SMART CONTRACT AUDIT REPORT

for

RYSK

Prepared By: Xiaomi Huang

PeckShield

May 19, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 19, 2022	Jing Wang	Final Release
1.0-rc	May 3, 2022	Jing Wang	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

Given the opportunity to review the RYSK protocol design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given branch of RYSK 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 RYSK

RYSK is an options settlement protocol that is developed based on the Opyn Gamma protocol which allows anyone to buy, sell, and create arbitrary option tokens (oTokens) on any ERC20 asset. In comparison with the original Opyn Gamma protocol, RYSK makes a number of extensions or innovations to enable the creation of naked options where partial collateralization is used. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of RYSK

Item	Description
Name	RYSK
Website	https://www.rysk.finance/
Type	Whitelist Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 19, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that RYSK protocol assumes a reasonable financial module for the modified partial collateralization system and the financial module itself is not part of this audit.

- https://github.com/rysk-finance/GammaProtocol.git (e404d5e)

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

- https://github.com/rysk-finance/GammaProtocol.git (90fe3b0)

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			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	*Medium*	*Low*
		Likelihood	

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.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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 `RYSK` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	🟦
Low	4	🟦🟦🟦
Informational	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 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 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key RYSK Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Sanity Checks Of whitelistProduct()	Coding Practices	Confirmed
PVE-002	Low	Suggested Uses of SafeMath	Numeric Errors	Fixed
PVE-003	Informational	Redundant State/Code Removal	Coding Practices	Confirmed
PVE-004	Low	Improved Sanity Checks for whitelistNakedCollateral()/whitelistCoveredCollateral()	Coding Practices	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-006	Low	Incompatibility With Deflationary Tokens in transferToPool()	Business Logic	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 Improved Sanity Checks Of whitelistProduct()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Whitelist`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

The `RYSK` protocol implements a `Whitelist` module which manages all restrictions set by the system admin. These restrictions validate whether a unique combination of `strike - asset - collateral - put/call` type is valid when the corresponding product is created and registered in the system.

During our analysis of `Whitelist::whitelistProduct()`, we notice the limits of creation for whitelisted products with non-underlying collateral for calls was removed to allow `USD` collateral to be used for call assets (and the same for puts). However, the complete removal of this requirement not only allows `USD` as collateral, but also allows any kinds of assets to be collateral. This might break the assumption when computing the margin for partial collateralization.

```
    function whitelistProduct(
        address _underlying,
        address _strike,
        address _collateral,
        bool _isPut
    ) external onlyOwner {
        require(whitelistedCollateral[_collateral], "Whitelist: Collateral is not
            whitelisted");
        require(
            (_isPut && (_strike == _collateral))  (!_isPut && (_collateral ==
                _underlying)),
            "Whitelist: Only allow fully collateralized products"
        );

        bytes32 productHash = keccak256(abi.encode(_underlying, _strike, _collateral,
            _isPut));
        whitelistedProduct[productHash] = true;

        emit ProductWhitelisted(productHash, _underlying, _strike, _collateral, _isPut);
    }
```

Listing 3.1: Gamma Whitelist::whitelistProduct()

```
    function whitelistProduct(
        address _underlying,
        address _strike,
        address _collateral,
        bool _isPut
    ) external onlyOwner {
        require(whitelistedCollateral[_collateral], "Whitelist: Collateral is not
            whitelisted");

        bytes32 productHash = keccak256(abi.encode(_underlying, _strike, _collateral,
            _isPut));

        whitelistedProduct[productHash] = true;

        emit ProductWhitelisted(productHash, _underlying, _strike, _collateral, _isPut);
    }
```

Listing 3.2: RYSK Whitelist:: whitelistProduct())

When there is a need to whitelist a product, the sanity checks of the collateral to either be the underlying asset or strike asset might be maintained to minimize the risk for the margin calculation system.

Recommendation Improve the sanity checks in the related `whitelistProduct()` function.

```
    function whitelistProduct(
        address _underlying,
        address _strike,
        address _collateral,
        bool _isPut
    ) external onlyOwner {
        require(whitelistedCollateral[_collateral], "Whitelist: Collateral is not
            whitelisted");
        require(
            ((_strike == _collateral)  (_collateral == _underlying),
            "Whitelist: Only allow fully collateralized products"
        );

        bytes32 productHash = keccak256(abi.encode(_underlying, _strike, _collateral,
            _isPut));

        whitelistedProduct[productHash] = true;

        emit ProductWhitelisted(productHash, _underlying, _strike, _collateral, _isPut);
    }
```

Listing 3.3: `Whitelist::whitelistProduct())(revised)`

Status The issue has been confirmed.

3.2 Suggested Uses of SafeMath

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Controller`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [3]

Description

`SafeMath` is a Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, we find that it is not used in all the related cases in the `Controller` contract.

In particular, while examining the logic of the `_liquidate()` routine, we notice in the arithmetic operations (lines 1011-1012) do not use the `SafeMath` library to prevent overflows, which may introduce unexpected behavior. We suggest to use `SafeMath` to avoid unexpected overflows.

```
1007    function _liquidate(Actions.LiquidateArgs memory _args) internal notPartiallyPaused
            {
1008        require(_checkVaultId(_args.owner, _args.vaultId), "C35");
1009        ...
1010        if (vaultLiqDetails.series == vault.shortOtokens[0]) {
1011            vaultLiqDetails.shortAmount += uint128(_args.amount);
1012            vaultLiqDetails.collateralAmount += uint128(collateralToSell);
1013        } else {
1014            vaultLiqDetails.series = vault.shortOtokens[0];
1015            vaultLiqDetails.shortAmount = uint128(_args.amount);
1016            vaultLiqDetails.collateralAmount = uint128(collateralToSell);
1017        }
1018        ...
1019    }
```

Listing 3.4: `Controller::_liquidate()`

Recommendation Use `SafeMath` to avoid unexpected overflows.

Status This issue has been fixed in this commit: `90fe3b0`.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Actions`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [5]

Description

The liquidation system in the `Opyn Gamma` protocol involves a reverse Dutch auction mechanism. As the `RYSK` protocol makes further customizations to enable `USD` collateralized calls and `ETH` collateralized puts, so there is a need for more liquidators to the system to handle vaults which may go under-water. To adapt it, the reverse Dutch auction mechanism are removed from the `RYSK` protocol and thus the `roundId` was also removed as the vault liquidates using the latest round spot price instead of one in the past.

However, we observe the `roundId` is still parsed in the `_parseLiquidateArgs()` routine, which is not used and can be safely removed.

```
    function _parseLiquidateArgs(ActionArgs memory _args) internal pure returns (
        LiquidateArgs memory) {
        require(_args.actionType == ActionType.Liquidate, "A18");
        require(_args.owner != address(0), "A19");
        require(_args.secondAddress != address(0), "A20");
        require(_args.data.length == 32, "A21");

        // decode chainlink round id from _args.data
        uint256 roundId = abi.decode(_args.data, (uint256));

        return
            LiquidateArgs({
                owner: _args.owner,
                receiver: _args.secondAddress,
                vaultId: _args.vaultId,
                amount: _args.amount,
                roundId: roundId
            });
    }
```

Listing 3.5: `Actions::_parseLiquidateArgs()`

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been confirmed.

3.4 Improved Sanity Checks for whitelistNakedCollateral()/whitelistCoveredCollateral()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Whitelist
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [2]

Description

As mentioned in Section 3.1, the `Whitelist` module manages all restrictions set by the system admin. As the `RYSK` protocol adds naked collateral option which only allows vaults to be collateralized with assets that cannot fully be collateralized with type-0 vaults, so the `coveredWhitelistedCollateral` and `nakedWhitelistedCollateral` maps are introduced to handle cases to prevent users from creating type-0 vaults with incorrect collateral. Specifically, collateral that is suitable for type-0 vaults is labeled as `isCoveredWhitelistedCollateral` and collateral that is suitable for type-1 vaults is labeled as `isNakedWhitelistedCollateral`.

During our analysis, we notice there is no checks to make sure the admin is adding the correct product in the target category, e.g., naked collateral is only allowed to be added into the `isNakedWhitelistedCollateral`. If the admin happens to add a naked collateral product into the `coveredWhitelistedCollateral` category, unexpected results may be encountered in the `_isMarginableCollateral()` routine, including the broken `_verifyFinalState()` routine in the controller. This specific routine is used in a number of scenarios, including when vaults are created, collateral is deposited or withdrawn, and/or when shorts/longs are created.

```
    function whitelistCoveredCollateral(
        address _collateral,
        address _underlying,
        bool _isPut
    ) external onlyOwner {
        bytes32 productHash = keccak256(abi.encode(_collateral, _underlying, _isPut));
        coveredWhitelistedCollateral[productHash] = true;
        emit CoveredCollateralWhitelisted(_collateral, _underlying, _isPut);
    }

    function whitelistNakedCollateral(
        address _collateral,
        address _underlying,
        bool _isPut
    ) external onlyOwner {
        bytes32 productHash = keccak256(abi.encode(_collateral, _underlying, _isPut));
        nakedWhitelistedCollateral[productHash] = true;
        emit NakedCollateralWhitelisted(_collateral, _underlying, _isPut);
    }
```

Listing 3.6: `Whitelist::whitelistCoveredCollateral()and whitelistNakedCollateral()`

Recommendation Apply necessary sanity checks on the `whitelistCoveredCollateral()` and `whitelistNakedCollateral()` routines.

```
    function whitelistCoveredCollateral(
        address _collateral,
        address _underlying,
        bool _isPut
    ) external onlyOwner {
        require(
            (_isPut && (_strike == _collateral))  (!_isPut && (_collateral ==
                _underlying)),
            "Whitelist: Only allow fully collateralized products"
        );
        bytes32 productHash = keccak256(abi.encode(_collateral, _underlying, _isPut));
        coveredWhitelistedCollateral[productHash] = true;
        emit CoveredCollateralWhitelisted(_collateral, _underlying, _isPut);
    }

    function whitelistNakedCollateral(
        address _collateral,
        address _underlying,
        bool _isPut
    ) external onlyOwner {
        require(
            (_isPut && (_strike != _collateral))  (!_isPut && (_collateral !=
                _underlying)),
            "Whitelist: Only allow naked products"
        );
        bytes32 productHash = keccak256(abi.encode(_collateral, _underlying, _isPut));
        nakedWhitelistedCollateral[productHash] = true;
        emit NakedCollateralWhitelisted(_collateral, _underlying, _isPut);
    }
```

Listing 3.7: `Whitelist::whitelistCoveredCollateral()and whitelistNakedCollateral()(revised)`

Status The issue has been confirmed.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [7]
- CWE subcategory: CWE-287 [4]

Description

In the RYSK protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., setting various system parameters). 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 owner account and its related privileged accesses in current contract.

To elaborate, we show the setController() routine to set the _controller, who has the ability to withdrawn all funds from the MarginPool.

```
    function setController(address _controller) external onlyOwner {
        updateImpl(CONTROLLER, _controller);
    }
```

Listing 3.8: AddressBook::setController()

```
    function transferToUser(
        address _asset,
        address _user,
        uint256 _amount
    ) public onlyController {
        require(_user != address(this), "MarginPool: cannot transfer assets to oneself")
            ;
        assetBalance[_asset] = assetBalance[_asset].sub(_amount);

        // transfer _asset _amount from pool to _user
        ERC20Interface(_asset).safeTransfer(_user, _amount);
        emit TransferToUser(_asset, _user, _amount);
    }
```

Listing 3.9: MarginPool::transferToUser()

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

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed.

3.6 Incompatibility With Deflationary Tokens in MarginPool::transferToPool()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MarginPool
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The RYSK protocol enables any user to create option tokens, which represent the right to buy or sell a certain asset in a predefined price (strike price) at expiry. At the core of the protocol is the MarginPool contract that moves and stores all the ERC20 tokens. Users only need to approve an asset to be used by MarginPool once. After that, the asset can be used to create multiple different options. In the following, we show the relatedtransferToPool() function that transfers an asset from a user to the pool and updates the assetBalance.

```
    function transferToPool(
        address _asset,
        address _user,
        uint256 _amount
    ) public onlyController {
        require(_amount > 0, "MarginPool: transferToPool amount is equal to 0");
        assetBalance[_asset] = assetBalance[_asset].add(_amount);

        // transfer _asset _amount from _user to pool
        ERC20Interface(_asset).safeTransferFrom(_user, address(this), _amount);
        emit TransferToPool(_asset, _user, _amount);
    }
```

Listing 3.10: MarginPool.sol

However, in the cases of deflationary tokens, as shown in the above code snippets, the input _amount may not be equal to the received amount due to the charged (and burned) transaction fee. In other words, when a deflationary token is used, the above operations may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

Recommendation Make use of the `balanceOf()` routine to update the `assetBalance` after the token transfer.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `RYSK` design and implementation. The system is a capital efficient option protocol that enables sellers to create spreads and other combinations, trade atomically, flash loan mint `otokens`, assign operators to roll over vaults, create perpetual instruments, and more. During the audit, we notice that the current code base is well structured and neatly organized, and those identified issues are promptly confirmed and fixed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

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

[10] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.