SMART CONTRACT AUDIT REPORT

for

DFORCE NETWORK

Prepared By: Shuxiao Wang

Hangzhou, China

Aug. 30, 2020

Document Properties

Client	dForce Network
Title	Smart Contract Audit Report
Target	USDx and USR
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Chiachih Wu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Aug. 30, 2020	Xuxian Jiang	Final Release
0.3	Aug. 28, 2020	Xuxian Jiang	Additional Findings #2
0.2	Aug. 26, 2020	Xuxian Jiang	Additional Findings #1
0.1	Aug. 24, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the recent updates on **USDx and USR**, we in the report outline 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About USDx and USR

As an on-chain synthetic indexed USD stablecoin protocol, USDx is 1:1 pegged to a basket of selected stablecoins. USDx is implemented as an ERC20 token and can be automatically minted with a basket of constituent stablecoins (mostly fiat-back with high transparency and liquidity) through smart contracts. USDx Savings Rate is an addition that allows any USDx holder to earn risk-free savings. The savings paid out to USDx holders are financed by depositing constituent stable coins into a number of decentralized lending markets (e.g., Compound and Aave) to earn interests.

The basic information of USDx and USR is as follows:

Table 1.1: Basic Information of USDx and USR

Item	Description
Issuer	dForce Network
Website	https://dforce.network/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Aug. 30, 2020

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/USR (96697d8)
- https://github.com/dforce-network/USDx_1.0 (228fe0b)

1.2 About PeckShield

PeckShield Inc. [20] 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 [15]:

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 an 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 USDx and USR implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	4	■■■■
Low	2	■■
Informational	3	■■■
Total	9	

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 4 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key USDx and USR Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility With Deflationary Tokens	Business Logics	Confirmed
PVE-002	Medium	Implicit Assumption of Pegged Ingredient Stablecoins	Business Logics	Confirmed
PVE-003	Medium	Lack of Sanity Checks in setMinBurnAmount()	Security Features	Confirmed
PVE-004	Informational	Improved Unwrapping of Ingredient Stablecoins	Numeric Errors	Confirmed
PVE-005	Medium	Fettered Admin Transfer of Upgradeable Contracts	Security Features	Confirmed
PVE-006	Informational	Wrapped Collateral Calculation in Assets Balance Check	Time and State	Fixed
PVE-07	Low	Improved Precision in Interest Calculation	Numeric Errors	Fixed
PVE-008	Informational	Better Allocation of Early Bird Bonus	Business Logics	Confirmed
PVE-009	Medium	Reentrancy Risks With ERC777 Tokens	Concurrency	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incompatibility With Deflationary Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: DFEngineV2
- Category: Business Logics [11]
- CWE subcategory: CWE-708 [7]

Description

USDx is an on-chain synthetic indexed USD stablecoin protocol and its tokens can be minted with a basket of constituent stablecoins (mostly fiat-back with high transparency and liquidity) in a trustless manner. Behind the scene, the DFEngineV2 smart contract acts as the engine to facilitate the minting or redemption of USDx tokens. The facilitation is enabled by providing a set of well-defined APIs, i.e., deposit(), withdraw(), claim(), destroy(), and oneClickMinting().

Naturally, these functions are involved in transferring users' assets into (or out of) the USDx protocol. Using the deposit() function as an example, it needs to transfer deposited assets from the user account to DFPoolV2 (line 88). When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts (lines 106 – 107).

```
82     function deposit(address _depositor, address _srcToken, uint _feeTokenIdx, uint
           _srcAmount) public auth nonReentrant returns (uint) {
83         address _tokenID = dfStore.getWrappedToken(_srcToken);
84         require(dfStore.getMintingToken(_tokenID), "Deposit: asset is not allowed.");

86         uint _amount = IDSWrappedToken(_tokenID).wrap(address(dfPool), _srcAmount);
87         require(_amount > 0, "Deposit: amount is invalid.");
88         dfPool.transferFromSender(_srcToken, _depositor, IDSWrappedToken(_tokenID).
               reverseByMultiple(_amount));
89         _unifiedCommission(ProcessType.CT_DEPOSIT, _feeTokenIdx, _depositor, _amount);

91         address[] memory _tokens;
```

```
        uint[] memory _mintCW;
        (, , , _tokens, _mintCW) = dfStore.getSectionData(dfStore.getMintPosition());

        uint[] memory _tokenBalance = new uint[](_tokens.length);
        uint[] memory _resUSDXBalance = new uint[](_tokens.length);
        uint[] memory _depositorBalance = new uint[](_tokens.length);
        //For stack limit sake.
        uint _misc = uint(-1);

        for (uint i = 0; i < _tokens.length; i++) {
            _tokenBalance[i] = dfStore.getTokenBalance(_tokens[i]);
            _resUSDXBalance[i] = dfStore.getResUSDXBalance(_tokens[i]);
            _depositorBalance[i] = dfStore.getDepositorBalance(_depositor, _tokens[i]);
            if (_tokenID == _tokens[i]){
                _tokenBalance[i] = add(_tokenBalance[i], _amount);
                _depositorBalance[i] = add(_depositorBalance[i], _amount);
            }
            _misc = min(div(_tokenBalance[i], _mintCW[i]), _misc);
        }
        if (_misc > 0) {
            return _convert(_depositor, _tokens, _mintCW, _tokenBalance, _resUSDXBalance
                , _depositorBalance, _misc);
        }
        ...
    }
```

Listing 3.1: DFEngineV2.sol

However, in the cases of deflationary tokens, as shown in the above code snippets, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate portfolio management of `USDx` and affects protocol-wide operation and maintenance.

One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()/transferFrom()` is expected and aligned well with the intended operation. Though these additional checks cost additional gas usage, we feel that they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `USDx`. With the convenient reconfiguration of stablecoin ingredients, `USDx` is indeed in the position to effectively regulate the set of assets allowed into the protocol.

Recommendation Regulate the set of ERC20 tokens supported in `USDx` and, if there is a need to support deflationary tokens, add necessary mitigation mechanisms to keep track of accurate balances.

Status This issue has been confirmed. As there is no comprehensive solution yet, the team decides no change for the time being, but will think of a future solution for it.

3.2 Implicit Assumption of Pegged Ingredient Stablecoins

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Proposal.sol`
- Category: Business Logics [11]
- CWE subcategory: CWE-841 [8]

Description

As mentioned in Section 3.1, `DFEngineV2` defines a number of routines (e.g., `deposit()`, `withdraw()`, `claim()`, `destroy()`, and `oneClickMinting()`) to greatly facilitate the the minting or redemption of `USDx` tokens. These routines are meant to support a variety of operations. For example, `deposit()` allows users to deposit assets of supported stablecoins to mint `USDx`; `withdraw()` permits users to retrieve back un-minted stablecoins that are earlier deposited; `claim()` enables users to claim currently-minted `USDx` stablecoins that were previously not mintable due to an unbalanced (or unmatched) supply of ingredient tokens; `destroy()` redeems `USDx` tokens for the return of ingredient stablecoins; and `oneClickMinting()` conveniently defines an one-click solution to smoothly facilitate the entire minting process by providing required ingredient tokens all from the same user.

```
    function _unifiedCommission(ProcessType ct, uint _feeTokenIdx, address depositor,
        uint _amount) internal {
        uint rate = dfStore.getFeeRate(uint(ct));
        if(rate > 0) {
            address _token = dfStore.getTypeToken(_feeTokenIdx);
            require(_token != address(0), "_UnifiedCommission: fee token not correct.");
            uint dfPrice = getPrice(dfStore.getTokenMedian(_token));
            uint dfFee = div(mul(mul(_amount, rate), WAD), mul(10000, dfPrice));
            require(
                doTransferFrom(
                    _token,
                    depositor,
                    dfFunds,
                    dfFee
                ),
                "_unifiedCommission: transferFrom fee failed"
            );
        }
```

```
    }
```

Listing 3.2: DFEngineV2.sol

For each above operation, the USDx protocol provides an unified helper to calculate and process related commission fee that is denominated in DF or USDx. The helper is called _unifiedCommission() and a close examination shows that the commission fee is calculated with an implicit assumption of all supported stablecoins are pegged. Using the mint() as an example, the fee calculation is based on the amount of the deposited token, meaning that it is possible to pay smaller amount of commission fee when minting with one stablecoin than another. While the deviation of each individual stablecoin normally may not be significant and it is always possible to activate a new set of ingredient tokens, it is worthy to further explore possible design space and make the fee calculation unbiased, say based on the amount of affected USDx tokens.

Meanwhile, this assumption is also applicable to the minting and redemption of USDx and the operation of USR. This assumption could open up a time window of arbitrage and the mitigation may require activating a new set of ingredient tokens or even excluding fluctuating stablecoins as ingredients.

Recommendation Revisit the implicit assumption and make commission fee calculation unbiased with respect to underlying ingredient tokens.

Status This issue has been confirmed. The team considers it part of the design and thus appropriate to leave it as is (unless the design assumption will be changed in the future).

3.3 Lack of Sanity Checks in setMinBurnAmount()

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: DFStore.sol
- Category: Security Features [9]
- CWE subcategory: CWE-284 [4]

Description

The USDx protocol supports a number of system-wide risk parameters that can be dynamically configured after deployment. One example is minimalBurnAmount. As the name indicates, it specifies the minimal amount that can be burned. The risk parameter is checked in destroy() when the given amount of USDx is being redeemed.

An analysis with minimalBurnAmount shows that it does not simply mean the minimal amount of redemption. In fact, any redeemed amount needs to be fully divisible by minimalBurnAmount (line

193 in the following code snippet). Further, the configuration of `minimalBurnAmount` requires extra caution in avoiding the introduction of rounding issues. Any rounding issue here can easily lead to discrepancy of `USDx` `totalSupply` and collateral amount. The discrepancy, if present, will fail the final check, i.e., `checkUSDXTotalAndColTotal()` (line 242), preventing any `USDx` tokens from being redeemed.

```
192     function destroy(address _depositor, uint _feeTokenIdx, uint _amount) public auth nonReentrant returns (bool) {
193         require(_amount > 0 && (_amount % dfStore.getMinBurnAmount() == 0), "Destroy: amount not correct.");
194         require(_amount <= usdxToken.balanceOf(_depositor), "Destroy: exceed max USDX balance.");
195         require(_amount <= sub(dfStore.getTotalMinted(), dfStore.getTotalBurned()), "Destroy: not enough to burn.");
196         address[] memory _tokens;
197         uint[] memory _burnCW;
198         uint _sumBurnCW;
199         uint _burned;
200         uint _minted;
201         uint _burnedAmount;
202         uint _amountTemp = _amount;
203         uint _tokenAmount;

205         _unifiedCommission(ProcessType.CT_DESTROY, _feeTokenIdx, _depositor, _amount);

207         while(_amountTemp > 0) {
208             (_minted, _burned, , _tokens, _burnCW) = dfStore.getSectionData(dfStore.getBurnPosition());

210             _sumBurnCW = 0;
211             for (uint i = 0; i < _burnCW.length; i++) {
212                 _sumBurnCW = add(_sumBurnCW, _burnCW[i]);
213             }

215             if (add(_burned, _amountTemp) <= _minted){
216                 dfStore.setSectionBurned(add(_burned, _amountTemp));
217                 _burnedAmount = _amountTemp;
218                 _amountTemp = 0;
219             } else {
220                 _burnedAmount = sub(_minted, _burned);
221                 _amountTemp = sub(_amountTemp, _burnedAmount);
222                 dfStore.setSectionBurned(_minted);
223                 dfStore.burnSectionMoveon();
224             }

226             if (_burnedAmount == 0)
227                 continue;

229             for (uint i = 0; i < _tokens.length; i++) {

231                 _tokenAmount = div(mul(_burnedAmount, _burnCW[i]), _sumBurnCW);
232                 IDSWrappedToken(_tokens[i]).unwrap(dfCol, _tokenAmount);
233                 dfPool.transferOutSrc(
```

```
234                     IDSWrappedToken(_tokens[i]).getSrcERC20(),
235                     _depositor,
236                     IDSWrappedToken(_tokens[i]).reverseByMultiple(_tokenAmount));
237                 dfStore.setTotalCol(sub(dfStore.getTotalCol(), _tokenAmount));
238             }
239         }

241         usdxToken.burn(_depositor, _amount);
242         checkUSDXTotalAndColTotal();
243         dfStore.addTotalBurned(_amount);

245         return true;
246     }
```

Listing 3.3: DFEngineV2.sol

Considering the above implication, it is important to ensure the correctness when updating the `minimalBurnAmount` risk parameter. Note that the rounding issue may be introduced in `_tokenAmount = div(mul(_burnedAmount, _burnCW[i]), _sumBurnCW)` (line 231). We can perform similar sanity checks to ensure that the update will not introduce any rounding issue.

Meanwhile, when a new set of ingredients is introduced, the protocol has a weight requirement – `require(mul(div(mul(_weight[i], factor), sum), sum)== mul(_weight[i], factor))`, where `factor = 10 ** 10`. With that, we can naturally choose `minimalBurnAmount` to be multiple of `factor`.

Recommendation Ensure the correctness of the `minimalBurnAmount` risk parameter to avoid unnecessary mis-configuration.

Status This issue has been confirmed. The team has no plan yet to update this part of contract as it is too risky to update it and the update may require the migration of certain key storages.

3.4 Improved Unwrapping of Ingredient Stablecoins

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DSWrapedToken.sol`
- Category: Numeric Errors [13]
- CWE subcategory: CWE-190 [2]

Description

To accommodate possible decimal differences of ingredient stablecoins and unify the interface of their handling, the `USDx` protocol wraps each ingredient stablecoin with its own `DSWrapedToken`. In particular, two helper routines are provided, i.e., `wrap()` and `unwrap()`, to facilitate the conversion between an ingredient stablecoin and its `DSWrapedToken` counterpart.

```
23      function wrap(address _dst, uint _amount) public auth returns (uint) {
24          uint _xAmount = changeByMultiple(_amount);
25          mint(_dst, _xAmount);

27          return _xAmount;
28      }

30      function unwrap(address _dst, uint _xAmount) public auth returns (uint) {
31          burn(_dst, _xAmount);

33          return _xAmount;
34      }
```

Listing 3.4: DSWrapedToken.sol

Besides the amount conversion, both `wrap()` and `unwrap()` routines also add the functionality of dynamically minting (line 25) or burning (line 31) `DSWrapedToken`s. The purpose here is to precisely keep track of the amount of wrapped stablecoins.

However, we notice that the `unwrap()` helper does not convert the given amount (that is denominated in `DSWrapedToken`) back in the corresponding ingredient stablecoin. And the converted number is indeed in a number of occasions. Using the `withdraw()` as an example, the conversion is performed right after calling `unwrap()` (line 158). If we had a proper `unwrap()` that can return back the converted number, we can save a cross-contract function call (besides the benefit of having a cleaner execution flow and a better consistency between `wrap()` and `unwrap()`).

```
141     function withdraw(address _depositor, address _srcToken, uint _feeTokenIdx, uint
            _srcAmount) public auth nonReentrant returns (uint) {
142         address _tokenID = dfStore.getWrappedToken(_srcToken);
143         uint _amount = IDSWrappedToken(_tokenID).changeByMultiple(_srcAmount);
144         require(_amount > 0, "Withdraw: amount is invalid.");

146         uint _depositorBalance = dfStore.getDepositorBalance(_depositor, _tokenID);
147         uint _tokenBalance = dfStore.getTokenBalance(_tokenID);
148         uint _withdrawAmount = min(_amount, min(_tokenBalance, _depositorBalance));

150         if (_withdrawAmount <= 0)
151             return (0);

153         _depositorBalance = sub(_depositorBalance, _withdrawAmount);
154         dfStore.setDepositorBalance(_depositor, _tokenID, _depositorBalance);
155         dfStore.setTokenBalance(_tokenID, sub(_tokenBalance, _withdrawAmount));
156         _unifiedCommission(ProcessType.CT_WITHDRAW, _feeTokenIdx, _depositor,
                _withdrawAmount);
157         IDSWrappedToken(_tokenID).unwrap(address(dfPool), _withdrawAmount);
158         uint _srcWithdrawAmount = IDSWrappedToken(_tokenID).reverseByMultiple(
                _withdrawAmount);
159         dfPool.transferOut(_srcToken, _depositor, _srcWithdrawAmount);

161         return (_srcWithdrawAmount);
```

```
    }
```

Listing 3.5: DFEngineV2.sol

Recommendation Revise `unwrap()` to convert the amount back to the related stablecoin, as shown in the following implementation.

```
    function wrap(address _dst, uint _amount) public auth returns (uint) {
        uint _xAmount = changeByMultiple(_amount);
        mint(_dst, _xAmount);

        return _xAmount;
    }

    function unwrap(address _dst, uint _xAmount) public auth returns (uint) {
        burn(_dst, _xAmount);
        uint _amount = reverseByMultiple(_xAmount);

        return _amount;
    }
```

Listing 3.6: DSWrapedToken.sol (revised)

Status This issue has been confirmed. The team has no plan yet to update this part of contract for the same reason as outlined in Section 3.3.

3.5 Fettered Admin Transfer of Upgradeable Contracts

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `DFProxy`, `USRProxy`
- Category: Security Features [9]
- CWE subcategory: CWE-287 [5]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new (logic) contracts that are just deployed to replace old (logic) contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize`) that basically executes all the setup logic.

Another caveat comes from the mixed upgradeability and authentication. In the following, we show the code snippet of relevant implementation. For simplicity, we call the implementation contract as the logic contract that receives calls from the proxy contract. Both proxy and logic contracts have their own, independent `admin` for separate management. We notice that the proxy contract has a built-in `ifAdmin` modifier that essentially differentiates the caller: if the caller is the pre-configured `admin`, then this particular call is intended for the proxy. Otherwise, this call is delegated to the logic contract.

```
    function changeAdmin(address _newAdmin) external ifAdmin {
        require(
            _newAdmin != address(0),
            "Cannot change the admin of a proxy to the zero address"
        );
        require(
            _newAdmin != _admin(),
            "The current and new admin cannot be the same ."
        );
        require(
            _newAdmin != _pendingAdmin(),
            "Cannot set the newAdmin of a proxy to the same address ."
        );
        _setPendingAdmin(_newAdmin);
        emit AdminChanged(_admin(), _newAdmin);
    }

    function updateAdmin() external {
        address _newAdmin = _pendingAdmin();
        require(
            _newAdmin != address(0),
            "Cannot change the admin of a proxy to the zero address"
        );
        require(
            msg.sender == _newAdmin,
            "msg.sender and newAdmin must be the same ."
        );
        _setAdmin(_newAdmin);
        _setPendingAdmin(address(0));
        emit AdminUpdated(_newAdmin);
    }
```

Listing 3.7: DFProxy.sol

However, there is an issue that occurs when there is a change of `admin`. Specifically, the `admin` transfer is a privileged, sensitive matter and current prototype takes a two-stepped approach. The first step indicates the intention to transfer to a new `admin` and the second step requires the initiation from the new `admin` and actually updates the real `admin`.

While this two-stepped approach is appropriate and highly recommended, current implementation unnecessarily restricts the `admin` transfer in the logic contract. Assuming the logic contract takes the very same approach with the exact APIs, the first step indeed proceeds smoothly. But, there is no way for the second step to complete. The reason is that the `updateAdmin()` is always intercepted by the proxy contract, meaning it will not be delegated to the logic contract. Without any chance of receiving the `updateAdmin()`, the logic contract cannot change its `admin`.

Note that a similar issue is also identified in `USRProxy`.

Recommendation Add a new modifier that properly differentiates the caller of `updateAdmin()` and delegates to the logic contract as intended.

```
    function updateAdmin() external ifPendingAdmin {
        address _newAdmin = _pendingAdmin();
        _setAdmin(_newAdmin);
        _setPendingAdmin(address(0));
        emit AdminUpdated(_newAdmin);
    }

    modifier ifPendingAdmin() {
        if (msg.sender == _pendingAdmin()) {
            _;
        } else {
            _fallback();
        }
    }
```

Listing 3.8: DFProxy.sol (revised)

Status This issue has been confirmed. Since current logic contracts do not use any APIs similar to the proxy's admin APIs, the team decided not to address it for the time being.

3.6 Wrapped Collateral Calculation in Assets Balance Check

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DFEngineV2`
- Category: Time and State [10]
- CWE subcategory: N/A

Description

For maximum security, the `USDx` protocol introduces a balance check to ensure the minted total amount of `USDx` is always consistent with total collateral amount. Internally, there are a few contracts, i.e., `DFPoolV2`, `DFFunds`, and `DFCollateral`. The `DFPoolV2` contract keeps all collateral assets that are deposited into the `USDx` protocol; `DFFunds` collects protocol fee to partially reimburse the development and management cost; and `DFCollateral` keeps a copy of wrapped tokens of ingredient stablecoins that have successfully been used to mint `USDx`.

We notice the balance check routine, i.e., `checkUSDXTotalAndColTotal()`, will be invoked whenever there is a change to `USDx totalSupply`. Such occasions include `deposit()`, `destroy()`, and `oneClickMinting()`. A close examination with the routine shows that the given token for collateral calculation (line 324) should be the wrapped tokens of ingredient stablecoins, not the actual stablecoins. A similar issue has been identified in the design document of `USDx`.

```
319     function checkUSDXTotalAndColTotal() internal view {
320         address[] memory _tokens = dfStore.getMintedTokenList();
321         address _dfCol = dfCol;
322         uint _colTotal;
323         for (uint i = 0; i < _tokens.length; i++) {
324             _colTotal = add(_colTotal, IDSToken(_tokens[i]).balanceOf(_dfCol));
325         }
326         uint _usdxTotalSupply = usdxToken.totalSupply();
327         require(_usdxTotalSupply <= _colTotal,
328                 "checkUSDXTotalAndColTotal : Amount of the usdx will be greater than
                    collateral.");
329         require(_usdxTotalSupply == dfStore.getTotalCol(),
330                 "checkUSDXTotalAndColTotal : Usdx and total collateral are not equal.");
331     }
```

Listing 3.9: DFPoolV2.sol

Recommendation Revise the above logic by replacing `IDSToken` with `IDSWrappedToken` in line 324.

```
319     function checkUSDXTotalAndColTotal() internal view {
320         address[] memory _tokens = dfStore.getMintedTokenList();
321         address _dfCol = dfCol;
322         uint _colTotal;
323         for (uint i = 0; i < _tokens.length; i++) {
324             _colTotal = add(_colTotal, IDSWrappedToken(_tokens[i]).balanceOf(_dfCol));
325         }
326         uint _usdxTotalSupply = usdxToken.totalSupply();
327         require(_usdxTotalSupply <= _colTotal,
328                 "checkUSDXTotalAndColTotal : Amount of the usdx will be greater than
                    collateral.");
329         require(_usdxTotalSupply == dfStore.getTotalCol(),
330                 "checkUSDXTotalAndColTotal : Usdx and total collateral are not equal.");
331     }
```

Listing 3.10: DFPoolV2.sol

Status This issue has been fixed and the related documentation has been updated accordingly.

3.7 Improved Precision in Interest Calculation

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: DFPoolV2
- Category: Numeric Errors [13]
- CWE subcategory: CWE-192 [3]

Description

The combination of USDx and USR introduces an interesting offering that benefits USDx holders. The interests generated from USR are derived from the integration of external lending platforms (e.g., Compound and Aave) by depositing the locked collateral in USDx.

The interests from these lending platforms need to be accurately calculated. Note that the lack of float support in Solidity makes the the interest calculation unusually complicated. In particular, the _getInterestByXToken() routine is responsible for inquiring the current balance from the integrated lending platforms and possible origination fee associated with their withdrawal. And any gain from the principle is considered as interests distributable to all stakers.

During this process, a rounding issue may occur. And if there is a rounding issue, it is always preferable to allow the calculation lean towards the pool. In other words, suppose the pool needs to pay a staker certain amount of interest. For the interest, the system charges a percentage. Because of possible rounding issue, we may intend to round-up the calculation so that the system pays the staker slightly less (but with only an extremely small difference).

```
    function getInterestByXToken(address _xToken) public returns (address, uint256) {

        address _token = IDSWrappedToken(_xToken).getSrcERC20();
        uint256 _xBalance = IDSWrappedToken(_xToken).changeByMultiple(getUnderlying(
            _token));
        uint256 _xPrincipal = IERC20(_xToken).balanceOf(dfcol);
        return (_token, _xBalance > _xPrincipal ? sub(_xBalance, _xPrincipal) : 0);
    }

    function getUnderlying(address _underlying) public returns (uint256) {
        address _dToken = IDTokenController(dTokenController).getDToken(_underlying);
        if (_dToken == address(0))
            return 0;

        (, uint256 _exchangeRate, , uint256 _feeRate,) = IDToken(_dToken).getBaseData();

        uint256 _grossAmount = rmul(IERC20(_dToken).balanceOf(address(this)),
            _exchangeRate);
        return sub(_grossAmount, rmul(_grossAmount, _feeRate));
```

```
    }
```

Listing 3.11: DFPoolV2.sol

In our case, we note that the underlying balance from lending platforms is `sub(_grossAmount, rmul(_grossAmount, _feeRate))`. If there is a rounding issue, such calculation essentially rounds up the calculation to make it unnecessarily (slightly) larger than the actual balance (after the withdrawal). With that, we suggest to perform a round-down calculation, i.e., `rmul( _grossAmount, sub(1e18, _feeRate))`.

Recommendation Improve the precision in the interest calculation as follows:

```
    function getInterestByXToken(address _xToken) public returns (address, uint256) {

        address _token = IDSWrappedToken(_xToken).getSrcERC20();
        uint256 _xBalance = IDSWrappedToken(_xToken).changeByMultiple(getUnderlying(
            _token));
        uint256 _xPrincipal = IERC20(_xToken).balanceOf(dfcol);
        return (_token, _xBalance > _xPrincipal ? sub(_xBalance, _xPrincipal) : 0);
    }

    function getUnderlying(address _underlying) public returns (uint256) {
        address _dToken = IDTokenController(dTokenController).getDToken(_underlying);
        if (_dToken == address(0))
            return 0;

        (, uint256 _exchangeRate, , uint256 _feeRate,) = IDToken(_dToken).getBaseData();

        uint256 _grossAmount = rmul(IERC20(_dToken).balanceOf(address(this)),
            _exchangeRate);
        return rmul( _grossAmount, sub(1e18, _feeRate));
    }
```

Listing 3.12: DFPoolV2.sol

Status This issue has been fixed by this particular commit: c28ff61ef39d5bc0212603ca33184146fba77f33.

3.8 Better Allocation of Early Bird Bonus

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `InterestProvider`
- Category: Business Logics [11]
- CWE subcategory: CWE-708 [7]

Description

`USDx` is intended to be the underlying token encapsulated in `USR`. By actively depositing the backing ingredient stablecoins into external lending platforms, the `USDx` protocol can generate continuous interest stream that funds the saving rate in `USR`.

With pro-active deposits into external lending platforms, it is possible for the integration to generate interests before any `USR` token is minted. In this case, the first staker has early-bird bonus in capturing whatever interests generated so far. Note that it could be recurring if all minted `USR` tokens are redeemed and then the subsequent earliest staker has similar early-bird bonus.

An alternative approach would be to enable better allocation of these interests generated before any staking. As an example, we can internally record these interests in the contract and then set them aside to cover maintenance and/or development cost at the very moment when the first staking occurs.

Recommendation Develop an alternative strategy to allocate early-bird bonus, instead of simply favoring the first staker.

Status This issue has been confirmed. The team considers it part of the design and thus appropriate to leave it as is (unless the design assumption will be changed in the future).

3.9 Reentrancy Risks With ERC777 Tokens

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `USR`
- Category: Concurrency [12]
- CWE subcategory: CWE-663 [6]

Description

The `USR` smart contract inherits from `ERC20Exchangeable`, which is a generic implementation of encapsulating a given underlying token and making the underlying token exchangeable with the wrapping one. The conversion between the underlying token and the wrapping one is based on the calculated `exchangeRate()`. By doing so, it can effectively take into account the dynamically-generated interest stream (that lively updates the `underlyingBalance()` – line 117) from the integration of external lending platforms.

```
113     function exchangeRate() public returns (uint256) {
114         uint256 totalSupply = totalSupply();
115         return
116             totalSupply > 0
```

```
117                 ? underlyingBalance().rdiv(totalSupply)
118                 : SafeRatioMath.base();
119     }
```

Listing 3.13: USR.sol

We highlight that the generic implementation of `ERC20Exchangeable` lacks re-entrancy prevention. If the underlying token faithfully implements the ERC777-like standard, then `ERC20Exchangeable` and its inheritance are vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer` or `transferFrom` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering a `tokensToSend` and `tokensReceived` hooks. Consequently, any `transfer` or `transferFrom` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, it can be exploited to manipulate the `exchangeRate()`. Using the `redeemUnderlying()` as an example, the above hook can be planted in `underlyingToken.safeTransfer(msg.sender, underlying)` (line 104) after the wrapping tokens were burned, but before the actual transfer of the underlying token occurs. By doing so, we can effectively reduce `totalSupply` (used for `exchangeRate()` calculation in 117), thus lifting the calculated `exchangeRate()`. With a higher `exchangeRate()`, the re-entered `redeemUnderlying()` is able to redeem more underlying tokens. It can be repeated to exploit this vulnerability for gains, just like earlier Uniswap/imBTC hack [21].

```
85      function redeemUnderlying(address account, uint256 underlying)
86          public
87          whenNotPaused
88          returns (bool)
89      {
90          uint256 fee = calcAdditionalFee(this.redeem.selector, underlying);
91          uint256 totalUnderlying = underlying.add(fee);
92          uint256 amount = totalUnderlying.rdivup(exchangeRate());
93
94          if (account == msg.sender) {
95              _burn(account, amount);
96          } else {
97              _burnFrom(account, amount);
98          }
99
100         // Allow sub contract to do something
101         checkRedeem(totalUnderlying);
102
103         transferFee(address(this), fee);
104         underlyingToken.safeTransfer(msg.sender, underlying);
105
106         return true;
```

```
    }
```

Listing 3.14: USR.sol

Our analysis shows that throughout the `USR` codebase, there are a few entries that need the reentrancy presentation: `mint()`, `redeem()`, and `redeemUnderlying()`. Fortunately, the `USR` contract is designed to work with `USDx` as the underlying token and `USDx` is not an ERC777 token. As a result, the deployment of this code base will not pose any issue from reentrancy. But the reentrancy risk and its notorious history bring up the necessity to implement effective reentrancy prevention in current codebase.

Recommendation Apply necessary reentrancy prevention by adding the following modifier to the above functions.

```
    bool internal locked;
    modifier noReentrancy() {
        require(!locked, "Reentrant call.");
        locked = true;
        _;
        locked = false;
    }

    function redeemUnderlying(address account, uint256 underlying)
        public
        whenNotPaused
        noReentrancy
        returns (bool)
    {
        uint256 fee = calcAdditionalFee(this.redeem.selector, underlying);
        uint256 totalUnderlying = underlying.add(fee);
        uint256 amount = totalUnderlying.rdivup(exchangeRate());

        if (account == msg.sender) {
            _burn(account, amount);
        } else {
            _burnFrom(account, amount);
        }

        // Allow sub contract to do something
        checkRedeem(totalUnderlying);

        transferFee(address(this), fee);
        underlyingToken.safeTransfer(msg.sender, underlying);

        return true;
    }
```

Listing 3.15: USR.sol (revised)

Status This issue has been addressed by this particular commit: 3d63e816360f7f8a74ade1eb16f1dcfa66d765d3.

3.10 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0;` instead of `pragma solidity >=0.6.0;`.

In addition, there is a known compiler issue that in all 0.5.x solidity prior to `Solidity 0.5.17`. Specifically, a private function can be overridden in a derived contract by a private function of the same name and types. Fortunately, there is no overriding issue in this code, but we still recommend using `Solidity 0.5.17` or above.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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.

4 Conclusion

In this audit, we thoroughly analyzed the design and implementation of USDx and USR. The system presents a unique offering in current stablecoin ecosystem and we are impressed by the design and implementation. 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [16, 17, 18, 19, 22].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [23] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-192: Integer Coercion Error. https://cwe.mitre.org/data/definitions/192.html.

[4] MITRE. CWE-284: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html.

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

[6] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[7] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

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

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

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

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

[12] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[13] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

[16] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[17] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[18] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[19] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[21] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[22] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[23] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.