SMART CONTRACT AUDIT REPORT

for

Velvet Capital

Prepared By: Xiaomi Huang

PeckShield

August 23, 2022

Document Properties

Client	Velvet
Title	Smart Contract Audit Report
Target	Velvet
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 23, 2022	Xuxian Jiang	Final Release
1.0-rc	July 18, 2022	Xuxian Jiang	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 Velvet Capital	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	Improper Corner Case Handling in _swapETHToToken()	11
3.2	Possible Sandwich/MEV For Reduced Returns	12
3.3	Flashloan-Based Oracle Price Manipulation	14
3.4	Accommodation of Non-ERC20-Compliant Tokens	16
3.5	Trust Issue of Admin Keys	18
4	**Conclusion**	**20**
	References	**21**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Velvet Capital` 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. This document outlines our audit results.

1.1 About Velvet Capital

`Velvet Capital` is a DeFi protocol that helps people and institutions create tokenized index funds, portfolios and other financial products with additional yield. The protocol provides all the necessary infrastructure for financial product development being integrated with AMMs, lending protocols and other DeFi primitives to give users a diverse asset management toolkit. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Stader` Protocol

Item	Description
Issuer	Velvet
Website	https://velvet.capital/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 23, 2022

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

- https://github.com/Velvet-Capital/protocols.git (be45896)

And here is the commit ID after all fixes for the issues found in the audit have been checked in. Note all the recommendations have been implemented and the issues have been resolved or mitigated.

- https://github.com/Velvet-Capital/protocols.git (0a6f765)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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.

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) [11], 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 design and implementation of the `Velvet Capital` 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings
Critical	1
High	0
Medium	3
Low	1
Informational	0
Total	5

We have so far identified a list of potential issues.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 critical-severity vulnerability, 3 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Velvet Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Corner Case Handling in _swapETHToToken()	Business Logic	Resolved
PVE-002	Medium	Possible Sandwich/MEV For Reduced Returns	Time and State	Resolved
PVE-003	Critical	Flashloan-Based Oracle Price Manipulation	Time and State	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improper Corner Case Handling in _swapETHToToken()

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

Description

The `Velvet Capital` protocol has an `Adapter` contract that is used for transferring funds from the vault to the contract and vice versa as well as swap tokens to and from native coins. Within the contract, there are a number of swap-related helper routines. Our analysis shows that a specific swap routine `_swapETHToToken()` needs to be improved to better handle possible corner cases.

To elaborate, we show below the implementation of this `_swapETHToToken()` routine. As the name indicates, this routine is used to swap ETH to a specific token. However, when the token being swapped is equal to `getETH()` (line 102), the resulting amount of `swapResult` is improperly calculated. Specifically, there is a missing assignment `swapResult = swapAmount` right before the `lendBNB()` call (line 104).

```
97      function _swapETHToToken(
98          address t,
99          uint256 swapAmount,
100         address to
101     ) public payable onlyIndexManager returns (uint256 swapResult) {
102         if (t == getETH()) {
103             if (tokenMetadata.vTokens(t) != address(0)) {
104                 lendBNB(t, tokenMetadata.vTokens(t), swapResult, to);
105             } else {
106                 IWETH(t).deposit{value: swapAmount}();
107                 swapResult = swapAmount;
108
109                 if (to != address(this)) {
```

```
                IWETH(t).transfer(to, swapAmount);
            }
        }
    } else {
        if (tokenMetadata.vTokens(t) != address(0)) {
            swapResult = pancakeSwapRouter.swapExactETHForTokens{
                value: swapAmount
            }(
                0,
                getPathForETH(t),
                address(this),
                block.timestamp // using 'now' for convenience, for mainnet pass deadline from frontend!
            )[1];
            lendToken(t, tokenMetadata.vTokens(t), swapResult, to);
        } else {
            swapResult = pancakeSwapRouter.swapExactETHForTokens{
                value: swapAmount
            }(
                0,
                getPathForETH(t),
                to,
                block.timestamp // using 'now' for convenience, for mainnet pass deadline from frontend!
            )[1];
        }
    }
}
```

Listing 3.1: `Adapter::_swapETHToToken()`

Recommendation Properly handle all possible cases in the above `_swapETHToToken()` routine.

Status This issue has been fixed in the following commit: `7e917cb`.

3.2 Possible Sandwich/MEV For Reduced Returns

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Adapter`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

As mentioned earlier, the `Velvet Capital` protocol has the constant need of swapping one asset to another. With that, the protocol has provided related helper routines to facilitate the asset conversion:

_swapETHToToken() and _swapTokenToETH(). Our analysis shows the current implementation does not have the necessary slippage control in place to mitigate possible risk.

```
    function _swapTokenToETH(
        address t,
        uint256 swapAmount,
        address to
    ) public onlyIndexManager returns (uint256 swapResult) {
        if (tokenMetadata.vTokens(t) != address(0)) {
            if (t == getETH()) {
                redeemBNB(tokenMetadata.vTokens(t), swapAmount, address(this));
                swapResult = address(this).balance;

                (bool success, ) = payable(to).call{value: swapResult}("");
                require(success, "Transfer failed.");
            } else {
                redeemToken(
                    tokenMetadata.vTokens(t),
                    t,
                    swapAmount,
                    address(this)
                );
                IERC20 token = IERC20(t);
                uint256 amount = token.balanceOf(address(this));
                require(amount > 0, "zero balance amount");

                TransferHelper.safeApprove(
                    t,
                    address(pancakeSwapRouter),
                    amount
                );
                swapResult = pancakeSwapRouter.swapExactTokensForETH(
                    amount,
                    0,
                    getPathForToken(t),
                    to,
                    block.timestamp
                )[1];
            }
        } else {
            TransferHelper.safeApprove(
                t,
                address(pancakeSwapRouter),
                swapAmount
            );
            if (t == getETH()) {
                IWETH(t).withdraw(swapAmount);
                (bool success, ) = payable(to).call{value: swapAmount}("");
                require(success, "Transfer failed.");
                swapResult = swapAmount;
            } else {
                swapResult = pancakeSwapRouter.swapExactTokensForETH(
```

```
                    swapAmount,
                    0,
                    getPathForToken(t),
                    to,
                    block.timestamp
                )[1];
            }
        }
    }
```

Listing 3.2: Adapter::_swapTokenToETH()

To elaborate, we show above one example helper routine `_swapTokenToETH()`. We notice the conversion is routed to `pancakeSwapRouter` in order to swap one asset to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

Recommendation Add necessary slippage control for token swaps.

Status This issue is being addressed in the following commit: `7e917cb`. Note the current `getSlippage()` may not provide the intended slippage control as the resulting `minAmount` is computed from the spot reserve, which could be influenced by a sandwich attack. Fortunately, the current protocol makes use of the `Chainlink` oracles to calculate the amount of index tokens to be minted, not the slippage when swapping.

3.3 Flashloan-Based Oracle Price Manipulation

- ID: PVE-003
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `PriceOracle`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [3]

Description

The `Velvet Capital` protocol has a `PriceOracle` contract to facilitate the token price discovery. Our analysis shows the current approach to compute the on-chain token price can be manipulated.

```
    function getTokenPrice(address token_address, address token1_address)
        external
        view
        override
        returns (uint256 price)
    {
        uint256 token_decimals = IERC20Metadata(token_address).decimals();
```

```
        uint256 min_amountIn = 1 * 10**token_decimals;
        if (token_address == token1_address) {
            price = min_amountIn;
        } else {
            (uint256 reserve0, uint256 reserve1) = getReserves(
                token_address,
                token1_address
            );
            price = uniswapV2Router.getAmountOut(
                min_amountIn,
                reserve0,
                reserve1
            );
        }
    }
```

Listing 3.3: `PriceOracle::getTokenPrice()`

To elaborate, we show above the related `getTokenPrice()` function. It comes to our attention that the conversion is routed to `UniswapV2`-based DEXs and the related spot reserves are used to compute the price! Therefore, they are vulnerable to possible front-running attacks, resulting in possible loss for the token conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of protocol users.

Status This issue has been fixed in the following commit: `7e917cb`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Adapter`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [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. 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 3.4: USDT Token **Contract**

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. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.5: `SafeERC20::safeApprove()`

In the following, we show the `lendBNB()` routine in the `Adapter` contract. If the `USDT` token is supported as `underlyingToken`, the unsafe version of `underlyingToken.approve(address(vToken), _amount)` (line 229) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)! Note the `lendToken()` routine in the same contract can be similarly improved.

```
220     function lendBNB(
221         address _underlyingAsset,
222         address _vAsset,
223         uint256 _amount,
224         address _to
225     ) internal {
226         IERC20 underlyingToken = IERC20(_underlyingAsset);
227         IVBNB vToken = IVBNB(_vAsset);
228 
229         underlyingToken.approve(address(vToken), _amount);
230         vToken.mint{value: _amount}();
231         uint256 vBalance = vToken.balanceOf(address(this));
232         TransferHelper.safeTransfer(_vAsset, _to, vBalance);
233     }
```

Listing 3.6: `Adapter::lendBNB()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. For the safe-version of `approve()`, there is a need to `safeApprove()` twice: the first one reduces the allowance to 0 and the second one sets the new allowance.

Status This issue has been fixed in the following commit: `7e917cb`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the `Velvet Capital` protocol, there is a privileged manager account (with the `DEFAULT_ADMIN_ROLE`) that plays a critical role in governing and regulating the system-wide operations (e.g., authorize other roles as well as configure various protocol risk parameters, etc.). Our analysis shows that the privileged account needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the privileged account.

```
    constructor() {
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);

        _setRoleAdmin(
            keccak256("ASSET_MANAGER_ROLE"),
            keccak256("DEFAULT_ADMIN_ROLE")
        );

        _setRoleAdmin(
            keccak256("INDEX_MANAGER_ROLE"),
            keccak256("DEFAULT_ADMIN_ROLE")
        );
    }

    modifier onlyAdmin(bytes32 role) {
        hasRole(getRoleAdmin(role), msg.sender);
        _;
    }

    function setupRole(bytes32 role, address account) public onlyAdmin(role) {
        _setupRole(role, account);
    }
```

Listing 3.7: Example Privileged Operations in `AccessController`

Specifically, the privileged functions in the `AccessController` contract allow for the authorization of various roles for different accounts. We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account may also be a counter-party risk to the contract 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.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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

This issue has been fixed in the following commit: `ba5b6b3`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Velvet Capital` protocol, which is a DeFi protocol that helps people and institutions create tokenized index funds, portfolios and other financial products with additional yield. The protocol provides all the necessary infrastructure for financial product development being integrated with AMMs, lending protocols and other DeFi primitives to give users a diverse asset management toolkit. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[10] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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