SMART CONTRACT AUDIT REPORT

for

Rage Trade Vault

Prepared By: Xiaomi Huang

PeckShield

June 17, 2022

Document Properties

Client	Rage Trade Protocol
Title	Smart Contract Audit Report
Target	Rage Trade Vault
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 17, 2022	Xuxian Jiang	Final Release
1.0-rc	May 26, 2022	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Rage Trade protocol, we outline in this 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 contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About Rage Trade protocol

Rage Trade protocol aims to build the most liquid, composable omnichain ETH perp (powered by Uniswap v3). The core features include ETH perp with $10x$ leverage, omnichain recycled liquidity, and yield-generating 80-20 vaults. Each 80-20 vault accepts a different LP position as collateral (e.g., Curve Tri-Crypto) and recycles these LP shares to provide liquidity in Rage's ETH perp. And the goal of the 80-20 vault is to earn additional yield on the LP position while replicating the payoff of an ETH-USD LP in Uniswap v2. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Rage Trade Vault

Item	Description
Name	Rage Trade Protocol
Website	https://www.rage.trade/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 17, 2022

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

- https://github.com/RageTrade/vaults.git (ac1f6d2)

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

- https://github.com/RageTrade/vaults.git (d81fb62)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

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

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], 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 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 `Rage Trade` 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 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	3	■ ■ ■
Informational	0	
Total	4	

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 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Rage Trade Vault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inaccurate Fee Withdrawal in withdrawFees()	Business Logic	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Confirmed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-004	Low	Proper USDC Approvals To New swapRouter	Coding Practices	Fixed

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 Inaccurate Fee Withdrawal in withdrawFees()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CurveYieldStrategy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Rage Trade` protocol provides a `withdrawFees()` routine for the owner to withdraw the `CRV` fees which are accumulated from the `_harvestFees()` routine. While examining the fees amount to withdraw from the contract, we notice the calculation of the fees amount needs to be improved.

To elaborate, we show below the code snippets of the `withdrawFees()` routine and the `_harvestFees()` routine. As the name indicates, the `_harvestFees()` routine is designed to harvest the `CRV` rewards earned from the `Gauge` (line 166). After deducting the fees (line 165), the remaining `CRV` rewards are swapped to `USDT`, which will be added to the `TriCrypto` pool as liquidity. The final received `LP` tokens will be staked back to the `Gauge` to earn more rewards. If the swap from `CRV` to `USDT` fails due to slippage, it will count the pending `CRV` which were not successfully swapped to the `crvPendingToSwap` variable (line 197) to be used in next swap. As a result, the `CRV` tokens in current contract contains both the accumulated fees and the `crvPendingToSwap`. However, it comes to our attention that the `withdrawFees()` routine will transfer all the `CRV` tokens in the contract as fees to the owner (lines 132 – 133), which may contain the `crvPendingToSwap` also. The correct fees amount shall be `crvToken.balanceOf(address(this))- crvPendingToSwap`.

```
130    /// @notice withdraw accumulated CRV fees
131    function withdrawFees() external onlyOwner {
132        uint256 bal = crvToken.balanceOf(address(this));
133        crvToken.transfer(msg.sender, bal);
134        emit Logic.FeesWithdrawn(bal);
```

```
}
```

Listing 3.1: `CurveYieldStrategy::withdrawFees()`

```
function _harvestFees() internal override {
  uint256 claimable = gauge.claimable_reward(address(this), address(crvToken)) +
      crvPendingToSwap;

  if (claimable > crvHarvestThreshold) {
      uint256 afterDeductions = claimable - ((claimable * FEE) / MAX_BPS);
      gauge.claim_rewards(address(this));

      emit Logic.Harvested(claimable);

      bytes memory path = abi.encodePacked(
          address(crvToken),
          uint24(3000),
          address(weth),
          uint24(500),
          address(usdt)
      );

      try
          SwapManager.swapCrvToUsdtAndAddLiquidity(
              afterDeductions,
              crvSwapSlippageTolerance,
              crvOracle,
              path,
              uniV3Router,
              triCryptoPool
          )
      {
          // stake CRV if swap is successful
          _stake(asset.balanceOf(address(this)));
          // set pending CRV to 0
          crvPendingToSwap = 0;
      } catch Error(string memory reason) {
          // if swap is failed due to slippage, it should not stop executing rebalance
          // uniswap router returns 'Too little received' in case of minOut is not
              matched
          if (keccak256(abi.encodePacked(reason)) == keccak256('Too little received'))
               {
              // account for pending CRV which were not swapped, to be used in next
                  swap
              crvPendingToSwap += claimable;
              // emit event with current slippage value
              emit Logic.CrvSwapFailedDueToSlippage(crvSwapSlippageTolerance);
          }
          // if external call fails due to any other reason, revert with same
          else revert CYS_EXTERAL_CALL_FAILED(reason);
      }
  }
```

```
}
```

Listing 3.2: `CurveYieldStrategy::_harvestFees()`

Recommendation Revise the above `withdrawFees()` logic to compute the right fees amount to withdraw from the contract.

Status This issue has been fixed by this commit: `ae6fd21`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseVault`, `CurveYieldStrategy`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

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 `approve()` 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
```

```
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.3: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

More importantly, the `approve()` function of some token may return false while not revert on failure. Accordingly, the call to `approve()` is expected to check the return value. If it returns false, the call to `approve()` shall be failed.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transfer()/transferFrom()` as well, i.e., `safeTransfer()/safeTransferFrom()`.

In the following, we show the `CurveYieldStrategy::grantAllowances()` routine and the `BaseVault::_grantBaseAllowances()` routine. If the `approve()` of the given token (`rageCollateralToken/rageSettlementToken/asset`, etc.) does not revert on failure, the unsafe version of `approve()` need to check the return value while not assuming it will revert internally.

```
    function grantAllowances() public override onlyOwner {
        _grantBaseAllowances();

        asset.approve(address(gauge), type(uint256).max);
        asset.approve(address(triCryptoPool), type(uint256).max);

        /// @dev USDT requires allowance set to 0 before re-approving
        usdc.approve(address(uniV3Router), 0);
        usdt.approve(address(uniV3Router), 0);
        usdt.approve(address(triCryptoPool), 0);

        usdc.approve(address(uniV3Router), type(uint256).max);
        usdt.approve(address(uniV3Router), type(uint256).max);
        usdt.approve(address(triCryptoPool), type(uint256).max);

        crvToken.approve(address(uniV3Router), type(uint256).max);
    }
```

Listing 3.4: CurveYieldStrategy::grantAllowances()

```
/// @notice grants allowances for base vault
function _grantBaseAllowances() internal {
    rageCollateralToken.approve(address(rageClearingHouse), type(uint256).max);
    rageSettlementToken.approve(address(rageClearingHouse), type(uint256).max);
```

```
}
```

Listing 3.5: BaseVault :: _grantBaseAllowances()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status This finding has been acknowledged by the team. And the team clarifies that the protocol only interacts with the known set of `ERC20` tokens (`usdc`, `usdt`, `crv`, `triCrypto`) and has double approvals (first approve to 0 and then to the actual amount) wherever required.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Rage Trade` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., set the fee rate and withdraw fees). In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
/// @notice changes the fee value for CRV yield generated
/// @param bps new fee value (less than MAX_BPS)
function changeFee(uint256 bps) external onlyOwner {
    if (bps > MAX_BPS) revert CYS_INVALID_FEES();
    FEE = bps;
    emit Logic.FeesUpdated(bps);
}

/// @notice withdraw accumulated CRV fees
function withdrawFees() external onlyOwner {
    uint256 bal = crvToken.balanceOf(address(this));
    crvToken.transfer(msg.sender, bal);
    emit Logic.FeesWithdrawn(bal);
}
```

Listing 3.6: Example Privileged Operations in `CurveYieldStrategy.sol`

```
function updateSwapRouter(address newRouter) external onlyOwner {
    if (newRouter == address(0)) revert ZeroValue();
    swapRouter = ISwapRouter(newRouter);
```

```
    }

    function updateEthOracle(address newOracle) external onlyOwner {
        if (newOracle == address(0)) revert ZeroValue();
        ethOracle = AggregatorV3Interface(newOracle);
    }
```

Listing 3.7: Example Privileged Operations in `VaultPeriphery.sol`

There are still other privileged routines not listed here. We point out that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Making the above privileges explicit among protocol users.

Status This finding has been acknowledged by the team. And the team clarifies that the protocol will be deployed as a guarded launch with a lower max deposit cap and owner as multi-sig and will be progressively decentralized to change owner to governance.

3.4 Proper USDC Approvals To New swapRouter

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VaultPeriphery`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

Description

The `Rage Trade` protocol provides a `VaultPeriphery` contract which is a periphery of the `Vault`. The `VaultPeriphery` contract provides a series of convenient interfaces for users to interact with the `Vault`. Specially, it provides a `depositUsdc()` routine which accepts `USDC` as input and provides shares back to user.

To elaborate, we show below the code snippet of the `depositUsdc()` routine, which is designed to accept `USDC` from the user, swap the `USDC` to `USDT` via the `swapRouter`, add the `USDT` to the `TriCrypto` pool as liquidity, and then deposit the received `LPs` to the `Vault` at last. Before the routine is invoked, this contract shall grant enough `USDC` allowance to the `swapRouter`, which then could transfer the `USDC` to the router on behalf of this contract.

```
function depositUsdc(uint256 amount) external returns (uint256 sharesMinted) {
  if (amount == 0) revert ZeroValue();
```

```
    usdc.transferFrom(msg.sender, address(this), amount);

    bytes memory path = abi.encodePacked(usdc, uint24(500), usdt);

    ISwapRouter.ExactInputParams memory params = ISwapRouter.ExactInputParams({
        path: path,
        amountIn: amount,
        amountOutMinimum: 0,
        recipient: address(this),
        deadline: block.timestamp
    });

    uint256 usdtOut = swapRouter.exactInput(params);

    uint256 beforeSwapLpPrice = lpOracle.lp_price();

    stableSwap.add_liquidity([usdtOut, 0, 0], 0);

    uint256 balance = lpToken.balanceOf(address(this));

    /// @dev checks combined slippage of uni v3 swap and add liquidity
    if (balance.mulDiv(beforeSwapLpPrice, 10**18) < (amount * (MAX_BPS - MAX_TOLERANCE) * 10**12) / MAX_BPS) {
        revert SlippageToleranceBreached();
    }

    sharesMinted = vault.deposit(balance, msg.sender);
    emit DepositPeriphery(msg.sender, address(usdc), amount, balance, sharesMinted);
}
```

Listing 3.8: VaultPeriphery :: depositUsdc()

However, it comes to our attention that the `swapRouter` could be updated by the owner via below `updateSwapRouter()` routine, it doesn't grant proper `USDC` allowance to the new `swapRouter`. As a result, the `depositUsdc()` may revert because of no `USDC` allowance. Based on this, it is suggested to grant `type(uint256).max` `USDC` allowance to the new `swapRouter` in the `updateSwapRouter()` routine. In the meantime, we can reset the previous allowance to be 0 for the old `swapRouter`.

```
function updateSwapRouter(address newRouter) external onlyOwner {
  if (newRouter == address(0)) revert ZeroValue();
  swapRouter = ISwapRouter(newRouter);
}
```

Listing 3.9: VaultPeriphery :: updateSwapRouter()

Recommendation Revisit the above `updateSwapRouter()` routine to grant enough `USDC` allowance to the new `swapRouter`.

Status The issue has been fixed by this commit: `e44cdef`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Rage Trade` protocol, which aims to build the most liquid, composable `omnichain` `ETH` perp (powered by `Uniswap v3`). The core features include `ETH` perp with $10x$ leverage, `omnichain` recycled liquidity, and yield-generating `80-20` vaults. The current code base is well organized and 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] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

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

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

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

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

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

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