SMART CONTRACT AUDIT REPORT

for

Unlimited Leverage

Prepared By: Xiaomi Huang

PeckShield

January 17, 2023

Document Properties

Client	Solidant
Title	Smart Contract Audit Report
Target	Unlimited Leverage
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 17, 2023	Jing Wang	Final Release
1.0-rc	January 5, 2023	Jing Wang	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 `Unlimited Leverage` design document and related smart contract source code, 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 Unlimited Leverage

`Unlimited Leverage` is a synthetic cryptocurrency trading platform, where users can trade across a range of cryptocurrencies and blockchain networks. The protocol offers zero price impact and minimal slippage during trading, as well as up to 100x leverage. This leverage enables users to place up to 100X more value into trades than would otherwise be possible. The basic information of `Unlimited Leverage` is as follows:

Table 1.1: Basic Information of Unlimited Leverage

Item	Description
Issuer	Solidant
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 17, 2023

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

- https://github.com/solidant/unlimited-contracts (cc0dc1f)

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

- https://github.com/solidant/unlimited-contracts (a9639b7)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [12]:

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

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

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

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

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered 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 `Unlimited Leverage` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	1	■
High	1	■
Medium	1	■
Low	1	■
Informational	2	■ ■
Undetermined	1	■
Total	7	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 critical-severity vulnerability, 1 high-severity vulnerability, 1 medium-severity vulnerability, 1 low-severity vulnerability, 2 informational recommendations, and 1 undetermined issue.

Table 2.1: Key Audit Findings of Unlimited Leverage

ID	Severity	Title	Category	Status
PVE-001	High	Exposure Of Permissioned UserManager::setUserReferrer()	Security Features	Fixed
PVE-002	Critical	Improved Logic of openPositionViaSignature()	Business Logic	Fixed
PVE-003	Low	Missing Validation for Value Of targetLeverage_	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Informational	Generation of Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-006	Informational	Redundant Code Removal	Coding Practices	Fixed
PVE-007	Undetermined	Potential Protocol Risk from Low-Liquidity Assets	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 Exposure Of Permissioned UserManager::setUserReferrer()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `UserManager`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

The `Unlimited Leverage` protocol has the `UserManager` contract to keep track of daily and monthly trading volume of users and accordingly facilitate the calculation of user fee. It also provides corresponding `setter` functions. By design, these `setter` functions need to be guarded to prevent unauthorized changes.

When examining these `setter` functions, we notice the presence of a specific routine, i.e., `setUserReferrer()`. As the name indicates, this routine is used to set the referrer of the user. To elaborate, we show below the code snippet of this function.

```
    function setUserReferrer(address user_, address referrer_) external {
        require(user_ != referrer_, "UserManager::setUserReferrer: User cannot be
            referrer");
        if (_userReferrer[user_] == address(0)) {
            if (referrer_ == address(0)) {
                _userReferrer[user_] = NO_REFERRER_ADDRESS;
            } else {
                _userReferrer[user_] = referrer_;
            }

            emit UserReferrerAdded(user_, referrer_);
        }
    }
```

Listing 3.1: `UserManager::setUserReferrer()`

However, we notice that this routine is currently permissionless, which means it can be invoked by anyone to set the referrer of the user. Also, the referrer can only be set once. A bad actor could monitor the open position transaction from memory pool and front run the transaction to set the user referrer to his own address. To fix this issue, the permissionless function needs to be changed to be permissioned such that only the intended pair contract is allowed to successfully invoke it.

Recommendation Add the `onlyValidTradePair(msg.sender)` modifier to the above `setUserReferrer()` function.

Status This issue has been fixed in the following commit: `c571c9a`.

3.2 Improved Logic of openPositionViaSignature()

- ID: PVE-002
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `TradeManagerOrders`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Unlimited Leverage` protocol, there are two types of trading methods. One is to interact directly with the contract from the order maker. The other is to authorize trades by signing an order by the order maker. When reviewing the second method, we found there is a logical issue. To elaborate, we show below the related `openPositionViaSignature()` routine as an example.

```
    function openPositionViaSignature(
        OpenPositionOrder calldata order_,
        UpdateData[] calldata updateData_,
        address maker_,
        bytes calldata signature_
    ) external onlyActiveTradePair(order_.params.tradePair) returns (uint256) {
        _updateContracts(updateData_);
        _processSignature(order_, maker_, signature_);
        _verifyConstraints(
            order_.params.tradePair, order_.constraints, order_.params.isShort ?
                UsePrice.MAX : UsePrice.MIN
        );
        uint256 positionId = _openPosition(order_.params);

        sigHashToTradeId[keccak256(signature_)] = TradeId(order_.params.tradePair,
            uint96(positionId));

        emit OpenedPositionViaSignature(order_.params.tradePair, positionId, signature_)
            ;
```

```

        return positionId;
    }


    function _openPosition(OpenPositionParams memory params_) internal returns (uint256)
     {
        ITradePair(params_.tradePair).collateral().safeTransferFrom(
            msg.sender, address(params_.tradePair), params_.margin
        );

        userManager.setUserReferrer(msg.sender, params_.referrer);

        uint256 id = ITradePair(params_.tradePair).openPosition(
            msg.sender, params_.margin, params_.leverage, params_.isShort, params_.
                whitelabelAddress
        );

        emit PositionOpened(params_.tradePair, id);

        return id;
    }
```

Listing 3.2: `TradeManagerOrders::openPositionViaSignature()`

It comes to our attention that the `_processSignature()` routine validates the order signed by the maker. However, the `_openPosition()` routine does not take funds from the maker. Instead, it transfers funds from the `msg.sender`, which will not work as expected.

Recommendation Revise the related routines to transfer funds from maker rather than `msg.sender`.

Status This issue has been fixed in the following commit: `c2b2408`.

3.3 Missing Validation for Value Of targetLeverage_

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TradePair`
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [5]

Description

In the `TradePair` contract, the `extendPositionToLeverage()` function is used to extend loan positions to target an intended leverage in respect to `LEVERAGE_MULTIPLIER`. To elaborate, we show below the related code snippet.

```
    function extendPositionToLeverage(address maker_, uint256 positionId_, uint256 targetLeverage_)
        external
        onlyTradeManager
        verifyOwner(maker_, positionId_)
        syncFeesBefore
        updatePositionFees(positionId_)
        onlyValidAlteration(positionId_)
        checkSizeLimitAfter
    {
        _extendPositionToLeverage(positionId_, targetLeverage_);
    }
```

Listing 3.3: `TradePair::extendPositionToLeverage()`

We notice that this function has an assumption that `targetLeverage_` is less than `maxLeverage`. However, there is no actual enforcement of this assumption in current implementation. If an invalid `targetLeverage_` is added, it may cause unexpected behaviors and affect users' loan positions.

Recommendation Add the requirement of `_verifyLeverage(targetLeverage_)` in `TradePair::extendPositionToLeverage()`.

Status This issue has been fixed in the following commit: `95378c7`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

In the `Unlimited Leverage` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameters setting). 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 `owner` account and its related privileged accesses in current contract.

To elaborate, we show the related routines from the `Controller` contract. These routines allow the `owner` account to configure parameters like `isPriceFeed` and `isLiquidityPoolAdapter`, which play important role for configuring oracle prices/holding user funds.

```
    function addPriceFeed(address priceFeed_) external onlyOwner onlyNonZeroAddress(priceFeed_) {
```

```
        isPriceFeed[priceFeed_] = true;

        emit PriceFeedAdded(priceFeed_);
    }

    function addLiquidityPoolAdapter(address liquidityPoolAdapter_)
        external
        onlyOwner
        onlyNonZeroAddress(liquidityPoolAdapter_)
    {
        isLiquidityPoolAdapter[liquidityPoolAdapter_] = true;

        emit LiquidityPoolAdapterAdded(liquidityPoolAdapter_);
    }
```

Listing 3.4: Admin Controlled Routines

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a 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 confirmed. The team clarifies the project will be deployed with a multi-signature account scheme for the owner, likely Gnosis Safe. Furthermore, all admin functionality will eventually be behind a timelock, ensuring that any potential unauthorized actions can be inspected before execution.

3.5 Generation of Meaningful Events For Important State Changes

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Controller`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [2]

Description

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

In the following, we use the Controller contract as an example. This contract has several privileged functions that are used to configure various system parameters. While examining the events that reflect their changes, we notice there is a lack of emitting important events that reflect important state changes. For example, when the isSigner map is updated, there is no respective event being emitted (line 217 and 225)

```
212     /**
213      * @notice Function to add a valid signer
214      * @param signer_ address of the signer
215      */
216     function addSigner(address signer_) external onlyOwner {
217         isSigner[signer_] = true;
218     }

220     /**
221      * @notice Function to remove a valid signer
222      * @param signer_ address of the signer
223      */
224     function removeSigner(address signer_) external onlyOwner {
225         isSigner[signer_] = false;
226     }
```

Listing 3.5: Controller::addSigner()/removeSigner()

Recommendation Properly emit respective events when important states are updated.

Status This issue has been fixed in the following commit: 8acdea5.

3.6 Redundant Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: TradePair
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

Description

The Unlimited Leverage protocol makes good use of a number of reference contracts, such as SafeERC20, OwnableUpgradeable, and Initializable, to facilitate its code implementation and organization. For example, the TradePair contract has so far imported at least three reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the _extendPosition() function in the TradePair contract, this function makes a redundant requirement of verifyLeverage() with extendPosition().

```
    function extendPosition(address maker_, uint256 positionId_, uint256 addedMargin_, uint256 addedLeverage_)
        external
        onlyTradeManager
        verifyOwner(maker_, positionId_)
        verifyLeverage(addedLeverage_)
        syncFeesBefore
        updatePositionFees(positionId_)
        onlyValidAlteration(positionId_)
        checkSizeLimitAfter
    {
        _extendPosition(maker_, positionId_, addedMargin_, addedLeverage_);
    }

    /**
     * @notice Should have received margin from TradeManager
     * @dev extendPosition simply "adds" a "new" position on top of the existing position. The two positions get merged.
     */
    function _extendPosition(address maker_, uint256 positionId_, uint256 addedMargin_, uint256 addedLeverage_)
        private
        verifyLeverage(addedLeverage_)
    {
        Position storage position = positions[positionId_];
        ...
    }
```

Listing 3.6: TradePair::extendPosition()

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

Status This issue has been fixed in the following commit: `99f7a7e`.

3.7 Potential Protocol Risk from Low-Liquidity Assets

- ID: PVE-007
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: Unlimited Leverage Protocol
- Category: Security Features [8]
- CWE subcategory: CWE-654 [6]

Description

With the occurrence of the `Mango Market Incident` on `Solana`, the risk of liquidity attacks on leverage platforms attracts much attention from the DeFi community. In the following, we give one possible attack vector to illustrate the risk. The malicious actor temporarily raises up the collateral value, and then takes massive loans from the `Mango` treasury.

To elaborate, the malicious actor firstly supplies `$5M` stablecoins as collateral (Step I), secondly buys `$483M` short `MNGO` perp (Step II) and sells it to another self-funded `accountB` at the price of 0.0382. By manipulating the `MNGO` price to 0.91 (Step III), `accountB` finally is in the profit of `483 m * ($0.91 - $0.0382)= $423m` and leaves the `Mango` market with bad debt. Tweet [1] shows more details.

Recommendation Remove the low-liquidity assets from `Unlimited Leverage` to avoid the above risk of market manipulation.

Status The issue has been confirmed by the team. The team clarifies that they are aware of the risk of external price manipulation and implemented following measures to mitigate the risk. First and foremost it uses three different liquidity pools with different risk levels (blue chip, altcoins, degen coins). So liquidity providers can choose the bluechip pool when they want to limit their risk in general. Second, each `TradePair` has a maximum volume, that will be set lower than the current and a realistic volatility-affected liquidity of the underlying asset. This prevents positions to be higher than the amount of a token needed for price manipulation. Further, `Unlimited Leverage` limits the payout (maximum profit) of one position to a certain percentage of the liquidity pools, which prevents single trades from draining the liquidity pool. In addition to that, `Unlimited Leverage` uses a min/max price pair for buy/sell actions to implement a price slippage and deducts open and close position fees to decrease arbitrage opportunities in general.

4 Conclusion

In this audit, we have analyzed the `Unlimited Leverage` design and implementation. `Unlimited Leverage` is a synthetic cryptocurrency trading platform, where user can trade across a range of cryptocurrencies and blockchain networks. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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] Joshua Lim. Mongo Market Attack Analysis. https://twitter.com/joshua_j_lim/status/1579987648546246658?s=21&t=naoumx_P6NGLinnucYCbYQ.

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

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

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[6] MITRE. CWE-654: Reliance on a Single Factor in a Security Decision. https://cwe.mitre.org/data/definitions/654.html.

[7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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