# SMART CONTRACT AUDIT REPORT

for

# UNIVERSE FINANCE

Prepared By: Yiqun Chen

PeckShield

August 23, 2021

## Document Properties

| | |
|---|---|
| Client | Universe Finance |
| Title | Smart Contract Audit Report |
| Target | Universe Finance |
| Version | 1.0 |
| Author | Xiaotao Wu |
| Auditors | Xiaotao Wu, Xuxian Jiang |
| Reviewed by | Yiqun Chen |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | August 23, 2021 | Xiaotao Wu | Final Release |

## 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 4
- 1.1 About Universe Finance . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Improved Sanity Checks Of System/Function Parameters . . . 11
- 3.2 Inaccurate Total Amounts Calculation . . . 13
- 3.3 Trust Issue of Admin Keys . . . 14
- 3.4 Meaningful Events For Important State Changes . . . 17

4 Conclusion 18

References 19

# 1 Introduction

Given the opportunity to review the design document and related source code of the Universe Finance smart contracts, 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 Universe Finance

`Universe Finance` is a risk graded `Uniswap V3` active liquidity management platform and can meet the needs of liquidity providers with different risk preferences through a series of composable products. These composable products include the `Uniswap V3` revenue back-testing system, the `Uniswap V3` active LP management quantitative strategy, as well as the `Uniswap V3` smart/private/hedge/leverage vault. The goals of `Universe Finance` are to solve the problems of high risk, high cost, rebalancing, reinvestment and hedging faced by liquidity providers in `Uniswap V3` liquidity mining. `Universe Finance` serves users with different risk preferences, such as conservative, robust, active and radical.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Universe Finance

| Item | Description |
|---|---|
| Name | Universe Finance |
| Website | https://beta.universe.finance/ |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | August 23, 2021 |

In the following, we show the Git repositories of reviewed files and the commit hash values used

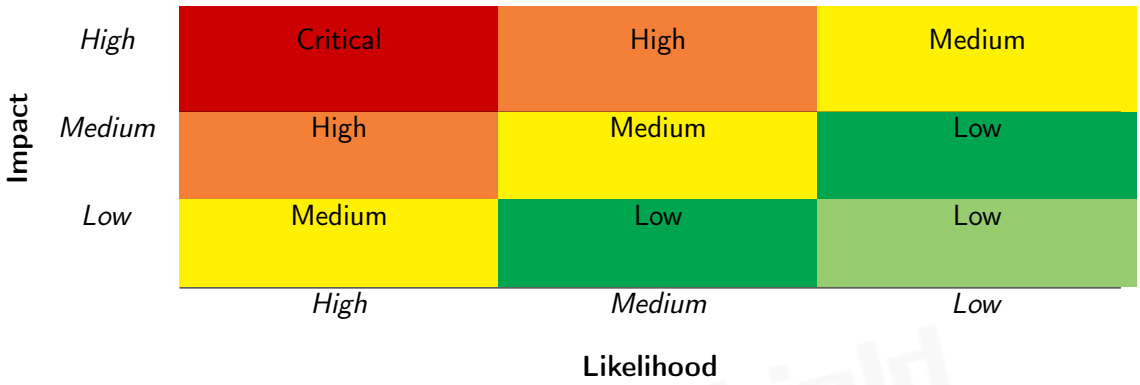
in this audit.

- https://github.com/UniverseFinance/UniverseFinancePrivateProtocol (ab33a3a)

## 1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification




## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be 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

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) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business 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 design and implementation of the `Universe Finance` smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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 | 1 | ■ |
| Low | 2 | ■■ |
| Informational | 1 | ■ |
| 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 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Improved Sanity Checks Of System/-Function Parameters | Coding Practices | Fixed |
| PVE-002 | Low | Inaccurate Total Amounts Calculation | Business Logic | Confirmed |
| PVE-003 | Medium | Trust Issue of Admin Keys | Security Features | Confirmed |
| PVE-004 | Informational | Meaningful Events For Important State Changes | Coding Practices | Fixed |

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 Sanity Checks Of System/Function Parameters

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

### Description

In the `PrivateStrategy` contract, the `_addLiquidity()` function is used to add liquidity to `Uniswap V3` pool. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the full implementation of the `_positionInit/_addLiquidity()` functions. Specifically, the execution of `pool.mint()` will revert when `lowerTick >= upperTick` (line 291). Since the state variable `boundaryThreshold` is defined as `int24` type and can be configured to be less than 0 (line 88), it is possible that `lowerTick >= upperTick` when the values of `lowerTick` and `upperTick` are updated in the `_positionInit()` routine (lines 279 and 280).

```
272     function _positionInit() internal {
273         if(isInit) {return; }
274         // get New Ticks
275         (uint160 sqrtPrice, , , , , , ) = pool.slot0();
276         int24 tick = TickMath.getTickAtSqrtRatio(sqrtPrice);
277         tick = _floor(tick, tickSpacing);
278         // update position
279         lowerTick = tick - boundaryThreshold;
280         upperTick = tick + boundaryThreshold;
281         isInit = true;
282     }
283
284     function _addLiquidity() internal {
285         // get balance0 & balance1
286         uint256 balance0 = token0.balanceOf(address(this));
```

```
287         uint256 balance1 = token1.balanceOf(address(this));
288         // add Liquidity on Uniswap
289         uint128 liquidity = _liquidityForAmounts(balance0, balance1);
290         if (liquidity > 0) {
291             pool.mint(
292                 address(this),
293                 lowerTick,
294                 upperTick,
295                 liquidity,
296                 ""
297             );
298         }
299     }
```

Listing 3.1: `PrivateStrategy::_positionInit()/_addLiquidity()`

```
83      function changeConfig(
84          int24 _boundaryThreshold,
85          int24 _reBalanceThreshold,
86          uint24 _swapPoolFee
87      ) external override onlyBindVault {
88          boundaryThreshold = _floor(_boundaryThreshold, tickSpacing);
89          reBalanceThreshold = _reBalanceThreshold;
90          swapPoolFee = _swapPoolFee;
91      }
```

Listing 3.2: `PrivateStrategy::changeConfig()`

Note a similar issue is also present in the `changeConfig()` routine of the `PrivateVault` contract. Currently, there is no range verification for `swapPoolFee` when the `operator` changes the configs for the intended `strategy`.

```
88      function changeConfig(
89          int24 boundaryThreshold,
90          int24 reBalanceThreshold,
91          uint24 swapPoolFee
92      ) external onlyOperator {
93          require(boundaryThreshold > reBalanceThreshold, "invalid params!");
94          strategy.changeConfig(boundaryThreshold, reBalanceThreshold, swapPoolFee);
95      }
```

Listing 3.3: `PrivateVault::changeConfig()`

**Recommendation** Validate the values of `lowerTick` and `upperTick` to ensure `lowerTick < upperTick` in the `PrivateStrategy` contract. Also add necessary range verification for `swapPoolFee` in `changeConfig()` to ensure it falls in the intended range.

**Status** The issue has been fixed by this commit: `b0a1feb`.

## 3.2 Inaccurate Total Amounts Calculation

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

### Description

The PrivateStrategy contract provides an external function getTotalAmounts() to query the equivalent numbers of token0 and token1 held by the contract. While reviewing the implementation of this routine, we notice that the total amounts calculation of token0 and token1 may not be accurate.

To elaborate, we show below the full implementation of the getTotalAmounts/_positionInfo() functions. Specifically, the values of tokensOwed0 and tokensOwed1 are retrieved from the Uniswap V3 pool and they may not up to date (line 200). In other words, the final calculation results may be inaccurate.

```
261     function getTotalAmounts() external view override returns (uint128, uint256, uint256
            ) {
262         // get position info
263         return _positionInfo();
264     }
```

Listing 3.4: PrivateStrategy::getTotalAmounts()

```
194     function _positionInfo() internal view returns (uint128, uint256, uint256) {
195         if (!isInit) {
196             return(0, 0, 0);
197         }
198         // query liquidity
199         bytes32 positionKey = keccak256(abi.encodePacked(address(this), lowerTick,
                upperTick));
200         (uint128 liquidity, , , uint128 tokensOwed0, uint128 tokensOwed1) = pool.
                positions(positionKey);
201         // get amount0 amount1
202         (uint256 amount0, uint256 amount1) = _amountsForLiquidity(liquidity);
203         amount0 = amount0.add(tokensOwed0);
204         amount1 = amount1.add(tokensOwed1);
205         return (liquidity, amount0, amount1);
206     }
```

Listing 3.5: PrivateStrategy::_positionInfo()

The above functions affect other public functions, including calBalance(), getUserBalance(), and getBalancedAmount() of the PrivateVault contract.

**Recommendation** Update the `PrivateStrategy` contract's `Uniswap V3` pool position status before retrieving the position parameters from the pool.

**Status** This issue has been confirmed. The possible impact of this issue is that the read results may not be up-to-date if these functions are called directly by users. There will be no deviation when users actually interact with the `Universe Finance` protocol.

## 3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

### Description

In the `Universe Finance` protocol, there are certain privileged accounts, i.e., `_owner` and `operator`. When examining the related contracts, i.e., `PrivateVault` and `PrivateStrategy`, we notice inherent trust on these privileged accounts. To elaborate, we show below the related functions.

Firstly, the `changeOperator()` function allows for the `_owner` to change the `operator` of the `PrivateVault` contract and the `updateWhiteList()` function allows for the `_owner` to update the `whiteLists` of the `PrivateVault` contract.

```
65      function changeOperator(address _operator) external onlyOwner {
66          require(_operator != address(0), "invalid address");
67          operator = _operator;
68      }
69
70      function updateWhiteList(address _address, bool status) external onlyOwner {
71          require(_address != address(0), "invalid address");
72          whiteLists[_address] = status;
73          emit UpdateWhiteList(msg.sender, _address, status);
74      }
```

Listing 3.6: `PrivateVault::changeOperator()/updateWhiteList()`

Secondly, the `operator` of the `PrivateVault` contract has the right to set the core parameters of the contract, change config parameters of the `PrivateStrategy` contract, rebalance and reinvest to the `Uniswap V3` pool.

```
78      function setCoreParams(
79          bool _canDeposit,
80          uint256 _reinvestMin0,
81          uint256 _reinvestMin1
```

```
82      ) external onlyOperator {
83          canDeposit = _canDeposit;
84          reinvestMin0 = _reinvestMin0;
85          reinvestMin1 = _reinvestMin1;
86      }
87
88      function changeConfig(
89          int24 boundaryThreshold,
90          int24 reBalanceThreshold,
91          uint24 swapPoolFee
92      ) external onlyOperator {
93          require(boundaryThreshold > reBalanceThreshold, "invalid params!");
94          strategy.changeConfig(boundaryThreshold, reBalanceThreshold, swapPoolFee);
95      }
96
97      function reBalance() external onlyOperator {
98          // ReBalance
99          (
100         bool status,
101         uint256 feesFromPool0,
102         uint256 feesFromPool1,
103         int24 lowerTick,
104         int24 upperTick
105         ) = strategy.reBalance();
106         if (status) {
107             // before mining
108             _transferToStrategy();
109             // add liquidity
110             strategy.mining();
111             // EVENT
112             (, uint256 amt0, uint256 amt1) = strategy.getTotalAmounts();
113             // EVENT
114             emit ReBalance(msg.sender, lowerTick, upperTick, amt0, amt1);
115             emit CollectFees(msg.sender, feesFromPool0, feesFromPool1, amt0, amt1,
                    _currentTick());
116         }
117     }
118
119     function reInvest() external onlyOperator {
120         // ReInvest
121         // update Commission
122         strategy.updateCommission();
123         // collect
124         (uint256 feesFromPool0, uint256 feesFromPool1) = strategy.collectCommission(
                address(0));
125         // before mining
126         _transferToStrategy();
127         // add liquidity
128         strategy.mining();
129         // EVENT
130         ( , int24 lowerTick, int24 upperTick) = strategy.positions(address(0));
131         ( , uint256 amt0, uint256 amt1) = strategy.getTotalAmounts();
```

```
132         emit CollectFees(msg.sender, feesFromPool0, feesFromPool1, amt0, amt1,
                _currentTick());
133         emit ReInvest(msg.sender, lowerTick, upperTick, amt0, amt1);
134     }
```

Listing 3.7: PrivateVault::setCoreParams()/changeConfig()/reBalance()/reInvest()

Lastly, the setVault() function allows for the _owner to set the vault address for the PrivateStrategy contract. If the _owner sets the wrong vault address, the invocations of the following functions may fail, including changeConfig(), mining(), stopMining(), reBalance(), updateCommission(), and collectCommission().

```
76     function setVault(address _vault) external onlyOwner {
77         require(vault == address(0), 'already INIT!');
78         vault = _vault;
79     }
```

Listing 3.8: PrivateStrategy::setVault()

We understand the need of the privileged function for contract operation, but at the same time the extra power to the _owner/operator may also be a counter-party risk to the contract 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 _owner/operator explicit to Universe Finance users.

**Status** This issue has been confirmed. The team clarifies that for the above vault address, it can only be set once. Otherwise, the team will re-deploy the PrivateStrategy contract.

## 3.4 Meaningful Events For Important State Changes

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

### Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the PrivateVault contract as an example. While examining the events that reflect the PrivateVault dynamics, we notice there is a lack of emitting related event that reflect important state changes. Specifically, when the operator is being changed, there is no corresponding event being emitted to reflect the change of operator (line 67).

```
65    function changeOperator(address _operator) external onlyOwner {
66        require(_operator != address(0), "invalid address");
67        operator = _operator;
68    }
```

Listing 3.9: PrivateVault::changeOperator()

Similarly, the setVault() function of the PrivateStrategy contract can be improved to emit a related event, i.e., SetVault(vault, _vault) (right after line 77), when the vault address is being set.

**Recommendation** Properly emit the related NewOperator event when the operator is being changed. Also properly emit the SetVault event when the vault address is being set.

**Status** The issue has been fixed by this commit: cd0c2db.

# 4 Conclusion

In this audit, we have analyzed the Universe Finance design and implementation. As a risk graded Uniswap V3 active liquidity management platform, Universe finance can meet the needs of liquidity providers with different risk preferences through a series of composable products. 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-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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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