SMART CONTRACT AUDIT REPORT

for

Gyro Protocol

Prepared By: Yiqun Chen

PeckShield

November 27, 2021

Document Properties

Client	Gyro
Title	Smart Contract Audit Report
Target	Gyro
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Stephen Bie, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 27, 2021	Xuxian Jiang	Final Release
1.0-rc	November 23, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4

1.1 About Gyro 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 6

2 Findings 10

2.1 Summary 10

2.2 Key Findings 11

3 Detailed Results 12

3.1 Proper Logic Of Distributor::nextRewardFor() 12

3.2 Fork-Resistant Domain Separator in Gyro And sGyro 13

3.3 Potentially Unwanted Reverts in BondCalculator::getKValue() 14

3.4 Trust Issue of Admin Keys 15

3.5 Meaningful Events For Important States Change 17

3.6 Potential Rebasing Perturbation 18

4 Conclusion 20

References 21

1 Introduction

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

Gyro is an algorithmic currency protocol that is forked from the OlympusDAO protocol. In essence, it introduces unique economic and game-theoretic dynamics into the market through asset-backing and protocol owned value. It is a value-backed, self-stabilizing, and decentralized stablecoin with unique collateral backing and algorithmic incentive mechanism. Different from other popular stablecoin solutions, it is proposed to create a stable asset through the management of treasury of assets without dependence on fiat currencies.

The basic information of Gyro is as follows:

Table 1.1: Basic Information of Gyro

Item	Description
Name	Gyro
Website	https://gyro.money/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 27, 2021

In the following, we show the list of reviewed contracts that are currently deployed on BSC.

- https://www.bscscan.com/address/0x8B1522402FECe066d83E0F6C97024248Be3C8c01 (Reservoir)

- https://www.bscscan.com/address/0x1B239ABe619e74232c827FBE5E49a4C072bD869D (Gyro)
- https://www.bscscan.com/address/0xdc93eb0eb1bf2ac6da14b3ee54a8d7fbb15bb058 (Staked Gyro)
- https://www.bscscan.com/address/0xd6b1997149f1114f6251B6df1D907770Ba6df819 (Bond)
- https://www.bscscan.com/address/0xE9C178CFDFEb917A46429714E5D51f6d4f296b75 (Vault)

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 the 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 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 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. 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

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

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.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Gyro` 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	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 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 Gyro Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Logic Of Distributor::nextRewardFor()	Business Logic	Confirmed
PVE-002	Low	Fork-Resistant Domain Separator in Gyro And sGyro	Business Logic	Confirmed
PVE-003	Low	Potentially Unwanted Reverts in BondCalculator::getKValue()	Coding Practice	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Meaningful Events For Important States Change	Business Logic	Confirmed
PVE-006	Informational	Potential Rebasing Perturbation	Time and State	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 Proper Logic Of Distributor::nextRewardFor()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Distributor
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The Gyro protocol has a Distributor contract that allows to set up distributions (in terms of the reward rate) to certain recipients. This is helpful in the rebase context to invoke necessary distributions. Our analysis on this contract shows one key function needs to be revised.

To elaborate, we show below the related nextRewardFor() function. As the name indicates, this function is designed to compute next reward for the given recipient address. It comes to our attention that the current function always return the last hit on the given recipient address. This works fine if we assume there is no duplicate in the configured recipient list. In the case of multiple occurrences of the same recipient, there is a need to accumulate all rewards together.

```
    /**
        @notice view function for next reward for specified address
        @param recipient_ address
        @return uint
     */
    function nextRewardFor(address recipient_) public view returns (uint256) {
        uint256 reward = 0;
        for (uint256 i = 0; i < info.length; i++) {
            if (info[i].recipient == recipient_) {
                reward = nextRewardAt(info[i].rate);
            }
        }
        return reward;
    }
```

Listing 3.1: Distributor::nextRewardFor()

Recommendation Accommodate the case of multiple occurrences of the same recipient in the `Distributor` contract.

Status This issue has been confirmed.

3.2 Fork-Resistant Domain Separator in Gyro And sGyro

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Gyro`, `sGyro`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The two key tokens in `Gyro` are designed to strictly follows the widely-accepted ERC20 specification. In the meantime, we notice the support of EIP-2612 with the `permit()` function that allows for approvals to be made via `secp256k1` signatures. Interestingly, we notice the state variable `DOMAIN_SEPARATOR` in `ERC20Permit` is initialized once inside the `initialize()` function (lines 48-56).

```
41     constructor() {
42         uint256 chainID;
43         // solhint-disable-next-line no-inline-assembly
44         assembly {
45             chainID := chainid()
46         }
47
48         DOMAIN_SEPARATOR = keccak256(
49             abi.encode(
50                 keccak256("EIP712Domain(string name,string version,uint256 chainId,
                       address verifyingContract)"),
51                 keccak256(bytes(name())),
52                 keccak256(bytes("1")), // Version
53                 chainID,
54                 address(this)
55             )
56         );
57     }
```

Listing 3.2: `ERC20Permit::initialize()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing crosschain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed DOMAIN_SEPARATOR, a valid signature for one chain could be replayed on the other.

```
    function permit(
        address owner,
        address spender,
        uint256 amount,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual override {
        require(block.timestamp <= deadline, "Permit: expired deadline");

        bytes32 hashStruct =
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, amount, _nonces[owner
                ].current(), deadline));

        bytes32 _hash = keccak256(abi.encodePacked(uint16(0x1901), DOMAIN_SEPARATOR,
            hashStruct));

        address signer = ecrecover(_hash, v, r, s);
        require(signer != address(0) && signer == owner, "ERC20Permit: Invalid signature
            ");

        _nonces[owner].increment();
        _approve(owner, spender, amount);
    }
```

Listing 3.3: ERC20Permit::permit()

Recommendation Recalculate the value of DOMAIN_SEPARATOR inside the permit() function.

Status This issue has been confirmed.

3.3 Potentially Unwanted Reverts in BondCalculator::getKValue()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BondCalculator
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [1]

Description

The Gyro protocol has the constant need to compute the current debt ratio as well as the bond price. The computation is assisted with the BondCalculator contract. While reviewing this BondCalculator contract, we notice one key function to compute the constant product can be improved to avoid unwanted friction.

To elaborate, we show below the `getKValue()` function. The issue stems from the need of accommodating different decimals of involved tokens. In particular, the internal variable `decimals` is calculated as `token0.add(token1).sub(IERC20(pair_).decimals())` (line 18). This calculation may be potentially reverted if the following condition is met, i.e., `token0+token1 < 18`. To avoid this, we can explicitly detect whether it will lead to underflow and adjust the constant product result accordingly.

```
15     function getKValue(address pair_) public view returns (uint256 k_) {
16         uint256 token0 = IERC20(IUniswapV2Pair(pair_).token0()).decimals();
17         uint256 token1 = IERC20(IUniswapV2Pair(pair_).token1()).decimals();
18         uint256 decimals = token0.add(token1).sub(IERC20(pair_).decimals());
19 
20         (uint256 reserve0, uint256 reserve1, ) = IUniswapV2Pair(pair_).getReserves();
21         k_ = reserve0.mul(reserve1).div(10**decimals);
22     }
```

Listing 3.4: `BondCalculator::getKValue()`

Recommendation Accommodate all possible situations in the calculation of the constant product in `getKValue()`.

Status This issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Gyro` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and contract 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 the related privileged accesses in current contracts.

```
    function initializeBondTerms(
        uint256 controlVariable_,
        uint256 period_,
        uint256 minPrice_,
```

```
        uint256 maxPayout_,
        uint256 fee_,
        uint256 maxDebt_,
        uint256 initialDebt_
    ) external onlyOwner() {
        require(terms.controlVariable == 0, "Bonds must be initialized from 0");
        terms = Terms({
            controlVariable: controlVariable_,
            period: period_,
            minPrice: minPrice_,
            maxPayout: maxPayout_,
            fee: fee_,
            maxDebt: maxDebt_
        });
        totalDebt = initialDebt_;
        lastDecay = block.number;
    }

    function setBondTerms(PARAMETER parameter_, uint256 input_) external onlyOwner() {
        if (parameter_ == PARAMETER.VESTING) {
            // 0
            require(input_ >= 40000, "Vesting must be longer than 36 hours"); // assuming, 3s block time
            terms.period = input_;
        } else if (parameter_ == PARAMETER.PAYOUT) {
            // 1
            require(input_ <= 1000, "Payout cannot be above 1 percent");
            terms.maxPayout = input_;
        } else if (parameter_ == PARAMETER.FEE) {
            // 2
            require(input_ <= 10000, "Treasury fee cannot exceed payout");
            terms.fee = input_;
        } else if (parameter_ == PARAMETER.DEBT) {
            // 3
            terms.maxDebt = input_;
        }
    }
```

Listing 3.5: Example Setters in GyroBond

It is worrisome if the privileged `owner` account is a plain EOA account. The on-chain analysis shows that the current deployer, i.e., `0xcb3ab04692c95c2124e05b8fe381b67fe7a30518`, is currently configured as the the `owner` for multiple protocol-wide contracts. Note that a multi-sig account can greatly alleviate this concern, though it is still not 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 This issue has been confirmed with the team. The team clarifies the plan to transfer the control a DAO governance as planned in https://vote.gyro.money.

3.5 Meaningful Events For Important States Change

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `GyroVault` contract as an example. This contract is designed to be the staking contract in `Gyro`. While examining the events that reflect the changes of protocol-wide contracts, we notice there is a lack of emitting important events that reflect their changes. Specifically, when the `escrowContract` is being updated, there is no respective event being emitted to reflect its change.

```
    function setContract(CONTRACTS contract_, address address_) external onlyOwner() {
        if (contract_ == CONTRACTS.DISTRIBUTOR) {
            // 0
            distributor = address_;
        } else if (contract_ == CONTRACTS.ESCROW) {
            // 1
            require(escrowContract == address(0), "Escrow cannot be set more than once");
            escrowContract = address_;
        } else if (contract_ == CONTRACTS.LOCKER) {
            // 2
            require(locker == address(0), "Locker cannot be set more than once");
            locker = address_;
        }
```

```
    }
```

Listing 3.6: `GyroVault::setContract()`

The same issue is also applicable to other functions, e.g., `setEscrowPeriod()`.

Recommendation Properly emit respective events to help off-chain monitoring and accounting tools.

Status This issue has been confirmed.

3.6 Potential Rebasing Perturbation

- ID: PVE-006
- Severity: Informational
- Likelihood: -
- Impact: -
- Target: `GyroVault`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

Description

As mentioned earlier, the `Gyro` protocol implements a unique expansion and contraction mechanism in order to be a stablecoin. In the following, we examine the rebasing mechanism implemented in the protocol.

To elaborate, we show below the `GyroVault::rebase()` routine that triggers `sGyro`-rebasing so that the accumulated profits can be evenly distributed to circulating `sGyro`. Note that the rebasing operation will not be triggered until the current block height reaches the specified `epoch.nextBlock` number.

```
    function rebase() public {
        if (epoch.nextBlock <= block.number) {
            uint256 prevEpoch = epoch.number;
            uint256 prevRewards = epoch.distribute;

            epoch.nextBlock = epoch.nextBlock.add(epoch.period);
            epoch.number++;

            uint256 balance = contractBalance();
            uint256 staked = ISGyro(sGyro).circulatingSupply();

            if (balance <= staked) {
                epoch.distribute = 0;
            } else {
                epoch.distribute = balance.sub(staked);
            }
```

```
            ISGyro(sGyro).rebase(prevRewards, prevEpoch);

            if (distributor != address(0)) {
                IDistributor(distributor).distribute();
            }

            emit LogRebase(epoch.number, epoch.nextBlock, epoch.distribute);
        }
    }
```

Listing 3.7: GyroVault::rebase()

With that, it is possible that right before `epoch.nextBlock` is reached, a user may choose to stake (or unstake) to increase (decrease) the circulating supply of `sGyro`. Either way, the current rebasing operation as well as the `epoch.distribute` amount may be influenced.

Note that this is a common sandwich-based arbitrage behavior 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. 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 to the above sandwich arbitrage behavior to better protect the rebasing operation in `Gyro`.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the Gyro design and implementation. Based on OlympusDAO, the Gyro protocol introduces unique economic and game-theoretic dynamics into the market through asset-backing and protocol owned value. The current code base is well structured and neatly organized. 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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: Concurrency. https://cwe.mitre.org/data/definitions/557.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.