SMART CONTRACT AUDIT REPORT

for

LatteSwap

Prepared By: Yiqun Chen

PeckShield
July 5, 2021

Document Properties

Client	LatteSwap
Title	Smart Contract Audit Report
Target	LatteSwap
Version	1.0
Author	Jing Wang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 5, 2021	Jing Wang	Final Release
1.0-rc	July 2, 2021	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
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 `LatteSwap` 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 LatteSwap

`LatteSwap` is a fair-launch fork of `UniswapV2` with a number of enhancements or extensions. The protocol focuses on fair distribution by making improvements on key contracts. In particular, the `MasterBarista` contract improves on the widely-used `MasterChef` contract by changing its core data structure to ensure rewards tokens are distributed fairly. This improvement will also help with ease of integration and composability. `LATTE` tokens will also be fairly distributed as pool allocations are always updated. The protocol also focuses on user experiences and gas efficiency with the ability to claim all rewards in a single transaction. It also includes certain bonus period which allows for configuration of the block rewards emission as well as the lock-up period which will help bootstrap its liquidity by rewarding early adopters.

The basic information of the `LatteSwap` protocol is as follows:

Table 1.1: Basic Information of The `LatteSwap` Protocol

Item	Description
Name	LatteSwap
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 5, 2021

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

- https://github.com/latteswap-official/latteswap-contract.git (cf982b3)

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

- https://github.com/latteswap-official/latteswap-contract.git (81f8630)

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

Likelihood

1.3 Methodology

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

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 1.3: 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the LatteSwap 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	0	
Medium	3	■ ■ ■
Low	2	■ ■
Informational	1	■
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 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key LatteSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Logic Of LATTE::transferAll()	Business Logic	Fixed
PVE-002	Medium	Possible Costly Share From Improper LatteVault Initialization	Time and State	Fixed
PVE-003	Informational	Redundant Code Removal	Coding Practices	Fixed
PVE-004	Low	Possible DoS Against Pool Removal	Business Logic	Confirmed
PVE-005	Low	Improved Logic Of LATTE::lock()	Business Logic	Fixed
PVE-006	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 Improved Logic Of LATTE::transferAll()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low

- Target: LATTE
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The LatteSwap protocol has a governance token, i.e., LATTE, which is an ERC20-compliant token with the voting support. During the analysis of the voting support, we notice a function transferAll() that has a flawed implementation in not honoring the voting functionality.

To elaborate, we show below the transferAll() function. This function is designed to move both locked and unlocked LATTE tokens to the specified recipient. However, it needs to be revised to also take into account the delegate, i.e., _moveDelegates(_delegates[_msgSender()], _delegates[_to], balanceOf(_msgSender())) (right after the token tranfer at line 195).

```
173   /// @dev Move both locked and unlocked LATTE to another account
174   /// @param _to The address to be received locked and unlocked LATTE
175   function transferAll(address _to) external {
176     require(msg.sender != _to, "LATTE::transferAll::no self-transferAll");

178     _locks[_to] = _locks[_to].add(_locks[msg.sender]);

180     if (_lastUnlockBlock[_to] < startReleaseBlock) {
181       _lastUnlockBlock[_to] = startReleaseBlock;
182     }

184     else if (block.number < endReleaseBlock) {
185         uint256 fromUnlocked = canUnlockAmount(msg.sender);
186         uint256 toUnlocked = canUnlockAmount(_to);
187         uint256 numerator = block.number.mul(_locks[msg.sender]).add(block.number.mul(
                _locks[_to])).sub(endReleaseBlock.mul(fromUnlocked)).sub(endReleaseBlock.mul
```

```
          (toUnlocked));
      uint256 denominator = _locks[msg.sender].add(_locks[_to]).sub(fromUnlocked).sub(
          toUnlocked);
      _lastUnlockBlock[_to] = numerator.div(denominator);
  }

  _locks[msg.sender] = 0;
  _lastUnlockBlock[msg.sender] = 0;

  _transfer(msg.sender, _to, balanceOf(msg.sender));
}
```

Listing 3.1: LATTE::transferAll()

Recommendation Properly move the delegate as well when the LATTE tokens are being transferred.

Status This issue has been fixed in this commit: 81f8630.

3.2 Possible Costly Share From Improper LatteVault Initialization

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: LatteVault
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

The LatteSwap protocol allows users to stake supported LATTE tokens into MasterBarista for additional protocol rewards. The rewards are presented in terms of the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the deposit() routine. This deposit() routine is used for participating users to deposit the supported asset (e.g., LATTE) and get respective rewards in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
function deposit(uint256 _amount) external whenNotPaused nonReentrant onlyEOA {
  require(_amount > 0, "LatteVault::deposit::nothing to deposit");

  IMasterBarista(masterBarista).harvest(address(latte));

  uint256 pool = balanceOf();
112     latte.safeTransferFrom(msg.sender, address(this), _amount);
113     uint256 currentShares = 0;
114     if (totalShares != 0) {
115       currentShares = (_amount.mul(totalShares)).div(pool);
116     } else {
117       currentShares = _amount;
118     }
119     UserInfo storage user = userInfo[msg.sender];
120 
121     user.shares = user.shares.add(currentShares);
122     user.lastDepositedTime = block.timestamp;
123 
124     totalShares = totalShares.add(currentShares);
125 
126     user.latteAtLastUserAction = user.shares.mul(balanceOf()).div(totalShares);
127     user.lastUserActionTime = block.timestamp;
128 
129     _earn();
130 
131     require(totalShares > 1e17, "LatteVault::deposit::no tiny shares");
132 
133     emit Deposit(msg.sender, _amount, currentShares, block.timestamp);
134   }
```

Listing 3.2: `LatteVault::deposit()`

Specifically, when the pool is being initialized (line 114), the share value directly takes the value of `currentShares = _amount` (line 711), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `currentShares = 1 WEI`. With that, the actor can further deposit a huge amount of `LATTE` with the goal of making the share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of share calculation to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been fixed in this commit: `81f8630`.

3.3 Redundant Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LatteVault`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

LatteSwap makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Pausable`, to facilitate its code implementation and organization. For example, the `LatteVault` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `LatteVault` contract, there is a specific modifier `onlyEOA()` which has a redundant logic in validating the current message sender is not a contract. Specifically, the `require(!_isContract(msg.sender))` check (line 89) is redundant and can be safely removed. The reason is that the requirement on `require(msg.sender == tx.origin)` (line 90) is sufficient to guarantee the message sender is an EOA account.

```
85   /**
86   * @notice Checks if the msg.sender is an EOA
87   */
88   modifier onlyEOA() {
89     require(!_isContract(msg.sender), "LatteVault::onlyEOA::contract not allowed");
90     require(msg.sender == tx.origin, "LatteVault::onlyEOA::only EOA");
91     _;
92   }
```

Listing 3.3: onlyEOA()

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status This issue has been fixed in this commit: `81f8630`.

3.4 Possible DoS Against Pool Removal

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterBarista`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The `LatteSwap` protocol extends the widely-used `MasterChef` contract to help bootstrap its liquidity by rewarding early adopters. Specifically, the protocol allows the owner to dynamically add or remove the reward pool. While examining the pool-removal logic, we notice the current implementation can be improved.

To facilitate our discussion, we show below the code snippet of the related `removePool()` routine. As the name indicates, it is designed to remove a previously-supported stake pool token from being supported. We notice the current logic validates the pool balance needs to equal to 0 before it can be successfully removed. With that, a malicious actor may intentionally send a tiny amount, e.g., `1 WEI`, to prevent the stake pool token from being removed.

```
/// @dev Remove pool. Can only be called by the owner.
/// @param _stakeToken The stake token pool to be removed
function removePool(
  address _stakeToken,
  address _prevToken
) external override onlyOwner {
  require(_stakeToken != address(latte), "MasterBarista::removePool::can't remove
      LATTE pool");
  require(pools.has(_stakeToken), "MasterBarista::removePool::pool not add yet");
  require(IERC20(_stakeToken).balanceOf(address(this)) == 0, "MasterBarista::
      removePool::pool not empty");

  massUpdatePools();

  totalAllocPoint = totalAllocPoint.sub(poolInfo[_stakeToken].allocPoint);

  pools.remove(_stakeToken, _prevToken);
  poolInfo[_stakeToken].allocPoint = 0;
  poolInfo[_stakeToken].lastRewardBlock = 0;
  poolInfo[_stakeToken].accLattePerShare = 0;
  poolInfo[_stakeToken].accLattePerShareTilBonusEnd = 0;

  updatePool0alloc();

  emit RemovePool(_stakeToken, 0, totalAllocPoint);
}
```

Listing 3.4: `MasterBarista::removePool()`

Recommendation Take a defensive approach to ensure the above removal logic will not be blocked.

Status This issue has been confirmed. After considerations, the team decides to leave it as is since the motive for this attacking vector is low and the attacker will not get anything from it.

3.5 Improved Logic Of LATTE::lock()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `LATTE`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the governance token `LATTE` has been enhanced with the voting support. In addition, the current token contract also supports the locking functionality which allows the current owner to lock certain amount from a given account. While examining the lock support, we find the current implementation can be improved.

To elaborate, we show below the `lock()` routine from the `LATTE` token contract. When the given account is the token contract itself, it is possible to arbitrarily increase the lock amount of the token contract, including `_locks[_account]` (line 130) and`_totalLock` (line 131). To mitigate that, it is suggested to validate the given `_account` so that it cannot be the token contract itself.

```
121  /// @dev Lock LATTE based-on the command from MasterBarista
122  /// @param _account The address that will own this locked amount
123  /// @param _amount The locked amount
124  function lock(address _account, uint256 _amount) external onlyOwner {
125    require(_account != address(0), "LATTE::lock::no lock to address(0)");
126    require(_amount <= balanceOf(_account), "LATTE::lock::no lock over balance");

128    _transfer(_account, address(this), _amount);

130    _locks[_account] = _locks[_account].add(_amount);
131    _totalLock = _totalLock.add(_amount);

133    if (_lastUnlockBlock[_account] < startReleaseBlock) {
134      _lastUnlockBlock[_account] = startReleaseBlock;
135    }
```

```
    emit Lock(_account, _amount);
  }
```

Listing 3.5: `LATTE::lock()`

Recommendation Improve the validation of the above `lock()` routine with `require(_account != address(0)&& _account != address(this))`.

Status This issue has been fixed in this commit: `81f8630`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MasterBarista`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `LatteSwap` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., adding new pools, and setting various parameters). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the `setLattePerBlock()` routine from the `MasterBarista` contract. This routine sets the reward number of `LATTE` tokens per block. Also, the `setPool0allocBps()` routine is designed to allow the owner to set the percentage of `pool 0`.

```
  /// @dev Set LATTE per block.
  /// @param _lattePerBlock The new emission rate for LATTE
  function setLattePerBlock(uint256 _lattePerBlock) external onlyOwner {
    massUpdatePools();
    lattePerBlock = _lattePerBlock;
  }

  /// @dev Set pool 0 alloc BPS
  /// @param _pool0allocBps The new pool0 alloc Bps
  function setPool0allocBps(uint256 _pool0allocBps) external onlyOwner {
    require(_pool0allocBps > 1000, "MasterBarista::setPool0allocBps::_pool0allocBps must
         > 1000");
    massUpdatePools();
    pool0allocBps = _pool0allocBps;
    updatePool0alloc();
    emit Pool0allocChanged(pool0allocBps);
  }
```

Listing 3.6: `MasterBarista::setLattePerBlock()/setPool0allocBps()`

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. We point out that a compromised `owner` account is capable of modifying current protocol configuaration with adverse consequences, including permanent lock-down of user funds.

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 confirmed by the teams. And the team mitigates this issue with the addition of a timelock for privileged operations.

4 Conclusion

In this audit, we have analyzed the `LatteSwap` protocol design and implementation. The `LatteSwap` protocol is a fair-launch fork of `UniswapV2` with a number of enhancements or extensions. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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