SMART CONTRACT AUDIT REPORT

for

Rango V2

Prepared By: Xiaomi Huang

PeckShield

June 26, 2023

Document Properties

Client	Rango
Title	Smart Contract Audit Report
Target	Rango V2
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 26, 2023	Stephen Bie	Final Release
1.0-rc	April 26, 2023	Stephen Bie	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 Rango V2, 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 Rango V2

Rango V2 implements a cross-chain decentralized exchange (DEX), which provides the cross-chain swap service with a one-transaction user experience. It also implements the multi-bridge aggregation, including Multichain, PolyNetwork, Synapse, cBridge, Axelar, and etc. It provides the user with unprecedented performance and flexibility. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Rango V2

Item	Description
Target	Rango V2
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 26, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the contracts/facets/thorchain sub-directory is out of the audit scope. Additionally, we assume all the aggregated bridge protocols are trusted and the bridge protocols themselves are not part of the audit.

- https://github.com/rango-exchange/rango-contracts-v2.git (0f81e2b)

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

- https://github.com/rango-exchange/rango-contracts-v2.git (44d24f6)

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 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 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) [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.

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 `Rango V2` 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	2	■ ■
Low	0	
Informational	2	■ ■
Undetermined	1	■
Total	5	

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 2 medium-severity vulnerabilities, 2 informational recommendations, and 1 undetermined issue.

Table 2.1: Key Rango V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Untrusted User Input in LibSwapper::onChainSwapsInternal()	Business Logic	Confirmed
PVE-002	Medium	Revisited Logic of LibInterchain::_handleCall()	Business Logic	Fixed
PVE-003	Informational	Improper Event Information in RangoArbitrumBridgeFacet::arbitrumSwapAndBridge()	Business Logic	Fixed
PVE-004	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	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 Untrusted User Input in LibSwapper::onChainSwapsInternal()

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: LibSwapper/LibInterchain
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Rango V2 protocol implements a cross-chain decentralized exchange (DEX) and multi-bridge aggregation (including Multichain, PolyNetwork, Synapse, cBridge, Axelar, and etc.). It allows the user to cross one token from the source chain to the destination chain. Especially, it allows the user to perform the intended token swap via the user specified DEXes on the source and/or destination chain. In particular, one entry routine, i.e., LibSwapper::onChainSwapsInternal(), is designed to perform the token swap on the source chain. While examining its logic, we notice there is a lack of input validation vulnerability that can be exploited.

To elaborate, we show below the related code snippet of the LibSwapper contract. Inside the onChainSwapsInternal() routine, the transferTokensFromUserForSwapRequest() and transferTokensFrom-UserForCalls() routines (lines 239/240) are called to pull the tokens from the user to the contract to prepare for the next token swap. Then the callSwapsAndFees() routine is called to perform the token swap via a series of external calls (lines 266 - 280). Apparently, there is no any validation for the external callData used in the external calls. It is possible for a malicious actor to hijack the input Call.swapFromToken/amount/callData to steal the funds locked in the contract.

```
228     function onChainSwapsInternal(
229         SwapRequest memory request,
230         Call[] calldata calls,
231         uint256 extraNativeFee
```

```
    ) internal returns (bytes[] memory, uint) {

        uint toBalanceBefore = getBalanceOf(request.toToken);
        uint fromBalanceBefore = getBalanceOf(request.fromToken);
        uint256[] memory initialBalancesList = getInitialBalancesList(calls);

        // transfer tokens from user for SwapRequest and Calls that require transfer
            from user.
        transferTokensFromUserForSwapRequest(request);
        transferTokensFromUserForCalls(calls);

        bytes[] memory result = callSwapsAndFees(request, calls);

        // check if any extra tokens were taken from contract and return excess tokens
            if any.
        returnExcessAmounts(request, calls, initialBalancesList);

        ...
    }

    function callSwapsAndFees(SwapRequest memory request, Call[] calldata calls) private
         returns (bytes[] memory) {
        bool isSourceNative = request.fromToken == ETH;
        BaseSwapperStorage storage baseSwapperStorage = getBaseSwapperStorage();

        for (uint256 i = 0; i < calls.length; i++) {
            require(baseSwapperStorage.whitelistContracts[calls[i].spender], "Contract
                spender not whitelisted");
            require(baseSwapperStorage.whitelistContracts[calls[i].target], "Contract
                target not whitelisted");
            bytes4 sig = bytes4(calls[i].callData[: 4]);
            require(baseSwapperStorage.whitelistMethods[calls[i].target][sig], "
                Unauthorized call data!");
        }

        ...

        // Execute swap Calls
        bytes[] memory returnData = new bytes[](calls.length);
        address tmpSwapFromToken;
        for (uint256 i = 0; i < calls.length; i++) {
            tmpSwapFromToken = calls[i].swapFromToken;
            bool isTokenNative = tmpSwapFromToken == ETH;
            if (isTokenNative == false)
                approveMax(tmpSwapFromToken, calls[i].spender, calls[i].amount);

            (bool success, bytes memory ret) = isTokenNative
            ? calls[i].target.call{value : calls[i].amount}(calls[i].callData)
            : calls[i].target.call(calls[i].callData);

            emit CallResult(calls[i].target, success, ret);
            if (!success)
```

```
                revert(_getRevertMsg(ret));
            returnData[i] = ret;
        }

        return returnData;
    }
```

Listing 3.1: `LibSwapper::onChainSwapsInternal()`

Recommendation Add necessary validation in above mentioned routines to prevent the input parameters from being hijacked. Note another routine, i.e., `LibInterchain::_handleCall()`, shares the similar issue.

Status The issue of the `LibInterchain::_handleCall()` routine has been addressed by the following commit: `618fece`. The similar issue in `LibSwapper::onChainSwapsInternal()` has been confirmed by the team. The team intends to leave it as is since the `RangoDiamond` contract will never lock any assets.

3.2 Revisited Logic of LibInterchain::_handleCall()

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

Description

As mentioned in Section 3.1, the `Rango V2` protocol allows the user to perform the intended token swap via the user specified `DEXes` on the source and/or destination chain. In particular, one entry routine, i.e., `LibInterchain::_handleCall()`, is designed to perform the token swap on the destination chain. While examining its logic, we notice its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `LibInterchain` contract. Inside the `_handleCall()` routine, it performs the user specified `preAction` if needed. If the `preAction` fails, it returns false (lines 228/229). However, we observe it will always return false if the user does not specify the `preAction` because the `ok` variable is not set to true in this scenario. Given this, we suggest to set the `ok` to true in this scenario.

Moreover, an external call (line 237) is executed to perform the token swap after the `preAction` is successful. If the external call fails, the third return value (i.e., `outToken`) is set to the input `_token` directly. Apparently, it ignores the fact that the `_token` may be wrapped or unwrapped in the `preAction` process. Give this, we suggest to improve the implementation as below: `return (false, _amount, sourceToken)` (line 245).

```
204     function _handleCall(
205         address _token,
206         uint _amount,
207         Interchain.RangoInterChainMessage memory _message,
208         LibSwapper.BaseSwapperStorage storage baseStorage
209     ) private returns (bool ok, uint256 amountOut, address outToken) {
210         Interchain.CallAction memory action = abi.decode((_message.action), (Interchain.CallAction));
211 
212         require(baseStorage.whitelistContracts[action.target] == true, "Action.target is not whitelisted");
213         require(baseStorage.whitelistContracts[action.spender] == true, "Action.spender is not whitelisted");
214 
215         address sourceToken = _token;
216 
217         if (action.preAction == Interchain.CallSubActionType.WRAP) {
218             require(_token == LibSwapper.ETH, "Cannot wrap non-native");
219             require(action.tokenIn == baseStorage.WETH, "action.tokenIn must be WETH");
220             (ok, amountOut, sourceToken) = _handleWrap(_token, _amount, baseStorage);
221         } else if (action.preAction == Interchain.CallSubActionType.UNWRAP) {
222             require(_token == baseStorage.WETH, "Cannot unwrap non-WETH");
223             require(action.tokenIn == LibSwapper.ETH, "action.tokenIn must be ETH");
224             (ok, amountOut, sourceToken) = _handleUnwrap(_token, _amount, baseStorage);
225         } else {
226             require(action.tokenIn == _token, "_message.tokenIn mismatch in call");
227         }
228         if (!ok)
229             return (false, _amount, _token);
230 
231         if (sourceToken != LibSwapper.ETH)
232             LibSwapper.approveMax(sourceToken, action.spender, _amount);
233 
234         uint value = sourceToken == LibSwapper.ETH ? _amount : 0;
235         uint toBalanceBefore = LibSwapper.getBalanceOf(_message.toToken);
236 
237         (bool success, bytes memory ret) = action.target.call{value: value}(action.callData);
238         if (success) {
239             emit ActionDone(Interchain.ActionType.CALL, action.target, true, "");
240 
241             uint toBalanceAfter = LibSwapper.getBalanceOf(_message.toToken);
242             return (true, toBalanceAfter - toBalanceBefore, _message.toToken);
243         } else {
244             emit ActionDone(Interchain.ActionType.CALL, action.target, false, LibSwapper._getRevertMsg(ret));
245             return (false, _amount, _token);
246         }
```

```
    }
```

Listing 3.2: `LibInterchain::_handleCall()`

Recommendation Correct the implementation of the `_handleCall()` routine as above-mentioned.

Status The issue has been addressed by the following commit: `a316b8a`.

3.3 Improper Event Information in RangoArbitrumBridgeFacet::arbitrumSwapAndBridge()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RangoArbitrumBridgeFacet`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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.

While examining the events that reflect the `RangoArbitrumBridgeFacet` dynamics, we notice there is an incorrect event information in the `arbitrumSwapAndBridge()` routine. To elaborate, we show below the related code snippet of the contract. By design, the `arbitrumSwapAndBridge()` routine is used to perform cross-chain transfer via `Arbitrum Bridge`. Within the routine, the `event RangoBridgeInitiated(address indexed requestId, address bridgeToken, uint256 bridgeAmount, address receiver, uint destinationChainId, bool hasInterchainMessage, bool hasDestinationSwap, uint8 indexed bridgeId, uint16 indexed dAppTag)` will be emitted to reflect the operation. According to the event definition, the eighth `bridgeId` parameter indicates the bridge protocol identity in the `Rango V2` protocol. However, we notice the `BridgeType.Hop` rather than the `BridgeType.ArbitrumBridge` is incorrectly used in the event (line 78).

```
50    function arbitrumSwapAndBridge(
51        LibSwapper.SwapRequest memory request,
52        LibSwapper.Call[] calldata calls,
53        IRangoArbitrum.ArbitrumBridgeRequest memory bridgeRequest
54    ) external payable nonReentrant {
55        uint out;
```

```
        uint bridgeAmount;
        // if toToken is native coin and the user has not paid fee in msg.value,
        // then the user can pay bridge fee using output of swap.
        if (request.toToken == LibSwapper.ETH && msg.value == 0) {
            out = LibSwapper.onChainSwapsPreBridge(request, calls, 0);
            bridgeAmount = out - bridgeRequest.cost;
        }
        else {
            out = LibSwapper.onChainSwapsPreBridge(request, calls, bridgeRequest.cost);
            bridgeAmount = out;
        }
        doArbitrumBridge(bridgeRequest, request.toToken, bridgeAmount);

        // event emission
        emit RangoBridgeInitiated(
            request.requestId,
            request.toToken,
            out,
            bridgeRequest.receiver,
            42161,
            false,
            false,
            uint8(BridgeType.Hop),
            request.dAppTag
        );
    }
```

Listing 3.3: `RangoArbitrumBridgeFacet::arbitrumSwapAndBridge()`

Recommendation Properly emit the above-mentioned event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commit: `9cec7b3`.

3.4 Redundant State/Code Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

While reviewing the implementation of `Rango V2` protocol, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. Using `OwnershipFacet::burnOwnership()` as an example, inside the routine, the statement of `Storage storage s = getStorage()`

(line 32) is executed to retrieve the privileged `owner`. However, we observe it is not used anywhere. Given this, we suggest to remove the redundant code safely (line 32).

```
30     function burnOwnership() external {
31         LibDiamond.enforceIsContractOwner();
32         Storage storage s = getStorage();
33         LibDiamond.setContractOwner(address(0));
34     }
```

Listing 3.4: `OwnershipFacet::burnOwnership()`

Note other routines, i.e., `LibInterchain::_handleUniswapV2()/_handleUniswapV3()` and `RangoMultichainFacet::multichainSwapAndBridge()/multichainBridge()`, can also be improved by removing the unnecessary redundancies.

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

Status The issue has been addressed by the following commit: `73d39e9`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Rango V2` implementation, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function addWhitelistContract(whitelistRequest[] calldata req) public {
        LibDiamond.enforceIsContractOwner();

        for (uint i = 0; i < req.length; i++) {
            LibSwapper.addMethodWhitelists(req[i].contractAddress, req[i].methodIds);
            emit ContractAndMethodsWhitelisted(req[i].contractAddress, req[i].methodIds)
                ;
            emit ContractWhitelisted(req[i].contractAddress);
        }
    }
```

Listing 3.5: `RangoAccessManagerFacet::addWhitelistContract()`

```
    function refund(address _tokenAddress, uint256 _amount) external onlyOwner {
        IERC20 ercToken = IERC20(_tokenAddress);
        uint balance = ercToken.balanceOf(address(this));
        require(balance >= _amount, 'Insufficient balance');

        SafeERC20.safeTransfer(IERC20(_tokenAddress), msg.sender, _amount);
        emit Refunded(_tokenAddress, _amount);
    }
```

Listing 3.6: `RangoBaseInterchainMiddleware::refund()`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it will be worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest a multi-sig account plays the privileged `owner` account to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Rango V2` protocol, which implements a cross-chain decentralized exchange (`DEX`). It provides the cross-chain swap service with a one-transaction user experience. Additionally, it implements the multi-bridge aggregation, including `Multichain`, `PolyNetwork`, `Synapse`, `cBridge`, `Axelar`, and etc. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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