SMART CONTRACT AUDIT REPORT

for

Fort Finance

Prepared By: Yiqun Chen

Hangzhou, China
February 12, 2022

Document Properties

Client	Fort Finance
Title	Smart Contract Audit Report
Target	Fort Finance
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Jing Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 12, 2022	Jing Wang	Final Release
1.0-rc	January 31, 2022	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the `Fort Finance` design document and related smart contract source code, 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 branch of `Fort Finance` 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 Fort Finance

`Fort Finance` is a decentralized finance protocol which aims to protect investor portfolios using customized fortifications based upon the portfolio management needs of individual investors. By incorporating smart contract-powered value preservation strategies into their portfolios, users can manage their participation in `DeFi` markets within defined risk parameters, thereby optimizing the risk-reward outcomes. These strategies fall into three main categories: `Leverage Fortifications`, `Liquidity Fortifications` and `Price Fortifications`.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Fort Finance

Item	Description
Name	Fort Finance
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 12, 2022

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

- https://github.com/Fort-Finance/fortfinance-contracts.git (57f2d99)

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

- https://github.com/Fort-Finance/fortfinance-contracts.git (e8c4266)

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

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

Table 1.3: The Full List of Check Items

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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 1.4: 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 `Fort Finance` 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	2	■■
Low	3	■■■
Informational	0	
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 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Fort Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Handling Of fortificationStatus[parentId] Deletion	Time and State	Fixed
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-003	Low	Proper Handling Of cTokenbalanceBefore Calculation	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Medium	Proper Handling Of gasDeposits[account] Calculation	Security Features	Fixed

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

3 | Detailed Results

3.1 Proper Handling Of fortificationStatus[parentId] Deletion

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Core`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

To facilitate fortifications configuration, the `Fort Finance` protocol organizes fortifications with two different data structures `userFortifications` and `fortificationStatus`. In addition, the protocol defines a number of functions (e.g., `addFortification()`, `updateFortification()` and `deleteFortification()`) for users to interact with fortifications. In the following, we examine the `deleteFortification()` routine.

To elaborate, we show below the implementation of the `deleteFortification()` routine. As the name indicates, this routine is designed to delete the user fortifications. However, the current logic only deletes the related entry from `userFortifications`, but not from `fortificationStatus`. An improvement can be made to delete the related entry in `fortificationStatus`.

```
    function deleteFortification(bytes32 existingHash, bytes32[] calldata childHashes) external {
        if (userFortifications[msg.sender][existingHash] == 0) revert UnknownFortification();

        delete userFortifications[msg.sender][existingHash];

        for (uint256 i = 0; i < childHashes.length; i++) {
            bytes32 childHash = childHashes[i];
            uint256 childId = userFortifications[msg.sender][childHash];
            delete fortificationStatus[childId];
            delete userFortifications[msg.sender][childHash];
            emit FortificationDeleted(msg.sender, childHash);
```

```
        }

        emit FortificationDeleted(msg.sender, existingHash);
    }
```

Listing 3.1: Core::deleteFortification()

Recommendation Revise the above `deleteFortification()` logic to properly delete the related entry in `fortificationStatus`.

Status The issue has been fixed by this commit: `fc354a0`.

3.2 Accommodation of approve() Idiosyncrasies

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OneInchSwap`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [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 `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
```

```
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.2: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `OneInchSwap::OneinchSwap()` routine as an example. This routine is designed to approve the `_router` contract to swap `_srcToken` into `_dstToken`. To accommodate the specific idiosyncrasy, for each `approve()` (line 29), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Also, the `IERC20` interface has defined the `approve()` interface with a `bool` return value, but the above implementation does not have the return value. As a result, a normal `IERC20`-based `approve()` with a non-compliant token may unfortunately revert the transaction. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of current `approve()` in the `OneinchSwap::OneInchSwap()`.

```
18      function OneinchSwap(
19          address _account,
20          address _router,
21          address _caller,
22          address _srcToken,
23          uint256 _amount,
24          uint256 _minReturn,
25          bytes calldata _data,
26          IAggregationRouterV4.SwapDescription calldata _swapDescription
27      ) public returns (uint256 amountOut) {

29          require(IERC20(_srcToken).approve(_router, _amount), "approval failed (2)");

31          (amountOut,,) = IAggregationRouterV4(_router).swap(_caller, _swapDescription,
                _data);

33      }
```

Listing 3.3: `OneinchSwap::OneInchSwap()`

Note other routines including `OneInchSwap::UnoOneinchSwap()` and `OneInchSwap::UniswapV3inchSwap()` share the same issue.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status The issue has been partial fixed by this commit: `fc354a0`.

3.3 Proper Handling Of cTokenbalanceBefore Calculation

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `CompoundAdapter`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned before, the `Fort Finance` protocol provides several kinds of strategies to support user fortifications. Among the strategies, the `Leverage Fortifications` strategy is mainly implemented via the `CompoundAdapter` contract. While examining the `CompoundAdapter` contract, we notice there is a logic error in the `mintReceiptTokensToAccount()` routine. To elaborate, we show below the related code snippet.

```
283     function mintReceiptTokensToAccount(
284         address account,
285         address token,
286         uint256 amount
287     ) external override returns (uint256 ctokenAmount) {
288         // Retrieve CToken
289         ICERC20 cToken = ICERC20(UNISWAP_ANCHOREDVIEW.getTokenConfigBySymbol(IERC20(
                token).symbol()).cToken);

291         uint256 cTokenbalanceBefore = IERC20(token).balanceOf(address(this));

293         // ensure tokens are in the contract right now
294         require(IERC20(token).balanceOf(address(this)) >= amount, "Not enough token");

296         // approve token to cToken contract
297         require(IERC20(token).approve(address(cToken), amount), "approval failed (2)");

299         // deposit tokens in Compound
300         cToken.mint(amount);

302         ctokenAmount = cToken.balanceOf(address(this));

304         // move the resulting CTokens to account
305         cToken.transfer(account, ctokenAmount);
306     }
```

Listing 3.4: `CompoundAdapter::mintReceiptTokensToAccount()`

The `mintReceiptTokensToAccount()` routine (see the code snippet above) is provided to deposit tokens into `Compound` to acquire `CTokens`, which are then moved to the specific account. It comes to our attention that the balance calculation of `cTokenbalanceBefore` takes the balance of `token` (line

291) rather than `CTokens`. Hence, without properly calculating the balance of `cTokenbalanceBefore`, `ctokenAmount` may include the balance of `CTokens` left in the contract before and result a wrong value of `ctokenAmount`.

Recommendation Correct the above calculation of `cTokenbalanceBefore`.

Status The issue has been fixed by this commit: `fc354a0`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Core`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Fort Finance` protocol, there is a special administrative account, i.e., `admin`. This `admin` account plays a critical role in governing and regulating the protocol-wide operations (e.g., grant `OPERATOR_ROLE` and `EXECUTOR_ROLE`). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `admin` account and its related privileged accesses in current contract.

To elaborate, we show below the function provided to support gas fee deduction when a transaction reverted. Note this routine is guarded with `onlySentinel()` and the `EXECUTOR_ROLE` could withdraw any amount of gas fee from any account.

```
    // @dev only usable for Sentinel in case a tx reverted and they needto claim
    function spendGasSentinel(address account, uint256 amountInWei) external
        onlySentinel {
        spendGas(account, amountInWei);

        emit SpentGas(account, msg.sender, amountInWei, true);
    }

    /// @dev Decreases the gas balance for the user. Emits an event so accounting can be
         done offchain
    //           This function does NOT handle the actual WETH in the wallet
    function spendGas(address account, uint256 amount) internal {

        uint256 gasDeposit = gasDeposits[account];

        // Check if the account has enough gas deposits
        if (amount > gasDeposit) revert InsufficientGasDeposit(amount, gasDeposit);
```

```
        // Adjust the balance
        gasDeposits[account] = gasDeposits[account] - amount;

        // Send the ETH to executor
        payable(msg.sender).transfer(amount);

        // Apply fees
        uint256 fee = (amount * (platformFee)) / (100);
        if (fee > 0) payable(treasury).transfer(fee);

        // Emit event for offchain accounting
        emit SpentGas(account, msg.sender, amount, false);
    }
```

Listing 3.5: `Core::spendGasSentinel()&& spendGas()`

Also, we show below the `executeFortification()` function, which allows the `EXECUTOR_ROLE` role to execute a `delegatecall` by self-defined parameters.

```
    function executeFortification(
        address account,
        address swapPlatform,
        uint256 tokenAmount,
        uint256 indexOfApprovedToken,
        uint256 expiryBlockNumber,
        bytes calldata swapData,
        FortificationInfo calldata fortification
    ) public onlySentinel fortificationCanExecute(fortification) nonReentrant {
        ...

        // If the approved tokens were reinvested
        if (approvedToken.reinvestedPlatformAddress != address(0)) {


            if (!platformAdapters[approvedToken.reinvestedPlatformAddress])
                revert UnknownPlatform(approvedToken.reinvestedPlatformAddress);


            // Retrieve the approved tokens from the user's wallet to the contract
            IERC20Upgradeable(approvedToken.tokenAddress).safeTransferFrom(account,
                address(this), tokenAmount);

            (bool success, bytes memory data) = approvedToken.reinvestedPlatformAddress.
                delegatecall(
                abi.encodeWithSignature("withdraw(address,uint256)", approvedToken.
                    tokenAddress, tokenAmount)
            );

            if (!success) revert Withdraw(approvedToken.tokenAddress, tokenAmount);
        ...
    }
```

Listing 3.6: `Core::executeFortification()`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `admin` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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.

3.5 Proper Handling Of gasDeposits[account] Calculation

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Core`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

`Fort Finance` asks users to pre-fund a gas tank and sign token approvals in order to utilize `Fort Finance`'s automation tools. When a fortification is executed, `Fort Finance` will utilize the pre-funded gas to pay for gas fees, utilizing the necessary amounts of pre-selected funding source tokens that are pre-approved by the user. While examining the `spendGas` routine, we notice the fee sent to `treasury` is not deducted from `gasDeposits[account]` when spending the user gas. To elaborate, we show below the related code snippet.

```
    function spendGas(address account, uint256 amount) internal {

        uint256 gasDeposit = gasDeposits[account];

        // Check if the account has enough gas deposits
        if (amount > gasDeposit) revert InsufficientGasDeposit(amount, gasDeposit);

        // Adjust the balance
        gasDeposits[account] = gasDeposits[account] - amount;

        // Send the ETH to executor
        payable(msg.sender).transfer(amount);

        // Apply fees
        uint256 fee = (amount * (platformFee)) / (100);
```

```
225         if (fee > 0) payable(treasury).transfer(fee);

227         // Emit event for offchain accounting
228         emit SpentGas(account, msg.sender, amount, false);
229     }
```

Listing 3.7: `Core::spendGas()`

The `spendGas()` routine (see the code snippet above) is provided to decrease the gas balance for the user. It comes to our attention that the calculation of `gasDeposits[account]` only takes the `amount` (line 218) out from the balance. Hence, the `fee` sent to `treasury` is taken from the `Core` contract rather than the user.

Recommendation Correct the calculation of `gasDeposits[account]`.

Status The issue has been fixed by this commit: `eb68c10`.

4 Conclusion

In this audit, we have analyzed the `Fort Finance` design and implementation. `Fort Finance` is a decentralized finance protocol aims to protect investor portfolios using customized fortifications based upon the portfolio management needs of individual investors. 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.