SMART CONTRACT AUDIT REPORT

for

Symmetry

Prepared By: Xiaomi Huang

PeckShield

June 26, 2023

Document Properties

Client	Symmetry
Title	Smart Contract Audit Report
Target	Symmetry
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 26, 2023	Stephen Bie	Final Release
1.0-rc	June 7, 2023	Stephen Bie	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 Symmetry 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 Symmetry

Symmetry is a decentralized exchange providing transparent, accessible, and efficient trading experiences for derivatives trading. It efficiently mitigates LPs risk, maximizes capital efficiency, and utilizes portfolio margin, which makes institutional adoptions possible. Symmetry uses a dynamic funding rate mechanism for LPs risk mitigation and price stability. Meanwhile, a unified underlying settlement and portfolio margin effectively increases capital efficiency. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Symmetry

Item	Description
Target	Symmetry
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 26, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the Symmetry protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit. Additionally, this is not an economic audit and the correctness/reasoning of the funding rate and price formula is not part of this audit.

- https://github.com/symmetrytrade/symmetry-contracts.git (3f3f3d2)

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

- https://github.com/symmetrytrade/symmetry-contracts.git (cbca287)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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) [9], 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 `Symmetry` 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	0	
Medium	3	■■■
Low	1	■
Informational	1	■
Total	5	

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 3 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Symmetry Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Deficit Loss Payment in PositionManager::liquidatePosition()	Business Logic	Fixed
PVE-002	Medium	Revisited Logic of Market::_logTrade()	Business Logic	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Informational	Meaningful Events for Important State Changes	Coding Practices	Fixed
PVE-005	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 Revisited Deficit Loss Payment in PositionManager::liquidatePosition()

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: PositionManager
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the Symmetry protocol, the PositionManager contract is designed to manage the user's LONG/SHORT position. In particular, the liquidatePosition() routine is designed to liquidate the user's position. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the PositionManager contract. Inside the liquidatePosition() routine, it firstly closes the liquidated position (line 345). Next, if the position's current margin is negative, the deficit will be paid by the protocol insurance and/or LPs (lines 349-355). However, we notice the returned currentMargin (i.e., portfolio margin) (line 349) is shared by all the positions of the user rather than the liquidated position. That is to say, it may pay the deficit for the user's open position, which is against the protocol design and user expectation.

```
332     function liquidatePosition(address _account, address _token, bytes[] calldata
            _priceUpdateData) external payable {
333         IMarket market_ = IMarket(market);
334         // update oracle price
335         if (_priceUpdateData.length > 0) {
336             IPriceOracle(market_.priceOracle()).updatePythPrice{value: msg.value}(msg.
                    sender, _priceUpdateData);
337         }
338         // update fees
339         market_.updateFee(_token);
340         // validate liquidation
```

```
        require(isLiquidatable(_account), "PositionManager: account is not liquidatable"
            );
        // compute liquidation price
        (int liquidationPrice, int size, int notionalLiquidated) = market_.
            computePerpLiquidatePrice(_account, _token);
        // close position
        market_.trade(_account, _token, size, liquidationPrice);
        // update global info
        market_.updateTokenInfo(_token);
        // post trade margin
        (, int currentMargin, ) = market_.accountMarginStatus(_account);
        // fill the exceeding loss from insurance account
        int deficitLoss;
        if (currentMargin < 0) {
            deficitLoss = -currentMargin;
            _coverDeficitLoss(_account, deficitLoss);
        }
        ...
    }
```

Listing 3.1: `PositionManager::liquidatePosition()`

Recommendation Properly pay the deficit for the liquidated position.

Status The issue has been addressed in the following commit: `6ce6c82`.

3.2 Revisited Logic of Market::_logTrade()

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Market`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

In the `Symmetry` protocol, the `Market` contract is one of the main entries for user interactions. In particular, one entry routine, i.e., `trade()`, is designed to open/close a `LONG/SHORT` position. While examining its logic, we notice a common internal `_logTrade()` routine needs to be improved.

To elaborate, we show below the related code snippet of the `Market` contract. By design, the protocol will charge a certain trading fee when opening/closing a `LONG/SHORT` position. Especially, part of the trading fee will be distributed to the holders of the `veABF` token. The `_logTrade()` routine is designed to meet the requirement. Inside the routine, we notice the `tokenToUsd()` is incorrectly called (line 399) to calculated the `baseToken` amount used as incentives, which directly undermines the assumption of the protocol design. Given this, we suggest to improve the implementation as below: `uint amountToDistribute = usdToToken(baseToken, int(_fee), false).multiplyDecimal (IMarketSettings(settings).getIntVals(VESYM_FEE_INCENTIVE_RATIO)).toUint256()` (line 399).

```
    function trade(address _account, address _token, int _sizeDelta, int _price)
        external onlyOperator returns (int) {
        IPerpTracker perpTracker_ = IPerpTracker(perpTracker);

        require(perpTracker_.latestUpdated(_token) == block.timestamp, "Market: fee is
            not updated");

        (int execPrice, uint tradingFee, uint couponUsed) = IFeeTracker(feeTracker).
            discountedTradingFee(
            _account,
            _sizeDelta,
            _price,
            true
        );

        // trade
        (int marginDelta, int oldSize, int newSize) = perpTracker_.settleTradeForUser(
            _account,
            _token,
            _sizeDelta,
            execPrice
        );
        _modifyMargin(_account, usdToToken(baseToken, marginDelta, false));
        liquidityBalance += usdToToken(
            baseToken,
            perpTracker_.settleTradeForLp(_token, -_sizeDelta, execPrice, oldSize,
                newSize),
            false
        );

        // log
        _logTrade(_account, _sizeDelta.multiplyDecimal(_price).abs().toUint256(),
            tradingFee - couponUsed);

        emit Traded(_account, _token, _sizeDelta, _price, tradingFee, couponUsed);
        return execPrice;
    }
```

Listing 3.2: `Market::trade()`

```
397     function _logTrade(address _account, uint _volume, uint _fee) internal {
398         // veSYM incentives
399         uint amountToDistribute = tokenToUsd(baseToken, int(_fee), false)
400             .multiplyDecimal(IMarketSettings(settings).getIntVals(
                    VESYM_FEE_INCENTIVE_RATIO))
401             .toUint256();
402         _transferLiquidityOut(feeTracker, amountToDistribute);
403         IFeeTracker(feeTracker).distributeIncentives(amountToDistribute);
```

```
        // Volume
        VolumeTracker(volumeTracker).logTrade(_account, _volume);
    }
```

Listing 3.3: `Market::trade()`

Recommendation Improve the implementation of the `_logTrade()` routine as above-mentioned.

Status The issue has been addressed in the following commit: `fc58fc4`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VotingEscrow`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }
    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
```

```
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we show the `unstake()` routine. If the `USDT` token is supported as `baseToken`, the unsafe version of `IERC20(baseToken).transfer(msg.sender, _value)` (line 486) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value). We may intend to replace `IERC20(baseToken).transfer(msg.sender, _value)` (line 486) with `safeTransfer()`.

```
469     function unstake(uint _value) external nonReentrant {
470         _claimVested(msg.sender);
471 
472         uint stakedValue = staked[msg.sender];
473         require(stakedValue > _value, "VotingEscrow: insufficient staked");
474         stakedValue -= _value;
475         staked[msg.sender] = stakedValue;
476 
477         StakedPoint memory oldStaked = getLastStakedPoint(msg.sender);
478         StakedPoint memory newStaked = StakedPoint({
479             bias: 0,
480             slope: -SafeCast.toInt128(SafeCast.toInt256(stakedValue / maxTime)),
481             ts: block.timestamp,
482             end: _startOfWeek(block.timestamp + maxTime)
483         });
484         _checkpointStaked(msg.sender, oldStaked, newStaked);
485 
486         IERC20(baseToken).transfer(msg.sender, _value);
487 
488         _tryCallback(msg.sender);
489         emit Unstake(msg.sender, _value, block.timestamp);
490     }
```

Listing 3.5: VotingEscrow::unstake()

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`.

Status The issue has been addressed in the following commit: `f0170ad`.

3.4 Meaningful Events for Important State Changes

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the protocol dynamics, we notice there are several privileged routines that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    function setOperator(address _operator, bool _status) external onlyOwner {
        isOperator[_operator] = _status;
    }

    function setOracle(address _priceOracle) external onlyOwner {
        priceOracle = _priceOracle;
    }

    function setSetting(address _settings) external onlyOwner {
        settings = _settings;
    }
```

Listing 3.6: Market

With that, we suggest to emit meaningful events in these privileged routines. Also, the key event information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being indexed.

Recommendation Properly emit meaningful events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed in the following commit: 9b13c40.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the Symmetry protocol, there are a series of privileged accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters and mint ABF token). In the following, we show the representative functions potentially affected by the privilege of the accounts.

```
    function mint(address _to, uint _amount) external {
        require(hasRole(MINTER_ROLE, msg.sender), "ABF: must have minter role to mint");

        _mint(_to, _amount);
    }
```

Listing 3.7: ABF::mint()

```
    function mint(address _to, uint _amount) public virtual {
        require(hasRole(MINTER_ROLE, msg.sender), "LPToken: must have minter role to
            mint");
        _mint(_to, _amount);
    }
```

Listing 3.8: LPToken::mint()

```
    function setOperator(address _operator, bool _status) external onlyOwner {
        isOperator[_operator] = _status;
    }

    function setOracle(address _priceOracle) external onlyOwner {
        priceOracle = _priceOracle;
    }

    function setSetting(address _settings) external onlyOwner {
        settings = _settings;
    }
```

Listing 3.9: Market

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. The multi-sig mechanism could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest to introduce the `multi-sig` mechanism to manage all the privileged accounts to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status The issue has been confirmed by the team. The teams intends to introduce `multi-sig` and `timelock` mechanisms to mitigate this issue.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Symmetry`, which is a non-custodial decentralized derivative exchange providing transparent, accessible, and efficient trading experiences. It efficiently mitigates `LPs` risk, maximizes capital efficiency, and utilizes portfolio margin. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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