SMART CONTRACT AUDIT REPORT

for

Synthswap

Prepared By: Xiaomi Huang

PeckShield

May 11, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 11, 2023	Xuxian Jiang	Final Release
1.0-rc	May 7, 2023	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 source code of the `Synthswap` 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 could potentially be improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Synthswap

`Synthswap` is one of the first decentralized exchanges (`DEX`) with an automated market-maker (`AMM`) in the `zkSync Era` ecosystem. Compared to its competitors, `Synthswap` will enable trading with the lowest fees! Rewards from `Staking` and `Yield Farming` will be among the most lucrative. All this and more will be possible thanks to highly efficient concentrated liquidity. Uniquely to the `zkSync Era` ecosystem, `Synthswap` will offer its users active liquidity management. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Synthswap

Item	Description
Name	Synthswap
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 11, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit only covers the following contracts: `Dividends.sol`, `SynthToken.sol`, `XSynthToken.sol.sol` as well as contracts under the `farm` subdirectory.

- https://github.com/Zyberswap-Arbitrum/zyberswap-contracts.git (32d37f5)

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

- https://github.com/Zyberswap-Arbitrum/zyberswap-contracts.git (ff02bf02)

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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) [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 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 design and implementation of the `Synthswap` protocol smart contracts. 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 logics, 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	3	■■■
Informational	0	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others may involve 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 3 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic in MultipleRewards::add()	Coding Practices	Resolved
PVE-002	Low	Incomplete Constructor Logic in SynthChef	Coding Practices	Resolved
PVE-003	Low	Timely Distribution of Pending Reward In Existing Rewarders	Business Logic	Resolved
PVE-004	Medium	Incorrect Calculation of lpPercent in SynthChef	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Medium	Incorrect Pending Dividends Calculation in Dividends	Business Logic	Resolved

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

3 | Detailed Results

3.1 Improved Logic in MultipleRewards::add()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MultipleRewards`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Synthswap` protocol has a built-in `MultipleRewards` contract to support the dissemination of multiple rewards. While examining the logic to add a new reward pool, we notice a possible improvement over the current implementation.

In the following, we show below the pool-adding logic in the `add()` routine. We notice a new pool may not be added unless the associated `lastRewardTimestamp` is equal to 0, which implicitly requires the new pool's `lastRewardTimestamp`, i.e., the given `_startTimestamp`, should be non-zero. In other words, we also need to add the following requirement: `require( _startTimestamp != 0)`.

```
    function add(
        uint256 _pid,
        uint256 _allocPoint,
        uint256 _startTimestamp
    ) public onlyOperator {
        require(poolInfo[_pid].lastRewardTimestamp == 0, "pool already exists");
        totalAllocPoint += _allocPoint;

        poolInfo[_pid] = PoolInfo({
            allocPoint: _allocPoint,
            startTimestamp: _startTimestamp,
            lastRewardTimestamp: _startTimestamp,
            accTokenPerShare: 0,
            totalRewards: 0
        });

```

```
        poolIds.push(_pid);
        emit AddPool(_pid, _allocPoint);
    }
```

Listing 3.1: `MultipleRewards::add()`

Recommendation Improve the above `add()` logic to ensure a new pool will not be accidentally added twice.

Status This issue has been resolved in the following commit: `ff02bf02`.

3.2 Incomplete Constructor Logic in SynthChef

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SynthChef`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Synthswap` protocol features a `MasterChef`-like incentive mechanism with the `SynthChef` contract. In the process of analyzing the current incentive mechanism, we notice its constructor routine needs to be improved to properly initialize the `marketingPercent` parameter.

In the following, we show below the implementation of current constructor function. We notice a given input argument `_marketingPercent` is not actually used. In fact, this input argument is intended to initialize the `marketingPercent` parameter. Namely, the following assignment should be added into the constructor routine: `marketingPercent = _marketingPercent;`.

```
    constructor(
        IBoringERC20 _synth,
        uint256 _synthPerSec,
        address _marketingAddress,
        uint256 _marketingPercent,
        address _teamAddress,
        uint256 _teamPercent,
        address _feeAddress,
        IXSynthToken _xsynth
    ) {
        require(
            _marketingPercent <= 200,
            "constructor: invalid marketing percent value"
        );

        startTimestamp = block.timestamp + (60 * 60 * 24 * 365);
```

```

        synth = _synth;
        synthPerSec = _synthPerSec;
        marketingAddress = _marketingAddress;
        teamAddress = _teamAddress;
        feeAddress = _feeAddress;
        teamPercent = _teamPercent;
        xSynth = _xsynth;
        IERC20(address(_xsynth)).approve(address(_xsynth), type(uint256).max);
    }
```

Listing 3.2: `SynthChef::constructor()`

Recommendation Improve the above `constructor()` to properly initialize the `marketingPercent` parameter.

Status This issue has been resolved in the following commit: `ff02bf02`.

3.3 Timely Distribution of Pending Reward In Existing Rewarders

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `SynthChef`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `Synthswap` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the allocation percentage update routines `setMarketingPercent()`/`setTeamPercent()`, we notice the need of timely invoking `_massUpdatePools()` to update the reward distribution before the new allocation percentage becomes effective.

```
    function setMarketingPercent(
        uint256 _newmarketingPercent
    ) public onlyOwner {
        require(_newmarketingPercent <= 200, "invalid percent value");
        emit SetmarketingPercent(marketingPercent, _newmarketingPercent);
        marketingPercent = _newmarketingPercent;
```

```
    }

    function setTeamPercent(uint256 _newTeamPercent) public onlyOwner {
        require(_newTeamPercent <= 200, "invalid percent value");
        emit SetTeamPercent(teamPercent, _newTeamPercent);
        teamPercent = _newTeamPercent;
    }
```

Listing 3.3: `SynthChef::setMarketingPercent()/setTeamPercent()`

If the call to `massUpdatePools()` is not immediately invoked before updating the allocation percentage, certain situations may be crafted to create an unfair reward distribution.

Recommendation Timely invoke `_massUpdatePools()` when the reward allocation percentages are updated.

Status This issue has been resolved in the following commit: `ff02bf02`.

3.4 Incorrect Calculation of lpPercent in SynthChef

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SynthChef`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `SynthChef` contract is motivated from the initial `MasterChef` with certain extensions. One specific extension is the support of different allocations on `marketingPercent`, `lpPercent` and `teamPercent`. In the process of calculating current rewards rate, we notice its logic should be improved.

To elaborate, we show below the `poolRewardsPerSec()` routine. As the name indicates, this routine is used to compute the current pool rewards per second. Note that there are three components to receive the reward with respective percentages, i.e., `marketingPercent`, `lpPercent` and `teamPercent`. And the total sum should be equal to `total = 1000`. It comes to our attention that the current calculation of `lpPercent = total - marketingPercent` (line 417) is incorrect and should be revised as `lpPercent = total - marketingPercent - teamPercent`. Note the same issue is also applicable to another routine `pendingTokens()`.

```
392     function poolRewardsPerSec(
393         uint256 _pid
394     )
395         external
```

```
        view
        validatePoolByPid(_pid)
        returns (
            address[] memory addresses,
            string[] memory symbols,
            uint256[] memory decimals,
            uint256[] memory rewardsPerSec
        )
    {
        PoolInfo storage pool = poolInfo[_pid];

        addresses = new address[](pool.rewarders.length + 1);
        symbols = new string[](pool.rewarders.length + 1);
        decimals = new uint256[](pool.rewarders.length + 1);
        rewardsPerSec = new uint256[](pool.rewarders.length + 1);

        addresses[0] = address(synth);
        symbols[0] = IBoringERC20(synth).safeSymbol();
        decimals[0] = IBoringERC20(synth).safeDecimals();

        uint256 total = 1000;
        uint256 lpPercent = total - marketingPercent;

        rewardsPerSec[0] =
            (pool.allocPoint * synthPerSec * lpPercent) /
            totalAllocPoint /
            total;

        for (
            uint256 rewarderId = 0;
            rewarderId < pool.rewarders.length;
            ++rewarderId
        ) {
            addresses[rewarderId + 1] = address(
                pool.rewarders[rewarderId].rewardToken()
            );

            symbols[rewarderId + 1] = IBoringERC20(
                pool.rewarders[rewarderId].rewardToken()
            ).safeSymbol();

            decimals[rewarderId + 1] = IBoringERC20(
                pool.rewarders[rewarderId].rewardToken()
            ).safeDecimals();

            rewardsPerSec[rewarderId + 1] = pool
                .rewarders[rewarderId]
                .poolRewardsPerSec(_pid);
        }
    }
```

Listing 3.4: `SynthChef::poolRewardsPerSec()`

Recommendation Revise the above two routines to properly compute the pending rewards.

Status This issue has been resolved in the following commit: `ff02bf02`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Synthswap` protocol, there is a privileged `owner` account (with the privilege to assign other roles such as `operator`) that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and reward adjustment). 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 account and their related privileged accesses in current contracts.

```
    function emergencyWithdraw(IERC20 token) public nonReentrant onlyOwner {
        uint256 balance = token.balanceOf(address(this));
        require(balance > 0, "emergencyWithdraw: token balance is null");
        _safeTokenTransfer(token, msg.sender, balance);
    }

    /**
     * @dev Emergency withdraw all dividend tokens' balances on the contract
     */
    function emergencyWithdrawAll() external nonReentrant onlyOwner {
        for (uint256 index = 0; index < _distributedTokens.length(); ++index) {
            emergencyWithdraw(IERC20(_distributedTokens.at(index)));
        }
    }
```

Listing 3.5: Example Privileged Opeations in `Dividends`

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 team has confirmed that the admin keys will be run behind by a `timelock`.

3.6 Incorrect Pending Dividends Calculation in Dividends

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Dividends`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Dividends` contract has a helper routine `pendingDividendsAmount()` to calculate the pending dividends amount. Our analysis shows the current logic needs to be improved.

To elaborate, we show below its implementation. It has a rather straightforward logic in calculating the pending dividends amount for the given `userAddress`. However, the final return statement should properly deduct the current reward debt and add back the previously recorded `pendingDividends` for the given user. In other words, the final return statement should be revised as `(usersAllocation[userAddress] * accDividendsPerShare)/ 1e18 - users[token][userAddress].rewardDebt + users[token][userAddress].pendingDividends;`.

```
    function pendingDividendsAmount(
        address token,
        address userAddress
    ) external view returns (uint256) {
        if (totalAllocation == 0) {
            return 0;
        }

        DividendsInfo storage dividendsInfo_ = dividendsInfo[token];

        uint256 accDividendsPerShare = dividendsInfo_.accDividendsPerShare;
        uint256 lastUpdateTime = dividendsInfo_.lastUpdateTime;
        uint256 dividendAmountPerSecond_ = _dividendsAmountPerSecond(token);

        // check if the current cycle has changed since last update
        if (_currentBlockTimestamp() > nextCycleStartTime()) {
            // get remaining rewards from last cycle
            accDividendsPerShare +=
```

```
                (nextCycleStartTime() - lastUpdateTime) *
                ((dividendAmountPerSecond_ * 1e16) / totalAllocation);

            lastUpdateTime = nextCycleStartTime();
            dividendAmountPerSecond_ =
                ((dividendsInfo_.pendingAmount *
                    dividendsInfo_.cycleDividendsPercent) / 100) /
                _cycleDurationSeconds;
        }

        // get pending rewards from current cycle
        accDividendsPerShare +=
            (((_currentBlockTimestamp() - lastUpdateTime) *
                dividendAmountPerSecond_) * 1e16) /
            totalAllocation;

        return
            ((usersAllocation[userAddress] * accDividendsPerShare) / 1e18) -
            (users[token][userAddress].rewardDebt +
                users[token][userAddress].pendingDividends);
    }
```

Listing 3.6: `Dividends::pendingDividendsAmount()`

Recommendation Revise the above routine to properly compute the pending dividends amount.

Status This issue has been resolved in the following commit: `ff02bf02`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Synthswap` protocol, which is one of the first decentralized exchanges (`DEX`) with an automated market-maker (`AMM`) in the `zkSync Era` ecosystem. Compared to its competitors, `Synthswap` will enable trading with the lowest fees! Rewards from `Staking` and `Yield Farming` will be among the most lucrative. All this and more will be possible thanks to highly efficient concentrated liquidity. Uniquely to the `zkSync Era` ecosystem, `Synthswap` will offer its users active liquidity management. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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