 function closeTradeMarketCallback(AggregatorAnswer memory a) external onlyPriceAggregator notDone {
247   StorageInterfaceV5.PendingMarketOrder memory o = storageT.reqID_pendingMarketOrder(a.orderId);
248
249   if (o.block == 0) {
250     return;
251   }
252
253   StorageInterfaceV5.Trade memory t = storageT.openTrades(o.trade.trader, o.trade.pairIndex, o.trade.index);
254
255   if (t.leverage > 0) {
256     StorageInterfaceV5.TradeInfo memory i = storageT.openTradesInfo(t.trader, t.pairIndex, t.index);
257
258     AggregatorInterfaceV6 aggregator = storageT.priceAggregator();
259     PairsStorageInterfaceV6 pairsStorage = aggregator.pairsStorage();
260
261     Values memory v;
262
263     v.levPosUsdc = (t.initialPosToken * i.tokenPriceUsdc * t.leverage) / PRECISION;
264     v.tokenPriceUsdc = aggregator.tokenPriceUsdc();
265     ...
266 }
```

Listing 3.7: HSTradingCallbacks::closeTradeMarketCallback()

In the meantime, when a user trade is unregistered, the protocol computes the remaining USDC value (usdcLeftInStorage) by deducting various fees from the position value. The purpose here is to credit the trader if the trade is winning or make necessary charges if the trade is losing. We notice this remaining USDC amount is currently computed as usdcLeftInStorage = currentUsdcPos - v.reward3 - v.reward2 (line 648), which should be revised as follows: usdcLeftInStorage = currentUsdcPos - v.reward3 - v.reward2 - v.reward1.

Recommendation Revise above routines to compute the right position value with latest oracle prices.

Status The issue has been confirmed and will be updated in the next release.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Holdstation` protocol, there is a privileged account (`manager` or `gov`) that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and price oracle adjustment). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
function setPairParams(uint256 pairIndex, PairParams memory value) public onlyManager {
  storeAccRolloverFees(pairIndex);
  storeAccFundingFees(pairIndex);

  pairParams[pairIndex] = value;

  emit PairParamsUpdated(pairIndex, value);
}

function setPairParamsArray(uint256[] memory indices, PairParams[] memory values) external onlyManager {
  require(indices.length == values.length, "WRONG_LENGTH");

  for (uint256 i = 0; i < indices.length; i++) {
    setPairParams(indices[i], values[i]);
  }
}

// Set one percent depth for pair
function setOnePercentDepth(uint256 pairIndex, uint256 valueAbove, uint256 valueBelow) public onlyManager {
  PairParams storage p = pairParams[pairIndex];

  p.onePercentDepthAbove = valueAbove;
  p.onePercentDepthBelow = valueBelow;

  emit OnePercentDepthUpdated(pairIndex, valueAbove, valueBelow);
}

function setOnePercentDepthArray(
  uint256[] memory indices,
  uint256[] memory valuesAbove,
  uint256[] memory valuesBelow
```

```
) external onlyManager {
  require(indices.length == valuesAbove.length && indices.length == valuesBelow.length
      , "WRONG_LENGTH");

  for (uint256 i = 0; i < indices.length; i++) {
    setOnePercentDepth(indices[i], valuesAbove[i], valuesBelow[i]);
  }
}

// Set rollover fee for pair
function setRolloverFeePerBlockP(uint256 pairIndex, uint256 value) public onlyManager
    {
  require(value <= 25000000, "TOO_HIGH");

  storeAccRolloverFees(pairIndex);

  pairParams[pairIndex].rolloverFeePerBlockP = value;

  emit RolloverFeePerBlockPUpdated(pairIndex, value);
}

function setRolloverFeePerBlockPArray(uint256[] memory indices, uint256[] memory
    values) external onlyManager {
  require(indices.length == values.length, "WRONG_LENGTH");

  for (uint256 i = 0; i < indices.length; i++) {
    setRolloverFeePerBlockP(indices[i], values[i]);
  }
}

// Set funding fee for pair
function setFundingFeePerBlockP(uint256 pairIndex, uint256 value) public onlyManager {
  require(value <= 10000000, "TOO_HIGH");

  storeAccFundingFees(pairIndex);

  pairParams[pairIndex].fundingFeePerBlockP = value;

  emit FundingFeePerBlockPUpdated(pairIndex, value);
}
```

Listing 3.8: Privileged Operations in `HSPairInfos`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed 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 by the team. For the time being, it is managed with a multi-sig (2/3) account at address `0x6471A875f55E5A1887f738aB128b3C7dc04CeB57`. The related tx is located here: https://explorer.zksync.io/tx/0x058bebbbbf786523793c323163bfc8588245252a31da047974d6dd0ca75b5c58
.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Holdstation` protocol, which is a self-custodial smart wallet and uses `zkSync` native `Account Abstraction (ERC-4337)` to optimize `UX` and enhance security for users. `Holdstation` uses `DPF (Dynamic Price Feed)` and single `USDC` vault to create unlimited scalability, allowing users to trade multiple trading pairs in multiple markets supported by the protocol. This model has only one general-purpose trading pool (i.e. one single smart contract) implementing the pricing and funding fee logic. And `80%` of `Holdstation`'s trading is shared with the `USDC` vault and `$HOLD` stakers. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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