SMART CONTRACT AUDIT REPORT

for

KratosDAO

Prepared By: Yiqun Chen

PeckShield

January 15, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	January 15, 2022	Xuxian Jiang	Final Release
1.0-rc	January 13, 2022	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

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

`KratosDAO` is an algorithmic currency protocol based on the initial `OlympusDAO` protocol. 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 existing stablecoin solutions, it is proposed as a non-pegged stablecoin by exploring a radical opportunity to achieve stability while eliminating dependence on fiat currencies. The basic information of the `KratosDAO` protocol is as follows:

Table 1.1: Basic Information of The `KratosDAO` Protocol

Item	Description
Issuer	KratosDAO
Website	https://kratosdao.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 15, 2022

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

- https://github.com/Kratos-Dao/contracts.git (b9f5fba)

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

- https://github.com/Kratos-Dao/contracts.git (dffa6fc)

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

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.

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

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 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 KratosDAO 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 logics, 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	0	
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 1 medium-severity vulnerability and 4 low-severity vulnerabilities.

Table 2.1: Key KratosDAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect Reserve/Liquidity Management in Treasury::toggle()	Business Logic	Fixed
PVE-002	Low	Improved Reward Calculation in StakingDistributor::nextRewardFor()	Business Logic	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Low	Potential Arithmetic Underflows of Bonding Calculation	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect Reserve/Liquidity Management in Treasury::toggle()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KratosTreasury`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `KratosDAO` protocol has a treasury contract, i.e., `KratosTreasury`, that allows for taking reserve tokens (e.g., `DAI`) and minting managed tokens (e.g., `Kratos`). The treasury contact can also take the principle tokens (e..g, `Kratos-DAI SLP`) and mint the managed tokens according to the bonding curve-based principle evaluation. In the following, we examine the management routine `toggle()` and report an issue for correction.

Specifically, this `toggle()` routine is designed to configure various queues of members and each member may be assigned one specific role in the protocol. It comes to our attention that two queues, i.e., `RESERVE MANAGER` and `LIQUIDITY DEPOSITOR`, are blindly appended in the current implementation (lines 586-606). By design, each queue requires proper validation before the new member can be added.

```
559     function toggle(
560         MANAGING _managing,
561         address _address,
562         address _calculator
563     ) external onlyManager() returns ( bool ) {
564         require( _address != address(0) );
565         bool result;
566         if ( _managing == MANAGING.RESERVEDEPOSITOR ) { // 0
567             if ( requirements( reserveDepositorQueue, isReserveDepositor, _address ) ) {
568                 reserveDepositorQueue[ _address ] = 0;
```

```
            if( !listContains( reserveDepositors, _address ) ) {
                reserveDepositors.push( _address );
            }
        }
        result = !isReserveDepositor[ _address ];
        isReserveDepositor[ _address ] = result;

    } else if ( _managing == MANAGING.RESERVESPENDER ) { // 1
        if ( requirements( reserveSpenderQueue, isReserveSpender, _address ) ) {
            reserveSpenderQueue[ _address ] = 0;
            if( !listContains( reserveSpenders, _address ) ) {
                reserveSpenders.push( _address );
            }
        }
        result = !isReserveSpender[ _address ];
        isReserveSpender[ _address ] = result;

    } else if ( _managing == MANAGING.RESERVETOKEN ) { // 2
        if ( requirements( reserveTokenQueue, isReserveToken, _address ) ) {
            reserveTokenQueue[ _address ] = 0;
            if( !listContains( reserveTokens, _address ) ) {
                reserveTokens.push( _address );
            }
        }
        result = !isReserveToken[ _address ];
        isReserveToken[ _address ] = result;

    } else if ( _managing == MANAGING.RESERVEMANAGER ) { // 3
        if ( requirements( ReserveManagerQueue, isReserveManager, _address ) ) {
            reserveManagers.push( _address );
            ReserveManagerQueue[ _address ] = 0;
            if( !listContains( reserveManagers, _address ) ) {
                reserveManagers.push( _address );
            }
        }
        result = !isReserveManager[ _address ];
        isReserveManager[ _address ] = result;

    } else if ( _managing == MANAGING.LIQUIDITYDEPOSITOR ) { // 4
        if ( requirements( LiquidityDepositorQueue, isLiquidityDepositor, _address )
            ) {
            liquidityDepositors.push( _address );
            LiquidityDepositorQueue[ _address ] = 0;
            if( !listContains( liquidityDepositors, _address ) ) {
                liquidityDepositors.push( _address );
            }
        }
        result = !isLiquidityDepositor[ _address ];
        isLiquidityDepositor[ _address ] = result;

    } else if ( _managing == MANAGING.LIQUIDITYTOKEN ) { // 5
        if ( requirements( LiquidityTokenQueue, isLiquidityToken, _address ) ) {
```

```
                LiquidityTokenQueue[ _address ] = 0;
                if( !listContains( liquidityTokens, _address ) ) {
                    liquidityTokens.push( _address );
                }
            }
            result = !isLiquidityToken[ _address ];
            isLiquidityToken[ _address ] = result;
            bondCalculator[ _address ] = _calculator;

        } else if ( _managing == MANAGING.LIQUIDITYMANAGER ) { // 6
            if ( requirements( LiquidityManagerQueue, isLiquidityManager, _address ) ) {
                LiquidityManagerQueue[ _address ] = 0;
                if( !listContains( liquidityManagers, _address ) ) {
                    liquidityManagers.push( _address );
                }
            }
            result = !isLiquidityManager[ _address ];
            isLiquidityManager[ _address ] = result;

        } else if ( _managing == MANAGING.DEBTOR ) { // 7
            if ( requirements( debtorQueue, isDebtor, _address ) ) {
                debtorQueue[ _address ] = 0;
                if( !listContains( debtors, _address ) ) {
                    debtors.push( _address );
                }
            }
            result = !isDebtor[ _address ];
            isDebtor[ _address ] = result;

        } else if ( _managing == MANAGING.REWARDMANAGER ) { // 8
            if ( requirements( rewardManagerQueue, isRewardManager, _address ) ) {
                rewardManagerQueue[ _address ] = 0;
                if( !listContains( rewardManagers, _address ) ) {
                    rewardManagers.push( _address );
                }
            }
            result = !isRewardManager[ _address ];
            isRewardManager[ _address ] = result;

        } else if ( _managing == MANAGING.SOHM ) { // 9
            sOHMQueue = 0;
            MEMOries = _address;
            result = true;

        } else return false;

        emit ChangeActivated( _managing, _address, result );
        return true;
    }
```

Listing 3.1: KratosTreasury::toggle()

Recommendation Revise the `toggle()` logic to properly add new members to the respective queues.

Status This issue has been fixed in the following commit: `9dba105`.

3.2 Improved Reward Calculation in StakingDistributor::nextRewardFor()

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

Description

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

To elaborate, we show below the `nextRewardFor()` routine that computes the next reward for specified address. Our analysis shows that the current implementation does not take into account that the same recipient may be rewarded twice for distributions. As a result, we need to compute the accumulative reward for the given recipient.

```
    function nextRewardFor( address _recipient ) public view returns ( uint ) {
        uint reward;
        for ( uint i = 0; i < info.length; i++ ) {
            if ( info[ i ].recipient == _recipient ) {
                reward = nextRewardAt( info[ i ].rate );
            }
        }
        return reward;
    }
```

Listing 3.2: `StakingDistributor::nextRewardFor()`

Recommendation Correct the above `nextRewardFor()` logic for the right amount of rewards.

Status This issue has been fixed in the following commit: `9dba105`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: KratosTreasury
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the transfer() routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the transfer() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.3: USDT::transfer()

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.

In current implementation, if we examine the KratosTreasury::incurDebt() routine that is designed to allow approved addresses to borrow reserves. To accommodate the specific idiosyncrasy, there is a need to user safeTransfer(), instead of transfer() (line 398).

```
382    function incurDebt( uint _amount, address _token ) external {
```

```
        require( isDebtor[ msg.sender ], "Not approved" );
        require( isReserveToken[ _token ], "Not accepted" );

        uint value = valueOf( _token, _amount );

        uint maximumDebt = IERC20( MEMOries ).balanceOf( msg.sender ); // Can only
            borrow against sOHM held
        uint availableDebt = maximumDebt.sub( debtorBalance[ msg.sender ] );
        require( value <= availableDebt, "Exceeds debt limit" );

        debtorBalance[ msg.sender ] = debtorBalance[ msg.sender ].add( value );
        totalDebt = totalDebt.add( value );

        totalReserves = totalReserves.sub( value );
        emit ReservesUpdated( totalReserves );

        IERC20( _token ).transfer( msg.sender, _amount );

        emit CreateDebt( msg.sender, _token, _amount, value );
    }
```

Listing 3.4: `KratosTreasury::incurDebt()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `9dba105`.

3.4 Potential Arithmetic Underflows of Bonding Calculation

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `KratosBondingCalculator`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the `KratosDAO` protocol has a treasury contract, i.e., `KratosTreasury`, that allows for taking reserve tokens (e.g., `DAI`) and minting managed tokens (e.g., `Kratos`). The treasury contact can also take the principle tokens (e..g, `Kratos-DAI SLP`) and mint the managed tokens according to the bonding curve-based principle evaluation. In the following, we examine the bonding curve evaluation.

To elaborate, we show below the key `getKValue()` function that is proposed to compute the current constant product `K` value. However, it comes to our attention that the `decimals` computation can be improved as the sum of two constituent tokens' decimals may not be greater than the pair's 18 decimal. In this case, the computation of `decimals` may be reverted!

```
function getKValue( address _pair ) public view returns( uint k_ ) {
    uint token0 = IERC20( IUniswapV2Pair( _pair ).token0() ).decimals();
    uint token1 = IERC20( IUniswapV2Pair( _pair ).token1() ).decimals();
    uint decimals = token0.add( token1 ).sub( IERC20( _pair ).decimals() );

    (uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();
    k_ = reserve0.mul(reserve1).div( 10 ** decimals );
}
```

Listing 3.5: `KratosTreasury::getKValue()`

Fortunately, the managed token `Kratos` has the decimal of 9 and the reserve token `DAI` has the decimal of 18. As a result, it still results in the same converted (absolute) amount. However, the revised conversion logic is generic in accommodating other token setups, especially when the managed token does not have 9 as its decimal.

Recommendation Revise the `getKValue()` logic to compute the constant product smoothly.

Status This issue has been fixed in the following commit: `9dba105`.

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 [2]

Description

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

```
/**
    @notice sets the contract address for LP staking
    @param _contract address
 */
function setContract( CONTRACTS _contract, address _address ) external onlyManager()
     {
    if( _contract == CONTRACTS.DISTRIBUTOR ) { // 0
```

```
            distributor = _address;
        } else if ( _contract == CONTRACTS.WARMUP ) { // 1
            require( warmupContract == address( 0 ), "Warmup cannot be set more than
                once" );
            warmupContract = _address;
        } else if ( _contract == CONTRACTS.LOCKER ) { // 2
            require( locker == address(0), "Locker cannot be set more than once" );
            locker = _address;
        }
    }

    /**
     * @notice set warmup period in epoch's numbers for new stakers
     * @param _warmupPeriod uint
     */
    function setWarmup( uint _warmupPeriod ) external onlyManager() {
        warmupPeriod = _warmupPeriod;
    }
```

Listing 3.6: Example Privileged Operations in `Staking`

```
    function setVault( address vault_ ) external onlyOwner() returns ( bool ) {
      _vault = vault_;

      return true;
    }

    function mint(address account_, uint256 amount_) external onlyVault() {
        _mint(account_, amount_);
    }
```

Listing 3.7: Example Privileged Operations in `KratosERC20Token`

We emphasize that the privilege assignment with various factory contracts is necessary and required for proper protocol operations. However, it will be worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised `owner` account would allow the attacker to change current `vault` to mint arbitrary number of `Kratos` or change other settings (e.g., `stakingContract`) to steal funds of currently staking users, which directly undermines the integrity of the `KratosDAO` protocol.

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 and partially mitigated with the planned multi-sig `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of `KratosDAO`, which utilizes the protocol owned value to enable price consistency and scarcity within an infinite supply system. During the audit, we notice that the current implementation still remains to be completed, though the overall code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

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