SMART CONTRACT AUDIT REPORT

for

AgentFi

Prepared By: Xiaomi Huang

PeckShield

May 2, 2024

Document Properties

Client	AgentFi
Title	Smart Contract Audit Report
Target	AgentFi
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 2, 2024	Xuxian Jiang	Final Release
1.0-rc1	April 24, 2024	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 `AgentFi` 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 AgentFi

`AgentFi` is an innovative platform that introduces the concept of on-chain `AI` agents to the world of decentralized finance and beyond. At its core, `AgentFi` is designed to empower users with autonomous, intelligent agents capable of executing a broad spectrum of tasks directly on the blockchain. These tasks range from complex financial transactions to dynamic roles within interactive gaming environments, all performed with a level of sophistication and adaptability previously unseen in traditional crypto bots. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The AgentFi Protocol

Item	Description
Name	AgentFi
Website	https://agentfi.io
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 2, 2024

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

- https://github.com/AgentFi/agentfi-contracts.git (c1e9f15)

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

- https://github.com/AgentFi/agentfi-contracts.git (23a42c6)

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

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

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.

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 implementation of the `AgentFi` 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	2	■■
Low	3	■■■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key AgentFi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Address Spoofing in ERC2771Context/Multicall	Business Logic	Resolved
PVE-002	Low	Duplicate Validation Avoidance in BlastooorStrategyAgentAccount	Coding Practices	Resolved
PVE-003	Low	Potential Sandwich-Based MEV With DexBalancerModule	Time And State	Confirmed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Potential Address Spoofing in ERC2771Context/Multicall

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BlastooorAccountFactory`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

To lower the barrier for user interaction, `AgentFi` has the built-in support of meta transactions, which allows a third party `Relayer` send the transaction on behalf of the user. In the meantime, it also supports multicall to facilitate the user interaction. Unfortunately, the simultaneous use of `Multicall` and meta transactions may come with a so-called address spoofing risk if the underlying implementation is not carefully engineered.

To elaborate, we show below the code snippet of two related routines, i.e., `multicall()` and `msgSender()`. While each routine is rather straightforward and achieves the intended functionality, the combined use allows for the complete spoofing of the received `msgSender()`. In particular, if a call is originated from a trusted forwarder, the actual caller's address is extracted from the last 20 bytes of the calldata. However, the `multicall()` routine does not properly propagate the caller adjustment into each internal call. The detailed description of this issue can be found here: https://blog.openzeppelin.com/arbitrary-address-spoofing-vulnerability-erc2771context-multicall-public-disclosure.

```
    function multicall(bytes[] calldata data) external payable virtual returns (bytes[]
        memory results) {
        results = new bytes[](data.length);
        for (uint256 i = 0; i < data.length; ) {
            results[i] = Calls.functionDelegateCall(address(this), data[i]);
            unchecked { i++; }
        }
        return results;
```

```
    }
```

Listing 3.1: `Multicall::multicall()`

```
    function _msgSender() internal view virtual override returns (address sender) {
        if (isTrustedForwarder(msg.sender) && msg.data.length >= 20) {
            // The assembly code is more direct than the Solidity version using `abi.
                decode`.
            /// @solidity memory-safe-assembly
            assembly {
                sender := shr(96, calldataload(sub(calldatasize(), 20)))
            }
        } else {
            return super._msgSender();
        }
    }
```

Listing 3.2: `ERC2771Context::msgSender()`

Recommendation Revise the multicall functions to ensure the caller will not be spoofed.

Status This issue has been fixed in the following commit: `23a42c6`.

3.2 Duplicate Validation Avoidance in BlastooorStrategyAgentAccount

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BlastooorStrategyAgentAccount`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `AgentFi` protocol makes innovative uses of nested `NFTs`, allowing for each strategy to be its own `NFT`. In the process of reviewing the strategy-related `TBAs` in `BlastooorStrategyAgentAccount`, we notice certain redundancy in current validation can be avoided.

To elaborate, we show below the implementation of the related routine `_strategyManagerPrecheck()`. As the name indicates, this routine is designed to perform a pre-check for all executions invoked by the strategy manager. Besides the call verification, it also ensures the contract account is not locked with timely state update. The call to another helper `_beforeExecute()` (line 202) makes the same validation again to ensure the contract account is not locked with the same state update. The duplicate validation is considered unnecessary and can be avoided.

```
    /**
     * @notice Precheck for all execute by strategy manager calls.
     */
    function _strategyManagerPrecheck() internal {
        _verifySenderIsValidExecutorOrHasRole(STRATEGY_MANAGER_ROLE);
        _verifyIsUnlocked();
        _updateState();
        _beforeExecute();
    }
```

Listing 3.3: `BlastooorStrategyAgentAccount:_strategyManagerPrecheck()`

```
    function _beforeExecute() internal override {
        super._beforeExecute();
        _verifyIsUnlocked();
        _updateState();
    }
```

Listing 3.4: `AccountV3:_beforeExecute()`

Recommendation Revise the above-mentioned routines to avoid duplicate validation and state updates.

Status This issue has been fixed in the following commit: `3372d5e`.

3.3 Potential Sandwich-Based MEV With DexBalancerModuleA

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `DexBalancerModuleA`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The `AgentFi` protocol has a built-in module named `DexBalancerModuleA`, which aims to automate high-yield strategies within the `Blast` ecosystem. It focuses on optimizing liquidity provider (`LP`) positions across multiple decentralized exchanges (`DEXs`). While examining the optimization logic, we notice it may expose certain `MEV` opportunity.

```
    function _depositThruster(uint256 wethAmount, uint256 usdbAmount) internal {
        // approve weth and usdb to router
        _checkApproval(_weth, _thrusterRouter030, wethAmount);
        _checkApproval(_usdb, _thrusterRouter030, usdbAmount);
        // add liquidity
        IThrusterRouter router = IThrusterRouter(_thrusterRouter030);
```

```
        router.addLiquidity(_weth, _usdb, wethAmount, usdbAmount, 0, 0, address(this),
            type(uint256).max);
        // stake lp token on hyperlock
        uint256 liquidity = IERC20(_thrusterLpToken).balanceOf(address(this));
        if(liquidity == 0) return;
        _checkApproval(_thrusterLpToken, _hyperlockStaking, liquidity);
        IHyperlockStaking(_hyperlockStaking).stake(_thrusterLpToken, liquidity, 0);
    }
```

Listing 3.5: DexBalancerModuleA::_depositThruster()

To elaborate, we show above the `_depositThruster()` routine. We notice the liquidity addition is routed to `hrusterRouter`. And the liquidity addition operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of operation. A similar issue also exists in other routines, including `_depositRingProtocol()` and `_depositBlasterswap()`.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above sandwich attacks to better protect the interests of protocol users.

Status This issue has been acknowledged. One solution to this issue is to calculate safe minimum amounts offchain and pass them in. The team chooses to optimize for lower gas cost and ease of use and chooses not to require these parameters. We agree with the team that the issues of this nature are not a major issue on L2s. And the team plans to revisit it if there is a need to a chain with more adversarial MEV.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DexBalancerModuleA`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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 analyze 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 3.6: USDT Token **Contract**

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. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
```

```
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.7: `SafeERC20::safeApprove()`

In current implementation, if we examine the `DexBalancerModuleA::_checkApproval()` routine that is designed to approve the recipient for the intended spending. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (line 250).

```
254    function _checkApproval(address token, address recipient, uint256 minAmount)
           internal {
255        if(IERC20(token).allowance(address(this), recipient) < minAmount) IERC20(token).
               approve(recipient, type(uint256).max);
256    }
```

Listing 3.8: `DexBalancerModuleA::_checkApproval()`

Note the `_swapTokenOut()` routine in the same contract can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in the following commit: `62053c9`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `AgentFi` contract, there is a privileged account (`owner`) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, manage permissions, and lock contract accounts). In the following, we show the representative functions potentially affected by the privilege of the privileged account.

```
    function postAgentCreationSettings(
        AgentCreationSettings calldata creationSettings
    ) external payable onlyOwner returns (
        uint256 creationSettingsID
    ) {
        // checks
        Calls.verifyHasCode(creationSettings.agentImplementation);
        // post
        _agentCreationSettings = creationSettings;
        emit AgentCreationSettingsPosted();
    }

    /**
     * @notice Adds a new signer that approve allowlist mints.
     * Can only be called by the contract owner.
     * @param signer The signer to add.
     */
    function addSigner(address signer) external payable onlyOwner {
        if(signer == address(0)) revert Errors.AddressZero();
        _isAuthorizedSigner[signer] = true;
        emit SignerAdded(signer);
    }

    /**
     * @notice Removes a signer.
     * Can only be called by the contract owner.
     * @param signer The signer to remove.
     */
    function removeSigner(address signer) external payable onlyOwner {
        _isAuthorizedSigner[signer] = false;
        emit SignerRemoved(signer);
    }

    /**
     * @notice Adds a new treasuryMinter that can mint from the treasury allocation.
     * Can only be called by the contract owner.
     * @param treasuryMinter The TreasuryMinter to add.
     */
    function addTreasuryMinter(address treasuryMinter) external payable onlyOwner {
        if(treasuryMinter == address(0)) revert Errors.AddressZero();
        _isAuthorizedTreasuryMinter[treasuryMinter] = true;
        emit TreasuryMinterAdded(treasuryMinter);
    }
```

Listing 3.9: Example Privileged Operations in `BlastooorGenesisFactory`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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

4 Conclusion

In this audit, we have analyzed the design and implementation of the `AgentFi` protocol, which is an innovative platform that introduces the concept of on-chain `AI` agents to the world of decentralized finance and beyond. At its core, `AgentFi` is designed to empower users with autonomous, intelligent agents capable of executing a broad spectrum of tasks directly on the blockchain. These tasks range from complex financial transactions to dynamic roles within interactive gaming environments, all performed with a level of sophistication and adaptability previously unseen in traditional crypto bots. The current code base is well structured and neatly organized. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.