SMART CONTRACT AUDIT REPORT

for

Chainhop

Prepared By: Xiaomi Huang

PeckShield

December 1, 2022

Document Properties

Client	ChainHop
Title	Smart Contract Audit Report
Target	Chainhop
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	December 1, 2022	Stephen Bie	Final Release
1.0-rc	November 28, 2022	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 **4**

1.1 About Chainhop 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Revisited Logic of ExecutionNode::execute() 11

3.2 Accommodation of Non-ERC20-Compliant Tokens 15

3.3 Improved Validation of Function Arguments 16

3.4 Trust Issue of Admin Keys 17

4 Conclusion **18**

References **19**

1 Introduction

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

`Chainhop` implements a cross-chain decentralized exchange (`DEX`), which provides the cross-chain swap service with a one-transaction user experience. It also implements the multi-bridge aggregation, including `cBridge`, `Multichain`, `Stargate`, and etc. It provides the user with unprecedented performance and flexibility. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Chainhop

Item	Description
Target	Chainhop
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	December 1, 2022

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

- https://github.com/chainhop-dex/chainhop-contracts.git (e90e50a)

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

- https://github.com/chainhop-dex/chainhop-contracts.git (3ed0fd2)

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

Likelihood

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

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

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 1.4: 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 `Chainhop` 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	1	■
Low	1	■
Informational	1	■
Total	4	

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

Table 2.1: Key Chainhop Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Revisited Logic of ExecutionNode::execute()	Business Logic	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Informational	Improved Validation of Function Arguments	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of 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 Revisited Logic of ExecutionNode::execute()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `ExecutionNode`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Chainhop` protocol implements a cross-chain decentralized exchange (`DEX`) and multi-bridge aggregation (including `cBridge`, `Multichain`, `Stargate`, and etc.). It allows the user to cross one token from the source chain to the intermediary chain, and then cross the token to the destination chain from the intermediary chain. Additionally, it allows the user to swap one token to another via the user specified `DEXes` on any chain (i.e., source chain, intermediary chain and destination chain). In particular, one entry routine, i.e., `execute()`, is designed to trigger the cross-chain transaction. While examining its logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `ExecutionNode` contract. In the following, we use a simple cross-chain transaction (from the source chain to the destination chain) as an example. When the user initiates a cross-chain transaction on the source chain via `execute()`, the assets will be transferred to a so-called `Pocket` address (a pre-calculated contract address based on the input `_id` and `exec.remoteExecutionNode`) (line 168) on the destination chain. And then the `execute()` routine on the destination chain will be called to transfer the assets to the recipient. Inside the `execute()` routine, it creates the `Pocket` contract based on the input `_id` firstly, secondly pulls the assets from the `Pocket` address and destroys the `Pocket` contract, and finally transfers the assets to the user specified `_dst.receiver`. After further analysis, we observe it is possible for a malicious actor to hijack the input `_id` and `_dst` parameters to transfer anyone else's cross-chain assets to himself. Given this, we suggest to whitelist the caller of the `execute()` routine to prevent the input parameters from being hijacked.

```
    function execute(
        bytes32 _id,
        Types.ExecutionInfo[] memory _execs,
        Types.SourceInfo memory _src,
        Types.DestinationInfo memory _dst
    ) public payable nonReentrant whenNotPaused returns (uint256 remainingValue) {
        require(_execs.length > 0, "nop");

        Types.ExecutionInfo memory exec;
        (exec, _execs) = _popFirst(_execs);

        remainingValue = msg.value;

        // pull funds
        uint256 amountIn;
        address tokenIn;
        if (_src.chainId == _chainId()) {
            // if there are more executions on other chains, verify sig so that we are
                sure the fees
            // to be collected will not be tempered with when we run those executions
            // note that quote sig verification is only done on the src chain. the
                security of each
            // subsequent execution's fee collection is dependent on the security of
                cbridge's IM
            if (_execs.length > 0) {
                _verify(_execs, _src, _dst);
            }
            (amountIn, tokenIn) = _pullFundFromSender(_src);
            if (_src.nativeIn) {
                remainingValue -= amountIn;
            }
        } else {
            (amountIn, tokenIn) = _pullFundFromPocket(_id, exec);
            // if amountIn is 0 after deducting fee, this contract keeps all amountIn as
                 fee and
            // ends the execution
            if (amountIn == 0) {
                emit StepExecuted(_id, 0, tokenIn);
                return remainingValue;
            }
            // refund immediately if receives bridge out fallback token
            if (tokenIn == exec.bridgeOutFallbackToken) {
                _sendToken(tokenIn, amountIn, _dst.receiver, false);
                emit StepExecuted(_id, amountIn, tokenIn);
                return remainingValue;
            }
        }

        // process swap if any
        uint256 nextAmount = amountIn;
        address nextToken = tokenIn;
        if (exec.swap.dex != address(0)) {
```

```
            bool success = true;
            (success, nextAmount, nextToken) = _executeSwap(exec.swap, amountIn, tokenIn
                );
            if (_src.chainId == _chainId()) require(success, "swap fail");
            // refund immediately if swap fails
            if (!success) {
                _sendToken(tokenIn, amountIn, _dst.receiver, false);
                emit StepExecuted(_id, amountIn, tokenIn);
                return remainingValue;
            }
        }

        // pay receiver if this is the last execution step
        if (_dst.chainId == _chainId()) {
            _sendToken(nextToken, nextAmount, _dst.receiver, _dst.nativeOut);
            emit StepExecuted(_id, nextAmount, nextToken);
            return remainingValue;
        }

        // funds are bridged directly to the receiver if there are no subsequent
            executions on the destination chain.
        // otherwise, it's sent to a "pocket" contract addr to temporarily hold the fund
             before it is used for swapping.
        address bridgeOutReceiver = (_execs.length > 0) ? _getPocketAddr(_id, exec.
            remoteExecutionNode) : _dst.receiver;
        _bridgeSend(exec.bridge, bridgeOutReceiver, nextToken, nextAmount);
        remainingValue -= exec.bridge.nativeFee;

        // if there are more execution steps left, pack them and send to the next chain
        if (_execs.length > 0) {
            bytes memory message = abi.encode(Types.Message({id: _id, execs: _execs, dst
                : _dst}));
            uint256 msgFee = IMessageBus(messageBus).calcFee(message);
            remainingValue -= msgFee;
            IMessageBus(messageBus).sendMessage{value: msgFee}(
                exec.remoteExecutionNode,
                exec.bridge.toChainId,
                message
            );
        }

        emit StepExecuted(_id, nextAmount, nextToken);
    }

    function _pullFundFromPocket(bytes32 _id, Types.ExecutionInfo memory _exec)
        private
        returns (uint256 amount, address token)
    {
        Pocket pocket = new Pocket{salt: _id}();

        uint256 fallbackAmount;
        if (_exec.bridgeOutFallbackToken != address(0)) {
```

```
            fallbackAmount = IERC20(_exec.bridgeOutFallbackToken).balanceOf(address(
                pocket)); // e.g. hToken/anyToken
        }
        uint256 erc20Amount = IERC20(_exec.bridgeOutToken).balanceOf(address(pocket));
        uint256 nativeAmount = address(pocket).balance;

        require(
            erc20Amount > _exec.bridgeOutMin
                nativeAmount > _exec.bridgeOutMin
                fallbackAmount > _exec.bridgeOutFallbackMin,
            "MSG::ABORT:pocket is empty"
        );
        if (fallbackAmount > 0) {
            pocket.claim(_exec.bridgeOutFallbackToken, fallbackAmount);
            amount = _deductFee(_exec.feeInBridgeOutFallbackToken, fallbackAmount);
            token = _exec.bridgeOutFallbackToken;
        } else {
            pocket.claim(_exec.bridgeOutToken, erc20Amount);
            if (erc20Amount > 0) {
                amount = _deductFee(_exec.feeInBridgeOutToken, erc20Amount);
            } else if (nativeAmount > 0) {
                require(_exec.bridgeOutToken == nativeWrap, "bridgeOutToken not
                    nativeWrap");
                amount = _deductFee(_exec.feeInBridgeOutToken, nativeAmount);
                IWETH(_exec.bridgeOutToken).deposit{value: amount}();
            }
            token = _exec.bridgeOutToken;
        }
    }
```

Listing 3.1: `ExecutionNode::execute()`

Recommendation Validate the caller of the `execute()` routine to prevent the input parameters from being hijacked.

Status The issue has been addressed by the following commits: `187ab9c`, `61a8d0d`, and `a89c814`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AcrossAdapter
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

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.

```
/**
* @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`

It is important to note that the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call may be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`. 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 `bridge()` routine in the `AcrossAdapter` contract. If the `USDT` token is supported as `_token`, the unsafe version of `IERC20(_token).approve(spokePool, _amount)` (line 38) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value).

```
29     function bridge(
30         uint64 _dstChainId,
31         address _receiver,
32         uint256 _amount,
33         address _token,
34         bytes memory _bridgeParams
35     ) external payable returns (bytes memory bridgeResp) {
36         BridgeParams memory params = abi.decode(_bridgeParams, (BridgeParams));
37         IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
38         IERC20(_token).approve(spokePool, _amount);
39         uint32 depositId = ISpokePool(spokePool).numberOfDeposits();
40         ...
41     }
```

Listing 3.3: `AcrossAdapter::bridge()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been addressed by the following commit: `5ee1f22`.

3.3 Improved Validation of Function Arguments

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DexRegistry`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

By design, the `DexRegistry` contract is used by the privileged `owner` to configure the codecs corresponding to the supported `DEXes`. Apparently, the `_setDexCodecs()` routine is designed to meet the requirement. While examining its logic, we observe the current implementation can be improved.

To elaborate, we show below the related code snippet of the contract. In the `_setDexCodecs()` routine, we notice it has the inherent assumption on the same length of the given three arrays, i.e., `_dexList`, `_funcs`, and `_codecs`. However, this is not enforced inside the `_setDexCodecs()` routine. For improvement, it is helpful to validate the length of these three arrays.

```
    function _setDexCodecs(address[] memory _dexList, string[] memory _funcs, address[]
        memory _codecs) private {
        for (uint256 i = 0; i < _dexList.length; i++) {
            bytes4 selector = bytes4(keccak256(bytes(_funcs[i])));
            _setDexCodec(_dexList[i], selector, _codecs[i]);
        }
    }
```

Listing 3.4: `DexRegistry::_setDexCodecs()`

Recommendation Validate the length of the input arrays inside the `_setDexCodecs()` routine.

Status The issue has been addressed by the following commit: `5ee1f22`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Chainhop` implementation, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setSigner(address _signer) public onlyOwner {
        _setSigner(_signer);
    }
```

Listing 3.5: `SigVerifier::setSigner()`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it will be worrisome if the privileged account is a plain EOA account. A multi-sig account 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 a multi-sig account plays the privileged `owner` account 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.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Chainhop` protocol, which implements a cross-chain decentralized exchange (`DEX`), which provides the cross-chain swap service with a one-transaction user experience. Additionally, it implements the multi-bridge aggregation, including `cBridge`, `Multichain`, `Stargate`, and etc. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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