SMART CONTRACT AUDIT REPORT

for

MaGaugeV2Upgradeable && ChronosMarketplace

Prepared By: Xiaomi Huang

PeckShield

August 16, 2023

Document Properties

Client	Chronos
Title	Smart Contract Audit Report
Target	MaGaugeV2Upgradeable && ChronosMarketplace
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	August 16, 2023	Stephen Bie	Final Release
1.0-rc	July 14, 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 source code of the specific contracts, i.e., MaGaugeV2Upgradeable and ChronosMarketplace, 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 Chronos

Chronos is a community-owned decentralized exchange (DEX) constructed on the Arbitrum Layer 2 (L2) network, aiming at fostering DeFi growth through sustainable liquidity incentives. This audit covers two contracts, i.e., MaGaugeV2Upgradeable and ChronosMarketplace. The first contract implements an incentive mechanism that rewards the staking of the supported LP token with the CHR token and the second one provides users with a trustless NFT trading market. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of MaGaugeV2Upgradeable && ChronosMarketplace

Item	Description
Target	MaGaugeV2Upgradeable && ChronosMarketplace
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 16, 2023

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. (In the first repository, our audit only covers the MaGaugeV2Upgradeable.sol contract.)

- https://github.com/ChronosEx/Chronos-ContractsV2.git (7bdf718)

- https://github.com/GruDev325/sc_chronos_marketplace.git (3918a2a)

And these are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/ChronosEx/Chronos-ContractsV2.git (1fd8feb)
- https://github.com/GruDev325/sc_chronos_marketplace.git (8733b13)

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.

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 1.3: 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

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 `MaGaugeV2Upgradeable && ChronosMarketplace` 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	2	■ ■
High	3	■ ■ ■
Medium	1	■
Low	0	
Informational	0	
Total	6	

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

2.2 Key Findings

Overall, these smart contracts should be improved by resolving the identified issues (shown in Table 2.1), including 2 critical-severity vulnerabilities, 3 high-severity vulnerabilities, and 1 medium-severity vulnerability.

Table 2.1: Key MaGaugeV2Upgradeable && ChronosMarketplace Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Revisited Logic of MaGaugeV2Upgradeable::split()	Business Logic	Fixed
PVE-002	High	Revisited Precision in MaGaugeV2Upgradeable::rewardPerToken()	Business Logic	Fixed
PVE-003	Critical	Revisited Logic of ChronosMarketplace::placeBidWithETH()	Business Logic	Fixed
PVE-004	High	Possible Front-Running for ChronosMarketplace::buyNow()	Time and State	Fixed
PVE-005	High	Lack of Input Validation in ChronosMarketplace::makeOfferWithETH()	Business Logic	Fixed
PVE-006	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 Revisited Logic of MaGaugeV2Upgradeable::split()

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: MaGaugeV2Upgradeable
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The MaGaugeV2Upgradeable contract implements an incentive mechanism, which allows the user to deposit the supported LP token to earn the CHR token. Meanwhile, an maNFT is minted to uniquely identify the user's deposit position. In particular, one entry routine, i.e., split(), is designed to split the given deposit position (specified by the input _maNFTId) into multiple new smaller positions according to the user specified weights. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the contract. Inside the split() routine, the given maNFT (representing the previous deposit position) is burnt (line 527) and multiple new maNFTs are minted (line 510) to represent the new deposit positions. However, we observe the reward-related state variables (e.g., idRewardPerTokenPaid and _positionLastWeights) are not timely initialized when the new maNFT is minted, which will result in the user getting more rewards than expected.

Moreover, we observe the reward of the previous deposit position is not timely transferred to the user. Apparently, it ignores the fact that the user will not be able to claim the reward after the corresponding maNFT is burnt. Note another routine, i.e., merge(), shares the same issue.

```
490     function split(
491         uint _maNFTId,
492         uint[] calldata weights
493     ) external nonReentrant isNotEmergency updateTotalWeight updateReward(_maNFTId) {
494         require(
495             _isApprovedOrOwner(_msgSender(), _maNFTId),
```

```
            "maNFT: caller is not token owner or approved"
        );

        // limit the weights length to avoid out of gas
        require(weights.length <= MAX_SPLIT_WEIGHTS, "Max splitted positions exceeded");

        uint weightsSum = 0;

        for (uint i; i < weights.length; i++) {
            weightsSum += weights[i];

            uint splitAmount = (weights[i] * _lpBalances[_maNFTId]) / WEIGHTS_MAX_POINTS;
            require(splitAmount > 0, "deposit(Gauge): cannot stake 0");
            uint _newMANFTId = tokenId;
            _mint(_msgSender(), _newMANFTId); // potentially, use ownerOf(_maNFTId)
            tokenId++;

            _lpBalances[_newMANFTId] = splitAmount;
            _positionEntries[_newMANFTId] = _positionEntries[_maNFTId];
            _nftToEpochIds[_newMANFTId] = _nftToEpochIds[_maNFTId];
        }

        // bps accuracy is used for e.g.
        require(weightsSum == WEIGHTS_MAX_POINTS, "Invalid weights sum");

        // total weight doesn't change as liquidity and maturity didn't change
        _lpBalances[_maNFTId] = 0;
        _positionEntries[_maNFTId] = 0;
        _positionLastWeights[_maNFTId] = 0;
        _nftToEpochIds[_maNFTId] = 0;

        _burn(_maNFTId);

        emit Split(_msgSender(), _maNFTId); // potentially, use owner of nft
    }
```

Listing 3.1: `MaGaugeV2Upgradeable::split()`

Recommendation Timely distribute the reward before the `maNFT` is burnt and initialize the reward-related state variables when the new `maNFT` is minted.

Status The issue has been addressed in the following commit: `67cbe1c`.

3.2 Revisited Precision in MaGaugeV2Upgradeable::rewardPerToken()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `MaGaugeV2Upgradeable`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

As mentioned in Section 3.1, the `MaGaugeV2Upgradeable` contract implements an incentive mechanism that rewards the staking of the supported `LP` token with the `CHR` token. In particular, one entry routine, i.e., `rewardPerToken()`, is designed to calculate the accumulated reward per token. While examining its logic, we observe its precision calculation needs to be improved.

To elaborate, we show below the related code snippet of the contract. Inside the `rewardPerToken()` routine, the formula of `rewardPerTokenStored + (((lastTimeRewardApplicable()- lastUpdateTime)* rewardRate * 1e18)/ _totalWeight)` is used to calculate the accumulated reward per token. If we only focus on its precision, it can be simplified as `1e18 * 1e18 / 1e18 * 1e18 = 1` (assuming the decimals of the `rewardToken` is 18). The precision of the accumulated reward per token should be `1e18` by design and thus the formula is incorrect.

```
    function rewardPerToken() public view returns (uint) {
        if (_totalWeight == 0) {
            return rewardPerTokenStored;
        } else {
            return
                rewardPerTokenStored +
                (((lastTimeRewardApplicable() - lastUpdateTime) *
                    rewardRate *
                    1e18) / _totalWeight);
        }
    }
```

Listing 3.2: `MaGaugeV2Upgradeable::rewardPerToken()`

Recommendation Revisit the precision calculation in above-mentioned routine.

Status The issue has been addressed in the following commit: `67cbe1c`.

3.3 Revisited Logic of ChronosMarketplace::placeBidWithETH()

- ID: PVE-003
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: ChronosMarketplace
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The ChronosMarketplace contract provides users with a trustless NFT trading market, which supports three kinds of trading modes: Fixed Price, English Auction, and Limit Order. In particular, one entry routine, i.e., placeBidWithETH(), is designed to bid for a given English Auction with ETH. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the contract. Inside the placeBidWithETH() routine, we notice payable(auctionInfo.highestBidder).transfer(oldBidPrice) (line 592) is called to refund the ETH to the last bidder. However, it comes to our attention that the auctionInfo.highestBidder is set to the new bidder (i.e., msg.sender, line 588) before, which directly undermines the assumption of the protocol design.

```
578     function placeBidWithETH(
579         uint256 _auctionId
580     ) external payable override nonReentrant whenNotPaused {
581         address bidder = msg.sender;
582         ...
583         require(bidPrice > minimumBidPrice, Errors.LOW_BID_PRICE);
584 
585         address oldWinner = auctionInfo.highestBidder;
586         uint256 oldBidPrice = auctionInfo.highestBidPrice;
587 
588         auctionInfo.highestBidder = bidder;
589         auctionInfo.highestBidPrice = bidPrice;
590 
591         if (oldWinner != address(0)) {
592             payable(auctionInfo.highestBidder).transfer(
593                 oldBidPrice
594             );
595         }
596 
597         emit PlaceBid(bidder, _auctionId, bidPrice);
598     }
```

Listing 3.3: ChronosMarketplace::placeBidWithETH()

Moreover, we observe it is exposed to potential `DoS` risks. If the last bidder is a malicious contract, it can always revert the transaction in its `receive()/fallback()` routine. By doing so, he can win the auction eventually. Note other routines, i.e., `cancelListNftForAuction()/finishAuction()/placeBid()`, are vulnerable to this `DoS` attack as well.

Recommendation Properly refund the assets to the last bidder and apply a defense mechanism to avoid possible `DoS` attack.

Status The issue has been addressed in the following commit: `9a0c870`.

3.4 Possible Front-Running for ChronosMarketplace::buyNow()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `ChronosMarketplace`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

As mentioned in Section 3.3, the `ChronosMarketplace` contract supports the `Fixed Price` trading mode. The buyer can purchase a listed `NFT` with a fixed price (specified by the owner of the `NFT`) via `buyNow()`. While examining its logic, we observe it is vulnerable to the possible `front-running` attack.

To elaborate, we show below the related code snippet of the contract. A malicious actor can list his `NFT` with a very low price via `listNftForFixed()`. If a user wants to purchase the `NFT` with the low price via `buyNow()`, the malicious actor may front-run `changeSaleInfo()` to make the `NFT` price higher. After that, the buyer will suffer from an unexpected loss.

```
    function changeSaleInfo(
        uint256 _saleId,
        uint256 _saleDuration,
        address _paymentToken,
        uint256 _price
    ) external override nonReentrant whenNotPaused {
        ...

        sellInfo.startTime = block.timestamp;
        sellInfo.endTime = sellInfo.startTime + _saleDuration;
        sellInfo.paymentToken = _paymentToken;
        sellInfo.price = _price;

        ...
    }

```

```
    /// @inheritdoc IChronosMarketPlace
    function buyNow(
        uint256 _saleId
    ) external override nonReentrant whenNotPaused {
        ...

        IERC20(saleInfo.paymentToken).safeTransferFrom(
            buyer,
            saleInfo.seller,
            saleInfo.price - fee
        );
        IERC20(saleInfo.paymentToken).safeTransferFrom(buyer, treasury, fee);

        IERC721(saleInfo.nft).safeTransferFrom(
            address(this),
            buyer,
            saleInfo.tokenId
        );

        emit Bought(_saleId, saleInfo.buyer);
    }
```

Listing 3.4: `ChronosMarketplace::changeSaleInfo()&&buyNow()`

Note another routine, i.e., `buyNowWithETH()`, shares the same issue.

Recommendation Apply necessary anti-frontrunning mechanism to above-mentioned routines.

Status The issue has been addressed in the following commits: `9a0c870` and `587320b`.

3.5 Lack of Input Validation in ChronosMarketplace::makeOfferWithETH()

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `ChronosMarketplace`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

As mentioned in Section 3.3, the `ChronosMarketplace` contract supports the `Limit Order` trading mode. In particular, one entry routine, `makeOfferWithETH()`, allows for submitting a limit order to buy the given `NFT` with `ETH`. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the contract. The `makeOfferWithETH()` routine allows the user to provide an arbitrary `_paymentToken` (e.g., `WBTC`) without any validation. With that, a malicious actor can steal the `WBTC` from the `ChronosMarketplace` contract via `cancelOffer()` even though he will lose the same amount of `ETH`.

```
    function makeOfferWithETH(
        address _nft,
        uint256 _tokenId,
        address _paymentToken,
        uint256 _offerPrice
    ) external payable override nonReentrant whenNotPaused {
        require(isChronosNft(_nft), Errors.NOT_CHRONOS_NFT);

        require(
            msg.value >= _offerPrice && _offerPrice > 0,
            Errors.INVALID_PRICE
        );

        address offeror = msg.sender;

        _setOfferId(offerId, offeror, true);

        offerInfos[offerId++] = OfferInfo(
            offeror,
            _paymentToken,
            _nft,
            _tokenId,
            _offerPrice
        );
        ...
    }

    function cancelOffer(uint256 _offerId) external override nonReentrant whenNotPaused
        {
        ...
        if (offerInfo.paymentToken == address(0)) {
            payable(sender).transfer(offerInfo.offerPrice);
        } else {
            IERC20(offerInfo.paymentToken).safeTransfer(
                sender,
                offerInfo.offerPrice
            );
        }

        emit CancelOffer(_offerId);
    }
```

Listing 3.5: `ChronosMarketplace::makeOfferWithETH()&&cancelOffer()`

Recommendation Validate the input `_paymentToken` parameter in the `makeOfferWithETH()` routine.

Status The issue has been addressed in the following commit: `9a0c870`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MaGaugeV2Upgradeable/ChronosMarketplace`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the audited `MaGaugeV2Upgradeable && ChronosMarketplace` contracts, 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 setDistribution(address _distribution) external onlyOwner {
        require(_distribution != address(0), "zero addr");
        require(_distribution != DISTRIBUTION, "same addr");
        DISTRIBUTION = _distribution;
    }

    function setInternalBribe(address _int) external onlyOwner {
        require(_int >= address(0), "zero");
        internal_bribe = _int;
    }
```

Listing 3.6: `MaGaugeV2Upgradeable::setDistribution()&&setInternalBribe()`

```
    function setAllowedToken(
        address[] memory _tokens,
        bool _isAdd
    ) external override onlyOwner {
        uint256 length = _tokens.length;
        require(length > 0, Errors.INVALID_LENGTH);
        for (uint256 i = 0; i < length; i++) {
            allowedTokens[_tokens[i]] = _isAdd;
        }

        emit AllowedTokenSet(_tokens, _isAdd);
    }
```

Listing 3.7: `ChronosMarketplace::setAllowedToken()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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

Status The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of two contracts, i.e., `MaGaugeV2Upgradeable` and `ChronosMarketplace`. The first contract implements an incentive mechanism that rewards the staking of the supported `LP` token with the `CHR` token and the second contract provides users with a trustless `NFT` trading market. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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