SMART CONTRACT AUDIT REPORT

for

DeltaPrimeLabs

Prepared By: Xiaomi Huang

PeckShield

December 1, 2022

Document Properties

Client	DeltaPrimeLabs
Title	Smart Contract Audit Report
Target	DeltaPrimeLabs
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	December 1, 2022	Jing Wang	Final Release
1.0-rc	November 27, 2022	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 DeltaPrimeLabs 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 DeltaPrimeLabs

DeltaPrimeLabs is a lending platform on Avalanche that will allow under-collateral borrowing from pooled deposits. The key mechanism is to enable fund-lending not to a personal account, but a special purpose smart-contract. The contract automatically guards solvency and every activity needs to undergo a series of checks. The insolvency risk is further mitigated by a decentralized liquidation mechanism allowing anyone to forcibly repay part of the loan due to assets price movements caused by external factors. The basic information of DeltaPrimeLabs is as follows:

Table 1.1: Basic Information of DeltaPrimeLabs

Item	Description
Issuer	DeltaPrimeLabs
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 1, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Smart Loans assumes a trusted price oracle with timely market price feeds for supported assets. The oracle is not part of this audit.

- https://github.com/DeltaPrimeLabs/deltaprime-primeloans.git (c336b1c)

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

- https://github.com/DeltaPrimeLabs/deltaprime-primeloans.git (adb88f5)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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

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
	DeltaPrimeLabs 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

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) [5], 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 DeltaPrimeLabs 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	2	■ ■
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 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, and 2 low-severity vulnerabilities.

Table 2.1: Key DeltaPrimeLabs Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Revisited Logic Of Pool::transfer()	Business Logic	Fixed
PVE-002	Critical	Missing remainsSolvent() Check In unwrapAndWithdraw()	Business Logic	Fixed
PVE-003	Low	Missing nativeToken Counting In getThresholdWeightedValue()	Business Logic	Fixed
PVE-004	Low	Missing Assets Adding When removeLiquidity()	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Revisited Logic Of Pool::transfer()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: Pool
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The DeltaPrimeLabs protocol has a Pool contract which is designed to allow users to deposit and borrow asset from a dedicated user account. Depositors are rewarded with the interest rates collected from borrowers. What's more, deposits and debts are tokenized for accounting purposes. To elaborate, we show below the related routines.

```
    function deposit(uint256 _amount) public virtual nonReentrant {
      require(_amount>0, "Deposit amount must be > 0");
      _accumulateDepositInterest(msg.sender);

      _transferToPool(msg.sender, _amount);

      _mint(msg.sender, _amount);
      _deposited[address(this)] += _amount;
      _updateRates();

      if (address(poolRewarder) != address(0)) {
          poolRewarder.stakeFor(_amount, msg.sender);
      }

      emit Deposit(msg.sender, _amount, block.timestamp);
  }

  function withdraw(uint256 _amount) external nonReentrant {
      require(IERC20(tokenAddress).balanceOf(address(this)) >= _amount, "Not enough
          funds in the pool");

```

```
        _accumulateDepositInterest(msg.sender);

        _burn(msg.sender, _amount);

        _transferFromPool(msg.sender, _amount);

        _updateRates();

        if (address(poolRewarder) != address(0)) {
            poolRewarder.withdrawFor(_amount, msg.sender);
        }

        emit Withdrawal(msg.sender, _amount, block.timestamp);
    }

    function transfer(address recipient, uint256 amount) external override returns (bool)
        {
      require(recipient != address(0), "ERC20: cannot transfer to the zero address");
      require(recipient != address(this), "ERC20: cannot transfer to the pool address");

      _accumulateDepositInterest(msg.sender);

      require(_deposited[msg.sender] >= amount, "ERC20: transfer amount exceeds balance");

      // (this is verified in "require" above)
      unchecked {
          _deposited[msg.sender] -= amount;
      }

      _accumulateDepositInterest(recipient);
      _deposited[recipient] += amount;

      emit Transfer(msg.sender, recipient, amount);

      return true;
}
```

Listing 3.1: `Pool::deposit()`

We notice in the `deposit()` routine, there are rewards staked for the `msg.sender` and these rewards are withdrawn when depositors exit by calling `withdraw()`. However, the rewards handling is missing when depositors transfer their tokens via the `transfer()` routine. In this case, the user who received the pool tokens will not be able to withdraw tokens from the pool as the rewards are not withdrawn.

Recommendation Add necessary handling of rewards in the `transfer()` routine.

Status The issue has been fixed by this commit: `adb88f5`.

3.2 Missing remainsSolvent() Check In unwrapAndWithdraw()

- ID: PVE-002
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `SmartLoanWrappedNativeTokenFacet`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `SmartLoanWrappedNativeTokenFacet` contract is part of the logic of `SmartLoan`, a contract deployed on behalf of borrower. `SmartLoan` is responsible for custody of the funds and borrowed tokens. Specifically, the `SmartLoanWrappedNativeTokenFacet` contract is handling basic operations with wrapped native tokens (like wrapped `ETH`). To elaborate, we show below the `unwrapAndWithdraw()` routine in this contract.

```
function unwrapAndWithdraw(uint256 _amount) onlyOwner public payable virtual {
  IWrappedNativeToken wrapped = IWrappedNativeToken(DeploymentConstants.getNativeToken
      ());
  require(wrapped.balanceOf(address(this)) >= _amount, "Not enough native token to
      unwrap and withdraw");

  wrapped.withdraw(_amount);

  payable(msg.sender).safeTransferETH(_amount);

  emit UnwrapAndWithdraw(msg.sender, msg.value, block.timestamp);
}
```

Listing 3.2: `SmartLoanWrappedNativeTokenFacet::unwrapAndWithdraw()`

When examining the logic of the `unwrapAndWithdraw()` routine, we notice the `remainsSolvent()` modifier is missing, which is used for checking whether a borrower is allowed to borrow at the `SmartLoan` side (Note there is no checking on the pool side which makes this part of logic critical). A bad borrower can use the `wrappedNativeToken` as collateral to borrow `wrappedNativeToken` again to drain all `wrappedNativeToken` from the pool by creating as many as `smartLoan` contracts needed.

```
function isSolvent() public view returns (bool) {
  return getHealthRatio() >= 1e18;
}

function getHealthRatio() public view virtual returns (uint256) {
  uint256 debt = getDebt();
  uint256 thresholdWeightedValue = getThresholdWeightedValue();

  if (debt == 0) {
      return type(uint256).max;
```

```
    } else {
        return thresholdWeightedValue * 1e18 / debt;
    }
}
```

Listing 3.3: `SolvencyFacet::isSolvent()`

Recommendation Add the `remainsSolvent()` modifier to make sure the account is solvent when initiating a withdrawal.

Status The issue has been fixed by this commit: `6468dc8`.

3.3 Missing nativeToken Counting In getThresholdWeightedValue()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `SolvencyFacet`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.2, the `SmartLoan` contract performs solvency checks via `isSolvent()` to check if an account is solvent. Basically, it validates the health ratio (the ratio between borrowing power and total value minus debt) is above the safe level. In order to get the total value of all the assets in the `SmartLoan` contract, the `getThresholdWeightedValue()` is introduced and below is the related code snippet.

```
    function getThresholdWeightedValue() public view virtual returns (uint256) {
        bytes32[] memory assets = DeploymentConstants.getAllOwnedAssets();
        uint256[] memory prices = getOracleNumericValuesFromTxMsg(assets);
        uint256 nativeTokenPrice = getOracleNumericValueFromTxMsg(DeploymentConstants.
            getNativeTokenSymbol());
        TokenManager tokenManager = DeploymentConstants.getTokenManager();

        uint256 weightedValueOfTokens;

        if (prices.length > 0) {
            for (uint256 i = 0; i < prices.length; i++) {
                require(prices[i] != 0, "Asset price returned from oracle is zero");

                IERC20Metadata token = IERC20Metadata(tokenManager.getAssetAddress(
                    assets[i], true));

```

```
                weightedValueOfTokens = weightedValueOfTokens + (prices[i] * 10 ** 10 * token.balanceOf(address(this)) * tokenManager.maxTokenLeverage(address(token)) / (10 ** token.decimals() * 1e18));
            }
        }

        IStakingPositions.StakedPosition[] storage positions = DiamondStorageLib.stakedPositions();

        uint256 weightedValueOfStaked;

        for (uint256 i; i < positions.length; i++) {
            //TODO: fetch multiple prices to reduce cost
            uint256 price = getOracleNumericValueFromTxMsg(positions[i].symbol);
            require(price != 0, "Asset price returned from oracle is zero");

            (bool success, bytes memory result) = address(this).staticcall(abi.encodeWithSelector(positions[i].balanceSelector));

            if (success) {
                uint256 balance = abi.decode(result, (uint256));

                IERC20Metadata token = IERC20Metadata(DeploymentConstants.getTokenManager().getAssetAddress(positions[i].symbol, true));

                weightedValueOfStaked += price * 10 ** 10 * balance * tokenManager.maxTokenLeverage(positions[i].vault) / (10 ** token.decimals());
            }
        }

        return weightedValueOfTokens + weightedValueOfStaked;
    }
```

Listing 3.4: `SolvencyFacet::getThresholdWeightedValue()`

The `getThresholdWeightedValue()` routine counts all assets in USD including tokens as well as staking and LP positions with their threshold weight. However, the missing consideration of native token when doing the calculation might cause a lower value returned from `getThresholdWeightedValue()` than the contract owns.

Recommendation Add the consideration of native token in the `getThresholdWeightedValue()` routine.

Status The issue has been fixed by this commit: `877a783`.

3.4 Missing Assets Adding When removeLiquidity()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The UniswapV2DEXFacet contract provides several routines for users to swap tokens in the SmartLoan when needed. During the analysis of these functions, we notice the tokens received after removing liquidity might not been counted into the assets owned by the contracts. To elaborate, we show below the related code snippet of the StakingPool contract.

```
function removeLiquidity(bytes32 _assetA, bytes32 _assetB, uint liquidity, uint amountAMin, uint amountBMin) internal remainsSolvent {
    IERC20Metadata tokenA = getERC20TokenInstance(_assetA, true);
    IERC20Metadata tokenB = getERC20TokenInstance(_assetB, false);

    IAssetsExchange exchange = IAssetsExchange(getExchangeIntermediaryContract());

    address lpTokenAddress = exchange.getPair(address(tokenA), address(tokenB));

    lpTokenAddress.safeTransfer(getExchangeIntermediaryContract(), liquidity);

    (uint amountA, uint amountB) = exchange.removeLiquidity(address(tokenA), address(tokenB), liquidity, amountAMin, amountBMin);

    // Remove asset from ownedAssets if the asset balance is 0 after the LP
    if (IERC20Metadata(lpTokenAddress).balanceOf(address(this)) == 0) {
        (bytes32 token0, bytes32 token1) = _assetA < _assetB ? (_assetA, _assetB) : (_assetB, _assetA);
        bytes32 lpToken = stringToBytes32(string.concat(
                bytes32ToString(getProtocolID()),
                '_',
                bytes32ToString(token0),
                '_',
                bytes32ToString(token1)
            )
        );
        DiamondStorageLib.removeOwnedAsset(lpToken);
    }

    emit RemoveLiquidity(msg.sender, lpTokenAddress, _assetA, _assetB, liquidity, amountA, amountB, block.timestamp);
}
```

Listing 3.5: UniswapV2DEXFacet::removeLiquidity()

Specifically, if we examine the implementation of the `removeLiquidity()` routine, the `tokenA` and `tokenB`, which might already been removed from the assets list when user adding the liquidity, are not added back to the assets list after removing the liquidity. In this case, the calculation of the total value owned by the contract will be inaccurate. Note another routine `VectorFinanceFacet::unstakeToken()` shares the same issue.

Recommendation Add the token back to the assets list when removing the liquidity.

Status The issue has been fixed by this commit: `3aa1d6d`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `DeltaPrimeLabs` 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., parameter configuration). 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 `setBorrowersRegistry()` and related routines from the `Pool` contract. This function allows the `owner` account set the address of `borrowersRegistry` which determines whether an account could take assets out of the pool.

```
    function setBorrowersRegistry(IBorrowersRegistry borrowersRegistry_) external
        onlyOwner {
        require(AddressUpgradeable.isContract(address(borrowersRegistry_)), "Must be a
            contract");

        borrowersRegistry = borrowersRegistry_;
        emit BorrowersRegistryChanged(address(borrowersRegistry_), block.timestamp);
    }

    modifier canBorrow() {
        require(address(borrowersRegistry) != address(0), "Borrowers registry not
            configured");
        require(borrowersRegistry.canBorrow(msg.sender), "Only authorized accounts may
            borrow");
        require(totalSupply() != 0, "Cannot borrow from an empty pool");
```

```
        _;
        require((totalBorrowed() * 1e18) / totalSupply() <=
            MAX_POOL_UTILISATION_FOR_BORROWING, "The pool utilisation cannot be greater
            than 95%");
    }


    function borrow(uint256 _amount) public virtual canBorrow nonReentrant {
        require(IERC20(tokenAddress).balanceOf(address(this)) >= _amount, "Not enough
            funds in the pool");

        _accumulateBorrowingInterest(msg.sender);

        borrowed[msg.sender] += _amount;
        borrowed[address(this)] += _amount;

        _transferFromPool(msg.sender, _amount);

        _updateRates();

        emit Borrowing(msg.sender, _amount, block.timestamp);
    }
```

Listing 3.6: DeltaPrimeLabs::setBorrowersRegistry()

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 mitigated. The team clarifies they will deploy the contract with a 72h timelock contract. In the beginning it will be a multisig wallet from the team, with time handled to DAO structure.

4 Conclusion

In this audit, we have analyzed the DeltaPrimeLabs design and implementation. DeltaPrimeLabs is a lending platform on Avalanche that will allow under-collateral borrowing from pooled deposits. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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