SECURITY AUDIT REPORT

for

Zenlink Stable AMM & Swap Router

Prepared By: Xiaomi Huang

PeckShield

August 24, 2022

Document Properties

Client	Zenlink
Title	Security Audit Report
Target	Zenlink Stable AMM & Swap Router
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 24, 2022	Xiaotao Wu	Final Release
1.0-rc	August 1, 2022	Xiaotao Wu	Release Candidate #1

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 source code of the Zenlink Stable AMM and Swap Router features, 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 Zenlink

Zenlink is an underlying cross-chain DEX protocol based on Polkadot, and is committed to become the DEX composable hub of Polkadot. By accessing the ultimate, open and universal cross-chain DEX protocol based on Substrate, Zenlink enables all parachains to build DEX and achieve liquidity sharing in one click. The protocol includes Module, WASM, and EVM implementations, which are flexible and adaptable, allowing for customizable compositions and interoperability with different DeFi modules. The audited Zenlink Stable AMM and Zenlink Swap Router are pallets built on Substrate, which is simply understood as a Substrate version of Zenlink Hybrid AMM that combines standard AMM and stable AMM and is connected by Zenlink Swap Router to provide a better trading experience for DeFi users. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Zenlink Stable AMM & Swap Router

Item	Description
Name	Zenlink
Website	https://zenlink.pro/en/
Type	Polkadot
Platform	Rust
Audit Method	Whitebox
Latest Audit Report	August 24, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Note this audit only covers the `zenlink-stable-amm/src/lib.rs` and `zenlink-swap-router/src/lib.rs`.

- https://github.com/zenlinkpro/Zenlink-DEX-Module/tree/audit (3445d21)

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

- https://github.com/zenlinkpro/Zenlink-DEX-Module/tree/audit (c147a07)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], 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

Table 1.3: The Full List of Check Items

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

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Zenlink Stable AMM and Swap Router` implementations. 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	1	■
Medium	1	■
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 1 high-severity vulnerability, 2 medium-severity vulnerability, and 1 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improved Implementation Logic for MetaPool	Business Logic	Fixed
PVE-002	Low	Revisited Logic in inner_remove_liquidity_imbalance()	Numeric Errors	Fixed
PVE-003	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Fixed
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 Improved Implementation Logic for MetaPool

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: zenlink-stable-amm/src/lib.rs
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Zenlink Stable AMM provides two kinds of stablecoin swap pools, i.e., standard StableSwap BasePool and MetaPool. These two kinds of pools were both originally designed by Curve Finance. The BasePool is an implementation of the StableSwap invariant for two or more tokens. While MetaPool is a pool where a stablecoin is paired against the LP token from another pool, i.e., BasePool. MetaPool allows a single coin to be pooled with all the coins in another (base) pool without diluting its liquidity.

While examining the Zenlink Stable AMM feature, we notice that the BasePool and MetaPool use the same implementation logic. The concern for this design is that the MetaPool is basically a stablecoin pool maintaining prices for stablecoins, while the LP token of a BasePool is not a stablecoin. In fact, the price of a LP token of a BasePool steadily increases with the accumulation of fees charged by the BasePool. Therefore, there is a need for MetaPool to scale up or scale down the amount of LP token being exchanged with the LP token's price and the price of a LP token of a BasePool can be obtained by invoking the calculate_virtual_price() function of the BasePool.

In the following, we use the calculate_swap_amount() routine as an example and show the related code snippet. If the pool is a MetaPool, the current implementation logic ignores the impact of the virtual price of the LP token, which means the value of the LP token is underestimated. In other words, less stablecoin could be swapped out (line 1501) or more LP tokens could be swapped out (lines 1504-1507). Note a similar issue also exists in other functions for the MetaPool.

```
pub fn calculate_swap_amount(
  pool: &Pool<T::CurrencyId, T::AccountId, BoundedVec<u8, T::PoolCurrencySymbolLimit>>,
  i: usize,
  j: usize,
  in_balance: Balance,
) -> Option<Balance> {
  let n_currencies = pool.currency_ids.len();
  if i >= n_currencies || j >= n_currencies {
    return None;
  }

  let fee_denominator = FEE_DENOMINATOR;

  let normalized_balances = Self::xp(&pool.balances, &pool.token_multipliers)?;
  let new_in_balance = normalized_balances[i].checked_add(in_balance.checked_mul(pool.token_multipliers[i])?)?;

  let out_balance = Self::get_y(pool, i, j, new_in_balance, &normalized_balances)?;
  let mut out_amount = normalized_balances[j]
    .checked_sub(out_balance)?
    .checked_sub(One::one())?
    .checked_div(pool.token_multipliers[j])?;

  let fee = U256::from(out_amount)
    .checked_mul(U256::from(pool.fee))?
    .checked_div(U256::from(fee_denominator))
    .and_then(|n| TryInto::<Balance>::try_into(n).ok())?;

  out_amount = out_amount.checked_sub(fee)?;

  Some(out_amount)
}
```

Listing 3.1: `zenlink-stable-amm/src/lib.rs`

Recommendation Take the the virtual price of the `LP` token into consideration for the `MetaPool` implementation logic.

Status This issue has been fixed in the following commit: `d656a11`.

3.2 Revisited Logic in inner_remove_liquidity_imbalance()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `zenlink-stable-amm/src/lib.rs`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

The `Zenlink Stable AMM` provides a public `remove_liquidity_imbalance()` function for users to remove liquidity from a pool. While examining the current logic, we notice one possible precision loss source that stems from the mixed uses when both multiplication and division are involved.

In the following, we show the related code snippet. The `remove_liquidity_imbalance()` function is used to remove the liquidity in a way that will introduce the imbalance to the current pool. Because of possible rounding issues that may be caused from internal computation (lines 1608-1612), there is a need to round up the `burn_amount` calculation by adding 1 to it (line 1236 and line 1240). By doing so, we can better protect the interest of the pool's remaining liquidity providers.

```
pub fn remove_liquidity_imbalance(
      origin: OriginFor<T>,
      pool_id: T::PoolId,
      amounts: Vec<Balance>,
      max_burn_amount: Balance,
      to: T::AccountId,
      deadline: T::BlockNumber,
  ) -> DispatchResult {
      let who = ensure_signed(origin)?;

      let now = frame_system::Pallet::<T>::block_number();
      ensure!(deadline > now, Error::<T>::Deadline);

      Self::inner_remove_liquidity_imbalance(&who, pool_id, &amounts, max_burn_amount, &to)?;

      Ok(())
  }
```

Listing 3.2: `zenlink-stable-amm/src/lib.rs::remove_liquidity_imbalance()`

```
1219 pub fn inner_remove_liquidity_imbalance(
1220   who: &T::AccountId,
1221   pool_id: T::PoolId,
1222   amounts: &[Balance],
1223   max_burn_amount: Balance,
1224   to: &T::AccountId,
1225 ) -> DispatchResult {
```

```
    Pools::<T>::try_mutate_exists(pool_id, |optioned_pool| -> DispatchResult {
      let pool = optioned_pool.as_mut().ok_or(Error::<T>::InvalidPoolId)?;
      let total_supply = T::MultiCurrency::total_issuance(pool.lp_currency_id);

      ensure!(total_supply > Zero::zero(), Error::<T>::InsufficientLpReserve);
      ensure!(amounts.len() == pool.currency_ids.len(), Error::<T>::MismatchParameter);

      let (burn_amount, fees, d1) = Self::calculate_remove_liquidity_imbalance(pool,
          amounts, total_supply)
        .ok_or(Error::<T>::Arithmetic)?;
      ensure!(
        burn_amount > Zero::zero() && burn_amount <= max_burn_amount,
        Error::<T>::AmountSlippage
      );

      T::MultiCurrency::withdraw(pool.lp_currency_id, who, burn_amount)?;

      ...
    })
  }
```

Listing 3.3: `zenlink-stable-amm/src/lib.rs::inner_remove_liquidity_imbalance()`

```
  fn calculate_remove_liquidity_imbalance(
    pool: &mut Pool<T::CurrencyId, T::AccountId, BoundedVec<u8, T::
        PoolCurrencySymbolLimit>>,
    amounts: &[Balance],
    total_supply: Balance,
  ) -> Option<(Balance, Vec<Balance>, Balance)> {
    ...

    let d1 = Self::get_d(&Self::xp(&new_balances, &pool.token_multipliers)?, amp)?;
    let burn_amount = d0
      .checked_sub(U256::from(d1))?
      .checked_mul(U256::from(total_supply))?
      .checked_div(d0)
      .and_then(|n| TryInto::<Balance>::try_into(n).ok())?;

    Some((burn_amount, fees, d1))
  }
```

Listing 3.4: `zenlink-stable-amm/src/lib.rs::calculate_remove_liquidity_imbalance()`

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the following commit: `d656a11`.

3.3 Improved Sanity Checks Of System/Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `zenlink-stable-amm/src/lib.rs`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the `Zenlink Stable AMM` provides a public `remove_liquidity()` function for users to remove liquidity from an existing pool. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below its code snippet. We notice the current implementation does not specify any restriction on the input argument `lp_amount`. As a result, a user could remove liquidity with the input argument `lp_amount` set to 0, which is a waste of gas.

```
    pub fn remove_liquidity(
        origin: OriginFor<T>,
        poo_id: T::PoolId,
        lp_amount: Balance,
        min_amounts: Vec<Balance>,
        to: T::AccountId,
        deadline: T::BlockNumber,
    ) -> DispatchResult {
        let who = ensure_signed(origin)?;

        let now = frame_system::Pallet::<T>::block_number();
        ensure!(deadline > now, Error::<T>::Deadline);

        Self::inner_remove_liquidity(poo_id, &who, lp_amount, &min_amounts, &to)?;

        Ok(())
    }
```

Listing 3.5: `zenlink-stable-amm/src/lib.rs::remove_liquidity()`

```
  pub fn inner_remove_liquidity(
    pool_id: T::PoolId,
    who: &T::AccountId,
    lp_amount: Balance,
    min_amounts: &[Balance],
    to: &T::AccountId,
  ) -> DispatchResult {
    Pools::<T>::try_mutate_exists(pool_id, |optioned_pool| -> DispatchResult {
      let pool = optioned_pool.as_mut().ok_or(Error::<T>::InvalidPoolId)?;
      let lp_total_supply = T::MultiCurrency::total_issuance(pool.lp_currency_id);

```

```
        ensure!(lp_total_supply >= lp_amount, Error::<T>::InsufficientReserve);
        let currencies_length = pool.currency_ids.len();
        let min_amounts_length = min_amounts.len();
        ensure!(currencies_length == min_amounts_length, Error::<T>::MismatchParameter);

        ...
    })
}
```

Listing 3.6: `zenlink-stable-amm/src/lib.rs::remove_liquidity()`

Note a similar issue also exists in the `inner_remove_liquidity_one_currency()` routine of the same contract.

Recommendation Validate the input argument by ensuring `lp_amount > 0` in the above mentioned functions.

Status This issue has been fixed in the following commit: `d656a11`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `zenlink-stable-amm/src/lib.rs`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In `Zenlink Stable AMM`, there is a privileged account, i.e., `root`. This account plays a critical role in governing and regulating the system-wide operations (e.g., create `BasePool` and `MetaPool`, update `admin_fee_receiver`, set `fee/admin_fee`, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the `root` account.

```
    pub fn create_pool(
        origin: OriginFor<T>,
        currency_ids: Vec<T::CurrencyId>,
        currency_decimals: Vec<u32>,
        a: Number,
        fee: Number,
        admin_fee: Number,
        admin_fee_receiver: T::AccountId,
        lp_currency_symbol: Vec<u8>,
    ) -> DispatchResult {
        ensure_root(origin)?;
```

```
        ensure!(
            T::EnsurePoolAsset::validate_pooled_currency(&currency_ids),
            Error::<T>::InvalidPooledCurrency
        );

        ensure!(
            currency_ids.len() == currency_decimals.len(),
            Error::<T>::MismatchParameter
        );
        ensure!(a < MAX_A, Error::<T>::ExceedMaxA);
        ensure!(fee <= MAX_SWAP_FEE, Error::<T>::ExceedMaxFee);
        ensure!(admin_fee <= MAX_ADMIN_FEE, Error::<T>::ExceedMaxAdminFee);

        ...
    }

    pub fn update_fee_receiver(
        origin: OriginFor<T>,
        pool_id: T::PoolId,
        fee_receiver: <T::Lookup as StaticLookup>::Source,
    ) -> DispatchResult {
        ensure_root(origin)?;
        let admin_fee_receiver = T::Lookup::lookup(fee_receiver)?;
        Pools::<T>::try_mutate_exists(pool_id, |optioned_pool| -> DispatchResult {
            let pool = optioned_pool.as_mut().ok_or(Error::<T>::InvalidPoolId)?;
            pool.admin_fee_receiver = admin_fee_receiver.clone();

            Self::deposit_event(Event::UpdateAdminFeeReceiver {
                pool_id,
                admin_fee_receiver,
            });
            Ok(())
        })
    }

    pub fn set_fee(
        origin: OriginFor<T>,
        pool_id: T::PoolId,
        new_swap_fee: Number,
        new_admin_fee: Number,
    ) -> DispatchResult {
        ensure_root(origin)?;
        Pools::<T>::try_mutate_exists(pool_id, |optioned_pool| -> DispatchResult {
            let pool = optioned_pool.as_mut().ok_or(Error::<T>::InvalidPoolId)?;
            ensure!(new_swap_fee <= MAX_SWAP_FEE, Error::<T>::ExceedThreshold);
            ensure!(new_admin_fee <= MAX_ADMIN_FEE, Error::<T>::ExceedThreshold);

            pool.admin_fee = new_admin_fee;
            pool.fee = new_swap_fee;

            Self::deposit_event(Event::NewFee {
```

```
                pool_id,
                new_swap_fee,
                new_admin_fee,
            });
            Ok(())
        })
    }
```

Listing 3.7: zenlink-stable-amm/src/lib.rs

We understand the need of the privileged functions for proper `Stable AMM` operations, but at the same time the extra power to the `root` may also be a counter-party risk to the `Stable AMM` 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 Make the list of extra privileges granted to `root` explicit to `Zenlink Stable AMM` users.

Status This issue has been mitigated. The `Zenlink` team confirms that the way to get root privilege should via governance.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Zenlink Stable AMM and Swap Router` features. The audited `Zenlink Stable AMM` and `Zenlink Swap Router` are pallets built on `Substrate`, which is simply understood as a `Substrate` version of `Zenlink Hybrid AMM` that combines standard `AMM` and stable `AMM` and is connected by `Zenlink Swap Router` to provide a better trading experience for `DeFi` users. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.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 CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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