SMART CONTRACT AUDIT REPORT

for

Xenify Protocol

Prepared By: Xiaomi Huang

PeckShield

November 15, 2023

Document Properties

Client	Xenify
Title	Smart Contract Audit Report
Target	Xenify
Version	1.0
Author	Xuxian Jiang
Auditors	Jonathan Zhao, Xuxian Jiang
Reviewed by	Colin Zhong
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 15, 2023	Xuxian Jiang	Final Release
1.0-rc	November 5, 2023	Xuxian Jiang	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 smart contract source code of the Xenify 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 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 Xenify

Xenify stands as a cross-chain meta-aggregator of aggregators, pioneering a new era of Swap to Earn. The protocol integrates inventive tokenomics and advanced cross-chain functionality into a single, powerful package. By incorporating a unique, game theory-based incentive model that actively rewards engagement, Xenify is primed to instigate a seismic shift in the world of cross-chain swapping. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Xenify Protocol

Item	Description
Name	Xenify
Website	https://xenify.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 15, 2023

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

- https://github.com/xenify-io/xenify-contracts.git (8600ca1)

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

- https://github.com/xenify-io/xenify-contracts.git (199b7f8)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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) [7], 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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Xenify` implementation. 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	4	■■■■
Low	1	■
Informational	0	
Total	6	

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 that 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, 4 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Xenify Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect totalSupplyAtT()/balanceOfNFTAt() Logic in veXNF	Business Logic	Resolved
PVE-002	Medium	Incorrect Reward-Claiming Logic in veXNF::withdraw()	Business Logic	Resolved
PVE-003	Low	Lack of Timely XNF Supply Update in veXNF::merge()	Business Logic	Resolved
PVE-004	High	Possible Day Misalignment of decayEnd in veXNF	Coding Practices	Resolved
PVE-005	Medium	Incorrect Token Split Logic in veXNF	Business Logic	Resolved
PVE-006	Medium	Improved pendingNative() Logic in Auction	Business Logic	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Incorrect totalSupplyAtT()/balanceOfNFTAt() Logic in veXNF

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `veXNF`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In `Xenify`, the `veXNF` contract escrows the deposited protocol token `XNFs` in the form of an `ERC721 NFT`, which entitles the owner the voting power. The voting power has a weight that decays linearly over time. While examining current logic to query total voting power (or a user's voting power) at an earlier timestamp, we notice the implementation may return in accurate result.

In the following, we show below its implementation in `totalSupplyAtT()`. This routine has a rather straightforward logic in computing the total voting power at a specific past time using a given point as a reference. However, it comes to our attention that it implicitly assumes the given reference is adjacent and always occurs ahead of the timestamp being queried. This implicit assumption may not hold and current logic may return a false voting power when this implicit assumption is violated.

```
    function totalSupplyAtT(uint t)
        public
        view
        override
        returns (uint)
    {
        uint _epoch = epoch;
        Point memory last_point = pointHistory[_epoch];
        return _supply_at(last_point, t);
    }
```

Listing 3.1: `veXNF::totalSupplyAtT()`

```
    function _supply_at(
        Point memory point,
        uint t
    )
        internal
        view
        returns (uint)
    {
        Point memory last_point = point;
        uint t_i = (last_point.ts / _DAY) * _DAY;
        for (uint i; i < 61; ++i) {
            t_i += _DAY;
            int128 d_slope = 0;
            if (t_i > t) {
                t_i = t;
            } else {
                d_slope = slope_changes[t_i];
            }
            last_point.bias -= last_point.slope * int128(int256(t_i) - int256(last_point.ts));
            if (t_i == t) {
                break;
            }
            last_point.slope += d_slope;
            last_point.ts = t_i;
        }
        if (last_point.bias < 0) {
            last_point.bias = 0;
        }
        return uint(uint128(last_point.bias));
    }
```

Listing 3.2: `veXNF::_supply_at()`

Recommendation Revise the above routine to properly calculate the total voting power at a specific timestamp. Note the same issue also applies to the `balanceOfNFTAt()` routine.

Status The issue has been resolved in the following commit: `b8fd6f3` and `ae3f3b9`.

3.2 Incorrect Reward-Claiming Logic in veXNF::withdraw()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `veXNF`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, `Xenify` escrows the deposited protocol token `XNFs` in the form of an `ERC721 NFT`. While deposited tokens are withdrawn, the protocol will compute the reward amount and send back to the withdrawing user. Our analysis on the withdrawing logic indicates that current implementation needs to be revised so that the rewards are sent to the intended owner, not the caller.

Specifically, we show below the code snippet from the `withdraw()` routine. This routine basically allows the user to withdraw all tokens from an expired NFT lock. Note that this function may be called not by the owner of the expired NFTs. In this case, we need to properly send the rewards to the owner `_idToOwner[_tokenId])`, instead of current `msg.sender` (lines 689 – 690).

```
672     function withdraw(uint _tokenId)
673         external
674         override
675         nonReentrant
676     {
677         if (!_isApprovedOrOwner(msg.sender, _tokenId)) {
678             revert NotApprovedOrOwnerForWithdraw(msg.sender, _tokenId);
679         }
680         LockedBalance memory _locked = locked[_tokenId];
681         if (block.timestamp < _locked.end) {
682             revert LockNotExpiredYet(_locked.end);
683         }
684         uint value = uint(int256(_locked.amount));
685         locked[_tokenId] = LockedBalance(0,0,0,0);
686         uint supply_before = supply;
687         supply = supply_before - value;
688         _checkpoint(_tokenId, _locked, LockedBalance(0,0,0,0));
689         IERC20(xnf).safeTransfer(msg.sender, value);
690         IAuction(Auction).claimAllForUser(msg.sender);
691         _burn(_tokenId);
692         emit Burn(msg.sender, _tokenId);
693         emit Withdraw(msg.sender, _tokenId, value);
694         emit Supply(supply_before, supply_before - value);
695     }
```

Listing 3.3: `veXNF::withdraw()`

Recommendation Revise the above logic to properly return the withdrawn tokens and rewards.

Status The issue has been resolved in the following commit: 94f6d77.

3.3 Lack of Timely XNF Supply Update in veXNF::merge()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: veXNF
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The escrow contract in Xenify has an advanced feature in allowing the merging of multiple NFTs into a single new NFT. The merging is indicated with a separate deposit type, i.e., MERGE_TYPE. Our analysis on the merging logic indicates it does not properly update the backing supply state.

To elaborate, we show below the implementation of the merge() logic. While it indeed properly burns the given NFTs for merging and mints a new NFT. It does not adjust supply = supply - value to reflect they are now burned. Notice that when the internal handler _depositFor() is called, the supply will be added back with the burnt amount. As a result, we may see an inflated supply without the exact backing of XNFs.

```
    function merge(uint[] memory _from)
        external
        override
    {
        address owner = _checkOwner(_from);
        (uint256 maxPeriod) = _getMaxPeriod(_from);
        uint value;
        uint256 length = _from.length;
        IAuction(Auction).claimAllForUser(msg.sender);
        for (uint256 i; i < length; i++) {
            LockedBalance memory _locked = locked[_from[i]];
            value += uint(int256(_locked.amount));
            locked[_from[i]] = LockedBalance(0, 0, 0, 0);
            _checkpoint(_from[i], _locked, LockedBalance(0, 0, 0, 0));
            _burn(_from[i]);
            emit Burn(msg.sender, _from[i]);
        }
        uint unlock_time = block.timestamp + maxPeriod * _DAY;
        uint decayEnd = block.timestamp + maxPeriod * _DAY / 6;
        ++_tokenID;
        uint _tokenId = _tokenID;
        _mint(owner, _tokenId);
        emit Mint(msg.sender, _tokenId, value, unlock_time);
        locked[_tokenId].daysCount = maxPeriod;
```

```
        _depositFor(_tokenId, value, unlock_time, decayEnd, locked[_tokenId],
            DepositType.MERGE_TYPE);
    }
```

Listing 3.4: `veXNF::merge()`

Recommendation Revise the above logic to properly adjust the `supply` state.

Status The issue has been resolved in the following commit: `3d1f7b7`.

3.4 Possible Day Misalignment of decayEnd in veXNF

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `veXNF`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1109 [1]

Description

The escrow contract in `Xenify` is also enhanced with a new state `decayEnd` to have a fine-grained control on the timestamp when the voting power decay ends. The addition of this field has an implicit assumption on its alignment, i.e., it must be dividable by `DAY`. However, this implicit assumption is not properly enforced and may be violated in a number of occasions.

To elaborate, we show below an example occasion where a user may deposit tokens into a specific `NFT` lock. We notice the `decayEnd` state is computed as `decayEnd = block.timestamp + locked[_tokenId].daysCount * _DAY / 6` (line 583), which may not be aligned on the `DAY` boundary at all. A misaligned `decayEnd` may greatly affect the voting slope changes and undermines the voting power calculation.

```
564     function depositFor(
565         uint _tokenId,
566         uint _value
567     )
568         external
569         override
570         nonReentrant
571     {
572         if (!_isApprovedOrOwner(msg.sender, _tokenId)) {
573             revert NotApprovedOrOwner(msg.sender, _tokenId);
574         }
575         LockedBalance memory _locked = locked[_tokenId];
576         if (_value == 0) {
577             revert ZeroValueDeposit();
578         }
579         if (_locked.end <= block.timestamp) {
```

```
            revert LockExpired();
        }
        uint unlock_time = block.timestamp + locked[_tokenId].daysCount * _DAY;
        uint decayEnd = block.timestamp + locked[_tokenId].daysCount * _DAY / 6;
        _depositFor(_tokenId, _value, unlock_time, decayEnd, _locked, DepositType.
            DEPOSIT_FOR_TYPE);
    }
```

Listing 3.5: `veXNF::depositFor()`

Recommendation Revisit the `decayEnd` adjustment to ensure it is properly aligned. Note this issue affects a few routines, including `depositFor()`, `increaseUnlockTime()`, and `merge()`.

Status The issue has been resolved in the following commit: `c640dc4`.

3.5 Incorrect Token Split Logic in veXNF

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `veXNF`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

In addition to the `NFT`-merging feature, `Xenify` also supports the `NFT`-splitting. Our analysis on the `NFT`-splitting feature indicates that the new splitted `NFTs` need to carry over the previous `daysCount` state.

To elaborate, we show below the implementation of `split()`. The goal here is to split a single `NFT` into multiple new `NFTs` with specified amounts. While the amounts for the new splitted `NFTs` may be different, they should share the same `daysCount`, `unlock_time`, and `decayEnd`. Currently, it only ensures the same `unlock_time` and `decayEnd`, but not the same `daysCount`.

```
    function split(
        uint[] calldata amounts,
        uint _tokenId
    )
        external
        override
    {
        if (!_isApprovedOrOwner(msg.sender, _tokenId)) {
            revert NotApprovedOrOwnerForSplit(msg.sender, _tokenId);
        }
        address _to = _idToOwner[_tokenId];
        LockedBalance memory _locked = locked[_tokenId];
```

```
        uint value = uint(int256(_locked.amount));
        if (value == 0) {
            revert LockedAmountZero();
        }
        supply = supply - value;
        uint totalWeight;
        uint256 length = amounts.length;
        for (uint i; i < length; i++) {
            totalWeight += amounts[i];
        }
        if (totalWeight == 0) {
            revert WeightIsZero();
        }
        locked[_tokenId] = LockedBalance(0, 0, 0, 0);
        _checkpoint(_tokenId, _locked, LockedBalance(0, 0, 0, 0));
        IAuction(Auction).claimAllForUser(_idToOwner[_tokenId]);
        _burn(_tokenId);
        emit Burn(msg.sender, _tokenId);
        uint unlock_time = _locked.end;
        if (unlock_time <= block.timestamp) {
            revert LockExpired();
        }
        uint _value;
        for (uint j; j < length; j++) {
            ++_tokenID;
            _tokenId = _tokenID;
            _mint(_to, _tokenId);
            _value = value * amounts[j] / totalWeight;
            emit Mint(msg.sender, _tokenId, _value, unlock_time);
            _depositFor(_tokenId, _value, unlock_time, _locked.decayEnd, locked[_tokenId], DepositType.SPLIT_TYPE);
        }
    }
```

Listing 3.6: `veXNF::split()`

Recommendation Ensure the new splitted `NFTs` share the same `daysCount`, `unlock_time`, and `decayEnd`.

Status The issue has been resolved in the following commit: `2167810`.

3.6 Improved pendingNative() Logic in Auction

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Auction
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The Xenify protocol has a core Auction contract that allows for earning rewards by burning tokens. This contract has a key pendingNative() helper that is designed to compute the native reward for related users. While reviewing its logic, we notice the current implementation needs to be revisited.

For elaboration, we show below the implementation of this pendingNative() routine. As the name indicates, this function calculates pending native token rewards for a user based on their NFT ownership and recycling activities. However, it comes to our attention that this routine makes use of cycleAccNative, cycleAccExactNativeFromSwaps, and cycleAccNativeFromAuction to compute the initial portion of rewards. It also needs to take into account of cycleAccNativeFromNativeParticipants for the reward calculation.

```
    function pendingNative(address _user)
        public
        view
        override
        returns (uint256 _pendingNative)
    {
        User memory user = userInfo[_user];
        UserLastActivity storage userLastActivity = userLastActivityInfo[_user];
        uint256 cycle = getCurrentCycle();
        if (userLastActivity.lastUpdatedStats < cycle) {
            uint256 cycleEndTs;
            for (uint256 i = userLastActivity.lastUpdatedStats; i < cycle; i++) {
                cycleEndTs = i_initialTimestamp + i_periodDuration * (i + 1) - 1;
                if (cycleInfo[i].cycleAccNative + cycleInfo[i].
                    cycleAccExactNativeFromSwaps + cycleInfo[i].
                    cycleAccNativeFromAuction != 0) {
                    if (IVeXNF(veXNF).totalBalanceOfNFTAt(_user, cycleEndTs) != 0) {
                        _pendingNative += (cycleInfo[i].cycleAccNative + cycleInfo[i].
                            cycleAccExactNativeFromSwaps + cycleInfo[i].
                            cycleAccNativeFromAuction)
                            * IVeXNF(veXNF).totalBalanceOfNFTAt(_user, cycleEndTs) /
                                IVeXNF(veXNF).totalSupplyAtT(cycleEndTs);
                    }
                }
            }
        }...
```

```
        if (userLastActivity.lastUpdatedStats < cycle) {
            _pendingNative += user.pendingNative;
        } else {
            _pendingNative = user.pendingNative;
        }
    }
```

Listing 3.7: `Auction::pendingNative()`

Recommendation Revisit the above logic to properly calculate the native rewards for the given user.

Status The issue has been resolved in the following commit: `e174d4f`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Xenify` protocol, which is poised to revolutionise the DeFi landscape. `Xenify` uniquely integrates a "Burn to Earn" model with "Swap to Earn" capabilities, further enriched by advanced cross-chain, buyback, and burn functionalities. Operating with optimal efficiency, the protocol also incentivises user engagement through a sophisticated, game theory-based rewards mechanism. The current code base is clearly organized and 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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