SMART CONTRACT AUDIT REPORT

for

Dyson Protocol

Prepared By: Xiaomi Huang

PeckShield

February 24, 2023

Document Properties

Client	Sphere Finance
Title	Smart Contract Audit Report
Target	Dyson Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 24, 2023	Luck Hu	Final Release
1.0-rc	December 19, 2022	Luck Hu	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 design document and related smart contract source code of the `Dyson` 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Dyson Protocol

`Dyson` is a multi-chain yield maximizer and optimizer which brings an easily-accessible suite of simple and exotic yield-farming strategies to `DeFi`. The main goal of `Dyson` is to simplify yield opportunities that are otherwise time-consuming, gas-intensive, and complicated, optimizing yield and making these opportunities available to everyone. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Dyson Protocol

Item	Description
Name	Sphere Finance
Website	https://www.sphere.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 24, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audit only covers `DysonMaximizerBalancerVault.sol`, `MaximizerBalancer.sol`, `StrategyBalancerAC.sol`, and `DysonBalancerVault.sol`.

- https://github.com/DysonFarm/dyson-contracts/tree/neo (205d5230)

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

- https://github.com/DysonFarm/dyson-contracts/tree/neo (78c04d43)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

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

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

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

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy

Table 1.3: The Full Audit Checklist

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

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

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

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

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

1.4 Disclaimer

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

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Dyson` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	2	■■
Informational	0	
Total	4	

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

2.2 Key Findings

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

Table 2.1: Key Dyson Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Lack of Slippage Control in _zapNativeToSecondaryWant()	Time and State	Confirmed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-003	Low	Possible Costly Vault Token from Improper Initialization	Time and State	Mitigated
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Lack of Slippage Control in _zapNativeToSecondaryWant()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

In the Dyson protocol, the MaximizerBalancer contract acts as the Maximizer strategy which keeps harvesting the primary strategy, converts all the earned Want tokens into the secondaryWant tokens, and stakes these secondaryWant tokens to the secondary vault to earn rewards. While reviewing the logic to convert the Want tokens to the secondaryWant tokens, we notice there is no slippage control for the token transformation.

To elaborate, we show below the code snippet of the MaximizerBalancer::_zapNativeToSecondaryWant () routine. After the earned Want tokens are withdrawn from the primary vault, they are converted to the native tokens. Next, the _zapNativeToSecondaryWant() routine is used to zap the native tokens to the secondaryWant tokens. At the beginning of the routine, half of the native tokens are converted to the token0 of the secondaryWant. However, we notice the minimum output amount, i.e., amountOutMin, is set to 0 (line 311), which means there is no slippage control in place for the swap. Similarly, there is no slippage control either for the swap from the other half of the native tokens to the token1 of the secondaryWant (line 316).

What is more, when the received token0/token1 are supplied to the secondaryRouter to add new liquidity, the minimum amounts for token0/token1 are both set to 0 (lines 327 – 328), which means there is no slippage control for the new liquidity adding.

```
305    function _zapNativeToSecondaryWant(uint256 nativeBalance) internal {
306        require(IERC20Upgradeable(native).balanceOf(address(this)) >= nativeBalance, "
               zap::doesn't have enough native balance");
```

```
        uint256 nativeHalf = nativeBalance / 2;

        if (native != secondaryLpToken0) {
            IDystopiaRouter.Route[] memory routeArray = getRoute(routerUtils.getNativeToSecondaryLpToken0Route(), routerUtils.getIsStableNativeToSecondaryLpToken0());
            IDystopiaRouter(routerUtils.secondaryRouter()).swapExactTokensForTokens(nativeHalf, 0, routeArray, address(this), block.timestamp);
        }

        if (native != secondaryLpToken1) {
            IDystopiaRouter.Route[] memory routeArray = getRoute(routerUtils.getNativeToSecondaryLpToken1Route(), routerUtils.getIsStableNativeToSecondaryLpToken1());
            IDystopiaRouter(routerUtils.secondaryRouter()).swapExactTokensForTokens(nativeHalf, 0, routeArray, address(this), block.timestamp);
        }

        uint256 secondaryLpToken0Bal = IERC20Upgradeable(secondaryLpToken0).balanceOf(address(this));
        uint256 secondaryLpToken1Bal = IERC20Upgradeable(secondaryLpToken1).balanceOf(address(this));
        IDystopiaRouter(routerUtils.secondaryRouter()).addLiquidity(
            secondaryLpToken0,
            secondaryLpToken1,
            routerUtils.getIsStableSecondaryLp0LP1(),
            secondaryLpToken0Bal,
            secondaryLpToken1Bal,
            0,
            0,
            address(this),
            block.timestamp
        );
    }
```

Listing 3.1: MaximizerBalancer::_zapNativeToSecondaryWant()

The lack of proper slippage control opens up the possibility for front-running and potentially results in a smaller converted amount. 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. 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 `Dystopia`. 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.

Note the same issue is also applicable to the `MaximizerBalancer::_zapPrimaryWantToNative()/StrategyBalancerAC::swap()/addLiquidity()` routines.

Recommendation Develop an effective mitigation to the above sandwich arbitrage to better protect the interests of users.

Status This issue has been confirmed and the team clarified that: `Dyson` will introduce off-chain slippage control via `Gelato` and harvests via off-chain calculated minimum thresholds to ensure front-running is mitigated.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MaximizerBalancer`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
```

```
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `_harvestRewardAndSecondaryLP()` routine in the `MaximizerBalancer` contract. If the `ZRX` token is supported as `reward1`, the unsafe version of `reward1.transfer(_sender, pendingReward))` (line 271) may return `false` while not revert. Without a validation on the return value, the transaction can proceed even when the transfer fails.

Note it shares the same issue at lines 279/290 in the same routine.

```
259     function _harvestRewardAndSecondaryLP(address _sender) internal {
260         UserInfo storage user = userInfo[_sender];
261         uint256 userBalance = maximizerVault.balanceBelongTo(_sender);
262 
263         uint256 pendingReward;
264         uint256 masterBalance;
265         if (userBalance > 0) {
266             // give reward1 to user
267             pendingReward = userBalance * accReward1PerShare / 1e18 - user.reward1Debt;
268             masterBalance = reward1.balanceOf(address(this));
269             if (pendingReward > masterBalance) pendingReward = masterBalance;
270             if (pendingReward > 0) {
271                 reward1.transfer(_sender, pendingReward);
272             }
273 
274             // give reward2 to user
275             pendingReward = userBalance * accReward2PerShare / 1e18 - user.reward2Debt;
276             masterBalance = reward2.balanceOf(address(this));
277             if (pendingReward > masterBalance) pendingReward = masterBalance;
278             if (pendingReward > 0) {
279                 reward2.transfer(_sender, pendingReward);
280             }
281 
282             // give secondaryWant to user
283             uint256 pendingSecondaryWant = userBalance * accSecondaryWantPerShare / 1e18
                    - user.secondaryWantDebt;
284             if (pendingSecondaryWant > 0) {
285                 IPenroseMasterChef(penroseChef).unstakeLpAndWithdraw(dystPoolAddress,
                        pendingSecondaryWant);
```

```
                uint256 masterBalanceSecondaryWant = secondaryWant.balanceOf(address(
                    this));
                if (pendingSecondaryWant > masterBalanceSecondaryWant)
                    pendingSecondaryWant = masterBalanceSecondaryWant;
            }
            if (pendingSecondaryWant > 0) {
                secondaryWant.transfer(_sender, pendingSecondaryWant);
            }
            ...
        }
    }
```

Listing 3.3: `MaximizerBalancer::_harvestRewardAndSecondaryLP()`

Recommendation Accommodate the above-mentioned idiosyncrasies with safe-version implementation of ERC20-related `transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `16258f2`.

3.3 Possible Costly Vault Token from Improper Initialization

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `DysonMaximizerBalancerVault`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

The `DysonMaximizerBalancerVault` contact is the `Maximizer` vault where the user can deposit the `Want` token for yield optimizing. The `DysonMaximizerBalancerVault` contact is then in charge of sending the `Want` token into the `Maximizer` strategy. The user will get the pro-rata share based on the deposit amount. While examining the share calculation with the given deposit, we notice an issue that may unnecessarily make the vault token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine, which is used for the user to deposit the `Want` tokens and get respective vault tokens in return. The issue occurs when the vault is being initialized under the assumption that the current vault is empty.

```
    function deposit(uint _amount) public nonReentrant {
        uint256 _pool = balance();
        want().safeTransferFrom(msg.sender, address(this), _amount);

        uint256 shares = 0;
```

```
108         if (totalSupply() == 0) {
109             shares = _amount;
110         } else {
111             shares = _amount * totalSupply() / _pool;
112         }
113         earn(shares);
114         _mint(msg.sender, shares);
115     }
```

Listing 3.4: DysonMaximizerBalancerVault::deposit()

Specifically, when the vault is being initialized, the share value directly takes the value of `_amount` (line 109), which is under control by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = _amount = 1WEI`. With that, the actor can further deposit a huge amount of the `Want` token with the goal of making the vault token extremely expensive.

An extremely expensive vault token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the vault without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial stake provider, but this cost is expected to be low and acceptable. Another alternative requires a guarded launch to ensure the pool is always initialized properly.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the share amount when the vault is being initialized.

Status This issue has been confirmed and the team will exercise extra caution in having a guarded launch to ensure the pool will be properly initialized.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `DysonMaximizerBalancerVault`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the Dyson protocol, there is a privileged accounts, e.g., owner, that plays a critical role in governing and regulating the system-wide operations (e.g., set the strategy for the Maximizer vault). Our analysis shows that the privileged account needs to be scrutinized. In the following, we use the DysonMaximizerBalancerVault contract as an example and show the representative functions potentially affected by the privileges of the owner account.

Specifically, the privileged functions in DysonMaximizerBalancerVault allow for the owner to set the strategy for the Maximizer vault, rescue stuck funds, set the boost pool, etc.

```
    function setStrategy(IMaximizerBalancer _strategy) public onlyOwner {
        require(!_isStrategyInitialized, 'strategy already initialized');
        strategy = _strategy;
    }

    /**
     * @dev Rescues random funds stuck that the strat can't handle.
     * @param _token address of the token to rescue.
     */
    function inCaseTokensGetStuck(address _token) external onlyOwner {
        require(_token != address(want()), "!token");

        uint256 amount = IERC20Upgradeable(_token).balanceOf(address(this));
        IERC20Upgradeable(_token).safeTransfer(msg.sender, amount);
    }

    function setBoostPool(address _address) public onlyOwner {
        boostPool = IBoostPoolBalancer(_address);
    }
```

Listing 3.5: Example Privileged Operations in the DysonMaximizerBalancerVault Contract

We understand the need of the privileged functions for protocol maintenance, but at the same time the extra power to the privileged account may also be a counter-party risk to the protocol users. It is worrisome if the privileged account is 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 This issue has been mitigated as the team confirmed they will use multi-sig for the owner account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Dyson` protocol. `Dyson` is a multichain yield maximizer and optimizer which brings an easily-accessible suite of simple and exotic yield-farming strategies to `DeFi`. The main goal of `Dyson` is to simplify yield opportunities that are otherwise time-consuming, gas-intensive, and complicated, optimizing yield and making these opportunities available to everyone. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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.