



SMART CONTRACT AUDIT REPORT

for

Hadouken Lending Protocol



Prepared By: Xiaomi Huang

PeckShield
October 21, 2022

Document Properties

Client	Hadouken Finance
Title	Smart Contract Audit Report
Target	Hadouken Lending Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 21, 2022	Luck Hu	Final Release
1.0-rc	October 14, 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	4
1.1	About Hadouken Lending Protocol	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Suggested Use of SafeMath in validateDeposit()	11
3.2	Incompatibility with Deflationary/Rebasing Tokens	13
3.3	Flashloan-Lowered StableBorrowRate for Mode-Switching Users	15
3.4	Trust Issue of Admin Keys	17
4	Conclusion	19
	References	20

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Hadouken` lending protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Hadouken Lending Protocol

`Hadouken` is a community-driven decentralized finance (DeFi) protocol with a suite of products: trading, portfolio management, lending and borrowing built on `Nervos Network`'s `Godwoken` layer-2 blockchain. The audited `Hadouken` lending protocol is built on top of `Aave v2`, which is one of the most successful and proven protocol to bring the best in class DeFi products to the `Nervos` ecosystem. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Hadouken Lending Protocol

Item	Description
Name	Hadouken Finance
Website	https://hadouken.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 21, 2022

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

- <https://github.com/hadouken-project/lending-contracts.git> (39b7425)

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

- <https://github.com/hadouken-project/lending-contracts.git> (d21b5be)

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 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
	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

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. 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 `Hadouken` lending smart contracts. 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	0	
High	0	
Medium	2	
Low	2	
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 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Hadouken Lending Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Suggested Use of SafeMath in validateDeposit()	Coding Practices	Fixed
PVE-002	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Mitigated
PVE-003	Low	Flashloan-Lowered StableBorrowRate for Mode-Switching Users	Time and State	Mitigated
PVE-004	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 Suggested Use of SafeMath in validateDeposit()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: LendingPool
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with `uint256` operands. While analyzing the LendingPool implementation, we observe it can be improved by taking advantage of the improved security from SafeMath.

In the computation of `totalDepositBalance + amount` (line 48) in the `ValidationLogic::validateDeposit()` routine, the addition of `totalDepositBalance` to `amount` is not guarded against possible overflow. As a result, if the computation overflows, a new deposit is allowed even when the deposit capability is exceeded.

```

41     function validateDeposit(DataTypes.ReserveData storage reserve, uint256 amount,
42         uint256 totalDepositBalance, uint256 depositCap) external view {
43         (bool isActive, bool isFrozen, , ) = reserve.configuration.getFlags();
44
45         require(amount != 0, Errors.VL_INVALID_AMOUNT);
46         require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
47         require(!isFrozen, Errors.VL_RESERVE_FROZEN);
48         if (depositCap != 0) {
49             require(depositCap >= totalDepositBalance + amount, Errors.DEPOSIT_CAP_REACHED);
50         }
51     }

```

Listing 3.1: The `ValidationLogic::validateDeposit()`

What's more, it shares the same issue in the `LendingPool::_executeBorrow()` routine, where the computation of `totalStableDebtTokens + totalVariableDebtTokens + vars.amount` (line 930) may overflow without the `SafeMath` protection. As a result, a new borrow request may be permitted even when the borrow capability is exceeded.

```

891     function _executeBorrow(ExecuteBorrowParams memory vars) internal {
892         DataTypes.ReserveData storage reserve = _reserves[vars.asset];
893         DataTypes.UserConfigurationMap storage userConfig = _usersConfig[vars.onBehalfOf
            ];

895         address oracle = _addressesProvider.getPriceOracle();

897         uint256 amountInETH = IPriceOracleGetter(oracle).getAssetPrice(vars.asset).mul(
            vars.amount).div(
898             10 ** reserve.configuration.getDecimals()
899         );

901         reserve.updateState();

903         ValidationLogic.validateBorrow(
904             vars.asset,
905             reserve,
906             vars.onBehalfOf,
907             vars.amount,
908             amountInETH,
909             vars.interestRateMode,
910             _maxStableRateBorrowSizePercent,
911             _reserves,
912             userConfig,
913             _reservesList,
914             _reservesCount,
915             oracle
916         );

918         uint256 borrowCap = reserve.configuration.getBorrowCap() * 10 ** reserve.
            configuration.getDecimals();

920         if (borrowCap != 0) {
921             uint256 totalVariableDebtTokens = IVariableDebtToken(reserve.
                variableDebtTokenAddress).scaledTotalSupply().rayMul(reserve.
                variableBorrowIndex);

923             (
924                 ,
925                 uint256 totalStableDebtTokens,
926                 ,

928             ) = IStableDebtToken(reserve.stableDebtTokenAddress).getSupplyData();

930             uint256 totalDebt = totalStableDebtTokens + totalVariableDebtTokens + vars.
                amount;

```

```

932         require(totalDebt <= borrowCap, Errors.BORROW_CAP_REACHED);
933     }
934     ...
935 }

```

Listing 3.2: The `LendingPool::_executeBorrow()`

Recommendation Make use of `SafeMath` in the above calculations to better mitigate possible overflows.

Status The issue has been fixed by this commit: [83523f1](#).

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LendingPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In Hadouken Lending Protocol, the `LendingPool` contract is designed to be the main entry point for interaction with users. In particular, one entry routine, i.e., `deposit()`, accepts asset transfer-in and mints the corresponding `aToken` to represent the user deposit. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of Hadouken Lending Protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```

107     function deposit(
108         address asset,
109         uint256 amount,
110         address onBehalfOf,
111         uint16 referralCode
112     ) external override whenNotPaused {
113         DataTypes.ReserveData storage reserve = _reserves[asset];
114
115         address aToken = reserve.aTokenAddress;
116
117         uint256 depositCap = reserve.configuration.getDepositCap() * 10 ** reserve.configuration.getDecimals();
118
119         reserve.updateState();

```

```

121     uint256 totalDepositBalance = IAToken(aToken).scaledTotalSupply().rayMul(reserve.
        liquidityIndex);
122     ValidationLogic.validateDeposit(reserve, amount, totalDepositBalance, depositCap);

124     reserve.updateInterestRates(asset, aToken, amount, 0);

126     IERC20(asset).safeTransferFrom(msg.sender, aToken, amount);

128     bool isFirstDeposit = IAToken(aToken).mint(onBehalfOf, amount, reserve.
        liquidityIndex);

130     if (isFirstDeposit) {
131         _usersConfig[onBehalfOf].setUsingAsCollateral(reserve.id, true);
132         emit ReserveUsedAsCollateralEnabled(asset, onBehalfOf);
133     }

135     emit Deposit(asset, msg.sender, onBehalfOf, amount, referralCode);
136 }

```

Listing 3.3: LendingPool::deposit()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the `LendingPool` contract before and after the `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted to be the collateral tokens. In fact, Hadouken Lending Protocol is indeed in the position to effectively regulate the set of assets that can be used as collaterals. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDt.

Status This issue has been mitigated as the team confirm they don't plan to support deflationary/rebasing tokens.

3.3 Flashloan-Lowered StableBorrowRate for Mode-Switching Users

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: LendingPool
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [3]

Description

Another unique feature implemented in the `Hadouken` protocol is the support of both variable and stable borrow rates. The variable borrow rate follows closely the market dynamics and can be changed on each user interaction (either borrow, deposit, withdraw, repayment or liquidation). The stable borrow rate instead will be unaffected by these actions. However, implementing a fixed stable borrow rate model on top of a dynamic reserve pool is complicated. The protocol provides the rate-rebalancing support to work around dynamic changes in market conditions or increased cost of money within the pool.

In the following, we show the code snippet of `swapBorrowRateMode()` which allows users to swap between stable and variable borrow rate modes. It follows the same sequence of convention by firstly validating the inputs (Step I), secondly updating relevant reserve states (Step II), then switching the requested borrow rates (Step III), next calculating the latest interest rates (Step IV), and finally performing external interactions, if any (Section V).

```

305  /**
306   * @dev Allows a borrower to swap his debt between stable and variable mode, or vice
      versa
307   * @param asset The address of the underlying asset borrowed
308   * @param rateMode The rate mode that the user wants to swap to
309   */
310  function swapBorrowRateMode(address asset, uint256 rateMode) external override
      whenNotPaused {
311      DataTypes.ReserveData storage reserve = _reserves[asset];

313      (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(msg.sender,
          reserve);

315      DataTypes.InterestRateMode interestRateMode = DataTypes.InterestRateMode(rateMode);

317      ValidationLogic.validateSwapRateMode(

```

```

318     reserve ,
319     _usersConfig[msg.sender] ,
320     stableDebt ,
321     variableDebt ,
322     interestRateMode
323 );

325     reserve.updateState();

327     if (interestRateMode == DataTypes.InterestRateMode.STABLE) {
328         IStableDebtToken(reserve.stableDebtTokenAddress).burn(msg.sender, stableDebt);
329         IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
330             msg.sender ,
331             msg.sender ,
332             stableDebt ,
333             reserve.variableBorrowIndex
334         );
335     } else {
336         IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
337             msg.sender ,
338             variableDebt ,
339             reserve.variableBorrowIndex
340         );
341         IStableDebtToken(reserve.stableDebtTokenAddress).mint(
342             msg.sender ,
343             msg.sender ,
344             variableDebt ,
345             reserve.currentStableBorrowRate
346         );
347     }

349     reserve.updateInterestRates(asset, reserve.aTokenAddress, 0, 0);

351     emit Swap(asset, msg.sender, rateMode);
352 }

```

Listing 3.4: LendingPool.sol

Our analysis shows this `swapBorrowRateMode()` routine can be affected by a flashloan-assisted sandwiching attack such that the new stable borrow rate becomes the lowest possible. Note this attack is applicable when the borrow rate is switched from variable to stable rate. Specifically, to perform the attack, a malicious actor can first request a flashloan to deposit into the reserve pool so that the reserve's utilization rate is close to 0, then invoke `swapBorrowRateMode()` to perform the variable-to-stable rate switch and enjoy the lowest `currentStableBorrowRate` (thanks to the nearly 0 utilization rate in current reserve), and finally withdraw to return the flashloan. A similar approach can also be applied to bypass `maxStableLoanPercent` enforcement in `validateBorrow()`.

Recommendation Revise current execution logic of `swapBorrowRateMode()` to defensively detect sudden changes to a reserve utilization and block malicious attempts.

Status This issue has been mitigated as the team confirm they will disable stable borrowing before the pool liquidity becomes big enough.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Hadouken` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., `set price oracle`). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `HadoukenOracle` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `HadoukenOracle` allow for the `owner` to set the price oracle and the fallback price oracle, both of which are used to provide the prices for the supported assets.

```

50  /// @notice Sets the oracle
51  /// - Callable only by the Hadouken governance
52  /// @param oracle The address of the oracle provider
53  function setOracle(address oracle) external onlyOwner {
54      _setOracle(oracle);
55  }
56
57  /// @notice Sets the fallbackOracle
58  /// - Callable only by the Hadouken governance
59  /// @param fallbackOracle The address of the fallbackOracle
60  function setFallbackOracle(address fallbackOracle) external onlyOwner {
61      _setFallbackOracle(fallbackOracle);
62  }

```

Listing 3.5: Example Privileged Operations in the `HadoukenOracle` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. 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 as the team confirmed they are using Gnosis Safe multi-sig wallet as the owner.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Hadouken` lending protocol. `Hadouken` is a community-driven decentralized finance (`Defi`) protocol with a suite of products: trading, portfolio management, lending and borrowing built on `Nervos Network`'s `Godwoken` Layer 2 blockchain. The audited `Hadouken` lending protocol is built on top of `Aave v2`, which is one of the most successful and proven protocol to bring the best in class `Defi` products to the `Nervos` ecosystem. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. <https://cwe.mitre.org/data/definitions/837.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>.

