SMART CONTRACT AUDIT REPORT

for

dForce Vote Escrow

Prepared By: Yiqun Chen

Hangzhou, China

February 26, 2022

Document Properties

Client	dForce Network
Title	Smart Contract Audit Report
Target	dForce Vote Escrow
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 26, 2022	Shulin Bie	Final Release
1.0-rc	February 25, 2022	Shulin Bie	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 `dForce Vote Escrow`, 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 dForce Vote Escrow

`dForce Network` serving as DeFi infrastructure in Web3 provides a complete set of decentralized finance protocols covering assets, lending, and trading. The audited `dForce Vote Escrow` protocol, as an important component of the `dForce Network` ecosystem, allows the user to stake the `DF` token and get in return the `sDF` token. Meanwhile, it also allows the user to stake the `sDF` token and get in return the `veDF` token, which represents the voting power of the staker in community governance. The stakers can profit from the stake of the `DF` token and `sDF` token. The `dForce Vote Escrow` protocol enriches the `dForce Network` ecosystem.

Table 1.1: Basic Information of dForce Vote Escrow

Item	Description
Target	dForce Vote Escrow
Website	https://dforce.network/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 26, 2022

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

- https://github.com/dforce-network/vDFStaking/tree/audit (6ad127a)

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

- https://github.com/dforce-network/vDFStaking/tree/audit (65b5243)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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.

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

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) [10], 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 `dForce Vote Escrow` 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 logic, 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	3	■ ■ ■
Undetermined	1	■
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 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key dForce Vote Escrow Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly *sDF* From Improper Stake Initialization In StakedDF	Time and State	Resolved
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Low	Redundant State/Code Removal	Coding Practices	Resolved
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-005	Undetermined	Revisited Logic Of StakedDF::unstake()	Business Logic	Resolved

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 Possible Costly *sDF* From Improper Stake Initialization In StakedDF

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `StakedDF`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [4]

Description

The `StakedDF` contract allows the user to stake the supported `DF` token and get in return LP token (i.e., `sDF`) to represent the pool shares. While examining the share calculation with the given stakes, we notice an issue that may unnecessarily make the pool token extremely expensive and bring hurdles (or even causes loss) for later stakers.

To elaborate, we show below the related code snippet of the `StakedDF` contract. The `stake()` routine is used for participating users to stake the supported `DF` token and get respective `sDF` token in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function stake(address _recipient, uint256 _rawUnderlyingAmount)
        external
        nonReentrant
        returns (uint256 _tokenAmount)
    {
        require(_recipient != address(0), "stake: Mint to the zero address!");
        require(
            _rawUnderlyingAmount != 0,
            "stake: Stake amount can not be zero!"
        );

        uint256 _exchangeRate = _calculateExchange();
        _tokenAmount = _rawUnderlyingAmount.rdiv(_exchangeRate);
```

```
211
212         _mint(_recipient, _tokenAmount);
213         DF.transferFrom(msg.sender, address(this), _rawUnderlyingAmount);
214
215         emit Stake(msg.sender, _recipient, _rawUnderlyingAmount, _tokenAmount);
216     }
```

Listing 3.1: `StakedDF::stake()`

Specifically, when the pool is being initialized, the amount of the minted `sDF` token directly takes the value of `_rawUnderlyingAmount` (line 210), which is under control by the malicious actor. As this is the first stake, the current total supply equals the calculated `_tokenAmount = _rawUnderlyingAmount .rdiv(_exchangeRate)= 1WEI`. With that, the actor can further transfer a huge amount of `DF` token to the `StakedDF` contract with the goal of making the `sDF` extremely expensive.

An extremely expensive pool token 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 staked assets. If truncated to be zero, the staked 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 `stake()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first stake to avoid being manipulated.

Status The issue has been acknowledged and the team will exercise extra care in safely initializing the pool.

3.2 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1109 [2]

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 `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transferFrom()`, there is a check, i.e., `if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event. The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: `ZRX.sol`

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, 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 `approve()` as well, i.e., `safeApprove()`.

In the following, we show the `stake()` routine in the `StakedDF` contract. If the `USDT` token is supported as `DF`, the unsafe version of `DF.transferFrom(msg.sender, address(this), _rawUnderlyingAmount )` (line 213) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value). We may intend to replace `DF.transferFrom(msg.sender, address(this), _rawUnderlyingAmount)` (line 213) with `safeTransferFrom()`.

```
198     function stake(address _recipient, uint256 _rawUnderlyingAmount)
199         external
200         nonReentrant
201         returns (uint256 _tokenAmount)
202     {
203         require(_recipient != address(0), "stake: Mint to the zero address!");
204         require(
205             _rawUnderlyingAmount != 0,
206             "stake: Stake amount can not be zero!"
207         );
208 
209         uint256 _exchangeRate = _calculateExchange();
210         _tokenAmount = _rawUnderlyingAmount.rdiv(_exchangeRate);
211 
212         _mint(_recipient, _tokenAmount);
213         DF.transferFrom(msg.sender, address(this), _rawUnderlyingAmount);
214 
215         emit Stake(msg.sender, _recipient, _rawUnderlyingAmount, _tokenAmount);
216     }
```

Listing 3.3: `StakedDF::stake()`

Note that a number of routines can be similarly improved, including `StakedDF::stake()`, `veDFManager::supplySDFUnderlying()/createInOne()/initialize()`, and `veDFCore::initialize()`.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transferFrom()` and `approve()`.

Status The issue has been addressed by the following commits: `84fc2af` and `abf7f0d`.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: GovernanceToken/veDF
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

In the dForce Vote Escrow protocol, the GovernanceToken and veDF contracts implement a non-transferrable ERC20 token (i.e., veDF), which allow the user to stake the sDF token and get in return the veDF token to represent the voting power of the staker. While examining their logic, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

To elaborate, we show below the related code snippet of the contracts. The internal GovernanceToken ::_transferTokens() function is designed to transfer the veDF token. However, we notice it is not called anywhere in the contract, which is consistent with the design that the veDF token is non-transferrable. Moreover, we notice the DOUBLE_BASE variable in the veDF contract is also not used anywhere. Given this, we suggest to remove them safely.

```
    function _transferTokens(address src, address dst, uint96 amount) internal {
        require(src != address(0), "vDF::_transferTokens: cannot transfer from the zero address");
        require(dst != address(0), "vDF::_transferTokens: cannot transfer to the zero address");

        balances[src] = sub96(balances[src], amount, "vDF::_transferTokens: transfer amount exceeds balance");
        balances[dst] = add96(balances[dst], amount, "vDF::_transferTokens: transfer amount overflows");
        emit Transfer(src, dst, amount);

        _moveDelegates(delegates[src], delegates[dst], amount);
    }
```

Listing 3.4: GovernanceToken::_transferTokens()

```
    contract veDF is Initializable, Ownable, ReentrancyGuard, GovernanceToken {
        using SafeRatioMath for uint256;
        using SafeMathUpgradeable for uint256;
        using SafeERC20Upgradeable for IERC20Upgradeable;
        using EnumerableSetUpgradeable for EnumerableSetUpgradeable.AddressSet;

        /// @dev Calc the base value
        uint256 internal constant BASE = 1e18;
        /// @dev Calc the double of the base value
```

```
    uint256 internal constant DOUBLE_BASE = 1e36;

        ...
}
```

Listing 3.5: `veDF`

Recommendation Consider the removal of the redundant state.

Status The issue has been addressed by the following commit: `0f9119c`.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `veDFCore/veDF`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `dForce Vote Escrow` protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., manage the privileged `minter` account). In the following, we show the representative functions potentially affected by the privilege of the account.

```
function _addMinter(address _minter) external onlyOwner {
    require(_minter != address(0), "_minter not accepted zero address.");
    if (minters.add(_minter)) {
        emit MinterAdded(_minter);
    }
}

function _removeMinter(address _minter) external onlyOwner {
    require(_minter != address(0), "invalid minter address.");
    if (minters.remove(_minter)) {
        emit MinterRemoved(_minter);
    }
}
```

Listing 3.6: `veDF::_addMinter()&&_removeMinter()`

```
function rescueTokens(
    IERC20Upgradeable _token,
    uint256 _amount,
    address _to
) external onlyRewardDistributor {
    _token.safeTransfer(_to, _amount);
```

```
    }
```

Listing 3.7: `veDFCore::rescueTokens()`

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 privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `dForce Vote Escrow` 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 This issue has been confirmed and the team clarifies that there is no locked token in `veDFCore` contract and `sDF` is locked in the `veDF` contract.

3.5 Revisited Logic Of StakedDF::unstake()

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `StakedDF`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.1, the `StakedDF` contract accepts the stake of the supported `DF` token with LP token (i.e., `sDF`) minted. In addition, the unstaking logic is supported to burn the LP token to withdraw the staked `DF` token. While examining its logic, we notice the allowance-related design may be revisited.

To elaborate, we show below the related code snippet of the `StakedDF` contract. By design, the `unstake()` routine allows the staker himself to withdraw the staked `DF` token. Meanwhile, it also allows the user that has the approval of the staker to withdraw the staked `DF` token on behalf of the staker. However, we notice the withdrawn `DF` token is transferred to the `_caller` (i.e., `msg.sender`) rather than the staker (line 289). Given this, we suggest to revise the implementation as below: `DF.safeTransfer(_from, _underlyingAmount)` (line 289).

```
272     function unstake(address _from, uint256 _rawTokenAmount)
273         external
274         nonReentrant
275     {
```

```
        require(
            _rawTokenAmount != 0,
            "unstake: Unstake amount can not be zero!"
        );

        uint256 _exchangeRate = _calculateExchange();
        uint256 _underlyingAmount = _rawTokenAmount.rmul(_exchangeRate);

        address _caller = msg.sender;

        _burnFrom(_from, _caller, _rawTokenAmount);

        getTokenFromVault(_underlyingAmount);
        DF.safeTransfer(_caller, _underlyingAmount);

        emit Unstake(_from, _caller, _underlyingAmount, _rawTokenAmount);
    }
```

Listing 3.8: `StakedDF::unstake()`

Note that the `unstakeUnderlying()` routine shares the same issue.

Recommendation Revisit the logic behind the `unstake()` and `unstakeUnderlying()` routines as above-mentioned.

Status The issue has been confirmed as the team clarifies that this is indeed part of design.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `dForce Vote Escrow` protocol, which is an important component of the `dForce Network` ecosystem. The protocol allows the user to stake the `DF` token with the `sDF` token minted. Meanwhile, it also allows the user to stake the `sDF` token with the `veDF` token minted, which represents the voting power of the staker. The stakers can profit from the stake of the `DF` token and `sDF` token. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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