SMART CONTRACT AUDIT REPORT

for

Nori Protocol

Prepared By: Xiaomi Huang

PeckShield

September 16, 2022

Document Properties

Client	Nori
Title	Smart Contract Audit Report
Target	Nori
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 16, 2022	Xuxian Jiang	Final Release
1.0-rc1	August 21, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

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

Contents

1 Introduction

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

With the goal of reversing climate change, the `Nori` protocol builds a better carbon marketplace with seamless transactions, transparent bookkeeping, and 100% verified and trusted carbon removal supply. It helps organizations, companies, and individuals meet their climate goals with high-quality carbon removal offsets. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Nori

Item	Description
Name	Nori
Website	https://nori.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 16, 2022

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

- https://github.com/nori-dot-eco/contracts.git (e146687)

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

- https://github.com/nori-dot-eco/contracts.git (e29f85e)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

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

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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

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

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

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

1.4 Disclaimer

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

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	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 implementation of the `Nori` protocol. 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	■■■
Undetermined	1	■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Nori Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Logic in RestrictedNORILib::_linearReleaseAmountAvailable()	Business Logic	Resolved
PVE-002	Low	Timely tokenHolders Update in RestrictedNORILib::withdrawFromSchedule()	Business Logic	Resolved
PVE-003	Undetermined	Revisited Logic in RestrictedNORILib::createSchedule()	Business Logic	Resolved
PVE-004	Medium	Improved Validation in Market And RemovalIdLib	Coding Practices	Resolved
PVE-005	Low	Incorrect Removal Retrieval Logic in RemovalsByYearLib	Business Logic	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides 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 in RestrictedNORILib::_linearReleaseAmountAvailable()

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `RestrictedNORILib`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The `Nori` protocol has a `RestrictedNORI` contract that basically wraps the `BridgedPolygonNORI` token contract with the capability of restricting the release of tokens for use as insurance collateral. To facilitate its logic, there is a helper function to compute the linearly released balance for a single schedule at the current block timestamp. Our analysis shows its current implementation needs to be improved.

To elaborate, we show below the related `_linearReleaseAmountAvailable()` function. It implements a rather straightforward logic in computing the linearly released balance for a single schedule at the current block timestamp while ignoring any released amount floor that has been set for the schedule. In particular, if the current block timestamp is larger than `schedule.endTime`, the expected linearly released balance for the given schedule is `schedule._scheduleTrueTotal(totalSupply)`, instead of the current `totalSupply` (line 61).

```
54  function _linearReleaseAmountAvailable(
55    Schedule storage schedule,
56    uint256 totalSupply
57  ) internal view returns (uint256) {
58    uint256 linearAmountAvailable;
59    /* solhint-disable not-rely-on-time, this is time-dependent */
60    if (block.timestamp >= schedule.endTime) {
61      linearAmountAvailable = totalSupply;
62    } else {
```

```
    uint256 rampTotalTime = schedule.endTime - schedule.startTime;
    linearAmountAvailable = block.timestamp < schedule.startTime
      ? 0
      : (schedule._scheduleTrueTotal(totalSupply) *
        (block.timestamp - schedule.startTime)) / rampTotalTime;
  }
  /* solhint-enable not-rely-on-time */
  return linearAmountAvailable;
}
```

Listing 3.1: `RestrictedNORILib::_linearReleaseAmountAvailable()`

Recommendation Properly revise the above `_linearReleaseAmountAvailable()` for the calculation of the linearly released balance.

Status The issue has been fixed by this commit: `933ca32`.

3.2 Timely tokenHolders Update in RestrictedNORILib::withdrawFromSchedule()

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

Description

As mentioned earlier, the `Nori` protocol has a `RestrictedNORI` contract that basically wraps the `BridgedPolygonNORI` token contract with the capability of restricting the release of tokens for use as insurance collateral. While reviewing the logic to claim the released tokens, we notice the current logic needs to be revisited.

To elaborate, we show below the related `withdrawFromSchedule()` function. As the name indicates, this function is used to claim the sender's released tokens and withdraw them to the given recipient address. It comes to our attention that when the calling user's balance drops to zero, there is a need to update the `tokenHolders` by removing the calling user.

```
function withdrawFromSchedule(
  address recipient,
  uint256 scheduleId,
  uint256 amount
) external returns (bool) {
  super._burn(_msgSender(), scheduleId, amount);
  Schedule storage schedule = _scheduleIdToScheduleStruct[scheduleId];
```

```
    schedule.totalClaimedAmount += amount;
    schedule.claimedAmountsByAddress[_msgSender()] += amount;
    emit TokensClaimed(_msgSender(), recipient, scheduleId, amount);
    _bridgedPolygonNORI.transfer(recipient, amount);
    return true;
  }
```

Listing 3.2: `RestrictedNORI::withdrawFromSchedule()`

Recommendation Properly revise the above `withdrawFromSchedule()` routine to remove the calling user if the token balance drops to zero.

Status The issue has been resolved as it allows for gas saving purposes.

3.3 Revisited Logic in RestrictedNORILib::createSchedule()

- ID: PVE-003
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `RestrictedNORI`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

As mentioned earlier, the `Nori` protocol has a `RestrictedNORI` contract that basically wraps the `BridgedPolygonNORI` token contract with the capability of restricting the release of tokens for use as insurance collateral. While reviewing the logic for setting up the restriction schedule, we notice the current logic needs to be revisited.

To elaborate, we show below the related `createSchedule()` function. It implements a rather straightforward logic in validating the given parameters and setting up a new restriction schedule. It comes to our attention that the current validation is insufficient as it does not ensure the given `projectId` is a new one. If an existing `projectId` is given, the current schedule may be re-configured, which may not be consistent with its design.

```
  function createSchedule(
    uint256 projectId,
    uint256 startTime,
    uint8 methodology,
    uint8 methodologyVersion
  ) external whenNotPaused onlyRole(SCHEDULE_CREATOR_ROLE) {
    uint256 restrictionDuration = getRestrictionDurationForMethodologyAndVersion({
        methodology: methodology,
        methodologyVersion: methodologyVersion
      });
```

```
    _validateSchedule({
      startTime: startTime,
      restrictionDuration: restrictionDuration
    });
    _createSchedule({
      projectId: projectId,
      startTime: startTime,
      restrictionDuration: restrictionDuration
    });
  }

  function _validateSchedule(uint256 startTime, uint256 restrictionDuration)
    internal
    pure
  {
    // todo this can probably be moved to the rNoriLib along with _createSchedule (if
        not, some schedule creator lib)
    require(startTime != 0, "rNORI: Invalid start time");
    require(restrictionDuration != 0, "rNORI: duration not set");
  }

  function _createSchedule(
    uint256 projectId,
    uint256 startTime,
    uint256 restrictionDuration
  ) internal {
    Schedule storage schedule = _scheduleIdToScheduleStruct[projectId];
    schedule.startTime = startTime;
    schedule.endTime = startTime + restrictionDuration;
    _allScheduleIds.add(projectId);
    emit ScheduleCreated(projectId, startTime, schedule.endTime);
  }
```

Listing 3.3: `RestrictedNORI::createSchedule()`

Recommendation Properly revise the above `createSchedule()` to ensure the `projectId` is a new one.

Status The issue has been fixed by this commit: `8fdc7c2`.

3.4 Improved Validation in Market And RemovalIdLib

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `RemovalIdLib`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Nori` protocol is no exception. Specifically, if we examine the `RemovalIdLib` contract, it has defined a number of removal-specific fields, such as `methodology` and `methodologyVersion`. In the following, we show the corresponding routine that creates a new removal.

```
function validate(DecodedRemovalIdV0 memory removal) internal pure {
  if (removal.idVersion != 0) {
    revert UnsupportedIdVersion({idVersion: removal.idVersion});
  }
  if (removal.methodologyVersion > 15) {
    revert MethodologyVersionTooLarge({
      methodologyVersion: removal.methodologyVersion
    });
  }
  if (
    !(isCapitalized(removal.country) && isCapitalized(removal.subdivision))
  ) {
    revert UncapitalizedString({
      country: removal.country,
      subdivision: removal.subdivision
    });
  }
}

/**
 * @notice Packs data about a removal into a 256-bit token id for the removal.
 * @dev Performs some possible validations on the data before attempting to create the
     id.
 *
 * @param removal removal data struct to be packed into a uint256 ID
 */
function createRemovalId(
  DecodedRemovalIdV0 memory removal // todo rename create
) internal pure returns (uint256) {
  removal.validate();
  uint256 methodologyData = (removal.methodology << 4) removal.methodologyVersion;return ...
```

Listing 3.4: `RemovalIdLib::createRemovalId()`

These fields define various aspects of the removal token and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `methodology` may accidently update another field `vintage` in the created `removal` token, hence compromising the integrity of the protocol design.

Moreover, the `Market` contract defines a number of parameters and one specific parameter `_noriFeePercentage` defines the fee percentage charged for every transaction. There is a need to ensure this fee percentage shall fall in the expected range, i.e., `require(noriFeePercentage_ < 100)`.

```
function setNoriFeePercentage(uint256 noriFeePercentage_)
  external
  onlyRole(MARKET_ADMIN_ROLE)
  whenNotPaused
{
  _noriFeePercentage = noriFeePercentage_;
  emit NoriFeePercentageUpdated(noriFeePercentage_);
}
```

Listing 3.5: `Market::setNoriFeePercentage()`

Recommendation Validate any changes regarding these parameters-related changes to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `8fdc7c2`.

3.5 Incorrect Removal Retrieval Logic in RemovalsByYearLib

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `RemovalsByYearLib`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The `Nori` protocol has a `Removal` contract that is an extended `ERC1155` token contract for carbon removal accounting. This contract contains a number of helper routines to expose internal dynamics of the carbon removal accounting. While reviewing one specific helper routine `getAllRemovalIds()`, we notice the current logic needs to be improved.

To elaborate, we show below the related `getAllRemovalIds()` function. As the name indicates, this function is used to get the total balance of all removals across all years. However, it comes to our attention that the current implementation only returns the balance of all removals in the earliest year, not all years!

```
function getAllRemovalIds(RemovalsByYear storage collection)
  internal
  view
  returns (uint256[] memory removalIds)
{
  uint256 latestYear = collection.latestYear;
  for (uint256 year = collection.earliestYear; year <= latestYear; ++year) {
    EnumerableSetUpgradeable.UintSet storage removalIdSet = collection
      .yearToRemovals[year];
    uint256[] memory ids = new uint256[](removalIdSet.length());
    uint256 numberOfRemovals = ids.length;
    // Skip overflow check as for loop is indexed starting at zero.
    unchecked {
      for (uint256 i = 0; i < numberOfRemovals; ++i) {
        ids[i] = removalIdSet.at(i);
      }
    }
    return ids;
  }
}
```

Listing 3.6: `RemovalsByYearLib::getAllRemovalIds()`

Recommendation Properly revise the above `getAllRemovalIds()` routine to properly compute the total balance of all removals across all years.

Status The issue has been fixed by this commit: `8fdc7c2`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SwopXLendingV3`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Nori` contract, there is a privileged account (with the `DEFAULT_ADMIN_ROLE` role) that plays a critical role in governing and regulating the system-wide operations (e.g., add new roles, configure various parameters, etc). 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 registerContractAddresses(
  Removal removal,
  Certificate certificate,
  BridgedPolygonNORI bridgedPolygonNORI,
  RestrictedNORI restrictedNORI
) external onlyRole(DEFAULT_ADMIN_ROLE) whenNotPaused {
  _removal = removal;
  _certificate = certificate;
  _bridgedPolygonNORI = bridgedPolygonNORI;
  _restrictedNORI = restrictedNORI;
  emit ContractAddressesRegistered(
    _removal,
    _certificate,
    _bridgedPolygonNORI,
    _restrictedNORI
  );
}

function setPriorityRestrictedThreshold(uint256 threshold)
  external
  whenNotPaused
  onlyRole(MARKET_ADMIN_ROLE)
{
  _priorityRestrictedThreshold = threshold;
  emit PriorityRestrictedThresholdSet(threshold);
}

function setNoriFeePercentage(uint256 noriFeePercentage_)
  external
  onlyRole(MARKET_ADMIN_ROLE)
  whenNotPaused
{
  _noriFeePercentage = noriFeePercentage_;
  emit NoriFeePercentageUpdated(noriFeePercentage_);
}
```

Listing 3.7: Various Privileged Functions in `Market`

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Moreover, it should be noted that if current contracts are planned to deploy behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed and the team clarifies that the current need of having centralized administration and appropriate controls in place for the admin keys.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Nori protocol, which is developed to reverse climate change by building a better carbon marketplace with seamless transactions, transparent bookkeeping, and 100% verified and trusted carbon removal supply. It helps organizations, companies, and individuals meet their climate goals with high-quality carbon removal offsets. The current code base can be further improved in both design and implementation. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that Solidity-based 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-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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