SMART CONTRACT AUDIT REPORT

for

DEFIDOLLAR

Prepared By: Shuxiao Wang

Hangzhou, China
August 18, 2020

Document Properties

Client	DefiDollar
Title	Smart Contract Audit Report
Target	DefiDollar
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	August 18, 2020	Xuxian Jiang	Final Release
1.0-rc3	August 15, 2020	Xuxian Jiang	Release Candidate #3
1.0-rc2	August 12, 2020	Xuxian Jiang	Release Candidate #2
1.0-rc1	August 6, 2020	Xuxian Jiang	Release Candidate #1
0.1	August 3, 2020	Huaguo Shi	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 **DefiDollar** smart contract source code, 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 DefiDollar

DefiDollar (DUSD) is an index of stable coins that uses DeFi primitives to stay near the dollar mark more robustly than each individual stable coin. The vision behind DUSD is to provide an avenue for diversifying users' crypto-dollars positions, and to dampen the potentially disastrous effects of a particular stable coin such as Tether failing (partially or completely) from its peg.

The basic information of DefiDollar is as follows:

Table 1.1: Basic Information of DefiDollar

Item	Description
Issuer	DefiDollar
Website	https://www.defidollar.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 18, 2020

In the following, we show the repository of the reviewed code used in this audit. We need to point out that DefiDollar assumes a trusted oracle with timely market price feeds and the oracle itself is not part of this audit. This is the commit ID the audit is based on:

- https://github.com/defidollar/defidollar-core (77dcbab)

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

- https://github.com/defidollar/defidollar-core (c0ae0bb)

1.2 About PeckShield

PeckShield Inc. [18] 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 the OWASP Risk Rating Methodology [13]:

- 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) [12], 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 DefiDollar 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	3	■■■
Medium	4	■■■■
Low	2	■■
Informational	2	■■
Total	11	

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

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incompatibility With Deflationary Tokens	Time and State	Confirmed
PVE-002	High	Possible Use of Outdated Price Feeds	Business Logics	Confirmed
PVE-003	Low	Improved Sanity Checks in whitelistTokens()	Error Conditions, Return Values, Status Codes	Fixed
PVE-004	Info.	Simplified Execution Logic in mint()	Error Conditions, Return Values, Status Codes	Fixed
PVE-005	High	Possible Unaccounted Staking Income	Business Logics	Fixed
PVE-006	Medium	Locked Non-SystemCoins-Assets From Yield-Farming	Business Logics	Confirmed
PVE-007	Medium	Possible Front-Running in syncSystem()	Time and State	Fixed
PVE-008	Info.	Code Simplification in getReward()	Business Logics	Fixed
PVE-009	Medium	Redemption Fee Miscalculation in dusdToUsd()	Business Logics	Fixed
PVE-010	Low	Suggested Padding in Logic Contracts	Coding Practices	Fixed
PVE-011	High	Inaccurate Delta Calculation During mint()/redeem()	Business Logics	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Incompatibility With Deflationary Tokens

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `CurveSusdPeak`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [3]

Description

The `CurveSusdPeak` contract or `peak` is designed to work with a target `Curve` pool and exposes important interfaces to `mint/redeem` the stable coin index, i.e., `DUSD`. A user may deposit certain amount of supported `SystemCoins` or `CurvePoolTokens` into it to mint `DUSDs`. Later on, the same user can redeem her `DUSDs` to get the `SystemCoins` or `CurvePoolTokens` back. For the above two operations, i.e., `mint` and `redeem`, `CurveSusdPeak` provides low-level routines to transfer assets into or out of the peak (see the code snippet below). These asset-transferring routines work as expected with standard ERC20 tokens: namely the peak's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    /**
     * @dev Mint DUSD
     * @param inAmounts Exact inAmounts in the same order as required by the curve pool
     * @param minDusdAmount Minimum DUSD to mint, used for capping slippage
     */
    function mint(
        uint[] calldata inAmounts,
        uint minDusdAmount
    ) external
        returns (uint dusdAmount)
    {
        address[N_COINS] memory coins = underlyingCoins;
        uint[N_COINS] memory pool_sizes;

        for (uint i = 0; i < N_COINS; i++) {
```

```
            pool_sizes[i] = curve.balances(i);
            if (inAmounts[i] == 0) {
                continue;
            }
            IERC20(coins[i]).safeTransferFrom(msg.sender, address(this), inAmounts[i]);
        }

        LPShareInfo memory info;
        info.old_lp_amount = curveToken.balanceOf(address(this));
        info.old_lp_supply = curveToken.totalSupply();

        curveDeposit.add_liquidity(inAmounts, 0);

        info.new_lp_amount = curveToken.balanceOf(address(this));
        info.new_lp_supply = curveToken.totalSupply();

        uint[] memory delta = new uint[](N_COINS);
        for (uint i = 0; i < N_COINS; i++) {
            delta[i] = _calcDepositDelta(info, pool_sizes[i], inAmounts[i]);
        }
        return core.mint(delta, minDusdAmount, msg.sender);
    }
```

Listing 3.1: peaks/curve/CurveSusdPeak.sol

```
    /**
    * @dev Burn DUSD
    * @param outAmounts Exact outAmounts in the same order as required by the curve pool
    * @param maxDusdAmount Max DUSD to burn, used for capping slippage
    */
    function redeem(
        uint[] calldata outAmounts,
        uint maxDusdAmount
    )
        external
        returns(uint dusdAmount)
    {
        uint[N_COINS] memory pool_sizes;
        for (uint i = 0; i < N_COINS; i++) {
            pool_sizes[i] = curve.balances(i);
        }

        LPShareInfo memory info;
        info.old_lp_amount = curveToken.balanceOf(address(this));
        info.old_lp_supply = curveToken.totalSupply();

        curveDeposit.remove_liquidity_imbalance(outAmounts, MAX);

        info.new_lp_amount = curveToken.balanceOf(address(this));
        info.new_lp_supply = curveToken.totalSupply();

        address[N_COINS] memory coins = underlyingCoins;
        uint[] memory delta = new uint[](N_COINS);
```

```
        for (uint i = 0; i < N_COINS; i++) {
            IERC20(coins[i]).safeTransfer(msg.sender, outAmounts[i]);
            delta[i] = _calcWithdrawDelta(info, pool_sizes[i], outAmounts[i]);
        }
        return core.redeem(delta, maxDusdAmount, msg.sender);
    }
```

Listing 3.2: peaks/curve/CurveSusdPeak.sol

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer or transferFrom. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `mint` and `redeem`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of `CurveSusdPeak` and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly 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` or `transferFrom` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider 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 DefiDollar for indexing. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary, which is out of DefiDollar's control.

Recommendation Apply necessary mitigation mechanisms to regulate non-compliant or unnecessarily-extended ERC20 tokens.

3.2 Possible Use of Outdated Price Feeds

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `UpgradableProxy`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

Throughout the entire DefiDollar system, the `Core` contract performs the actual `mint/redeem` operations, pulls latest oracle feeds, and distributes protocol income to stakers. All these operations rely on latest price feeds for precise measurement and calculation of system-wide assets or `totalAssets` (that is needed to calculate the actual amount during `mint` or `redeem` for example). We notice that the contract exhibits a public function named `syncSystem()` to allow anyone to pull latest oracle feeds. Specifically, `syncSystem()` calls `_updateFeed()` to get the latest prices of supported `systemCoins` (line 190).

```
182     /**
183     * @notice Pull prices from the oracle and update system stats
184     * @dev Anyone can call this
185     */
186     function syncSystem()
187         external
188         checkAndNotifyDeficit
189     {
190         _updateFeed();
191         totalAssets = totalSystemAssets();
192     }
```

Listing 3.3: Core.sol

Though all these operations rely on latest price feeds, many of them do not always retrieve the latest price feeds. As an example, the `mint` operation may not get real-time prices from oracle (see the code snippets below at lines 116 122): the calculation of asset difference is performed with a cached price feeds. Notice that the use of outdated prices likely lead to inaccurate measurement of system-wide assets. Other affected operations include `redeem()`, `rewardDistributionCheckpoint()` and `lastPeriodIncome()`. We consider the freshness of these price feeds critical even though their guarantee may introduce additional gas cost.

```
94      /**
95      * @notice Mint DUSD
96      * @dev Only whitelisted peaks can call this function
97      * @param delta Delta of system coins added to the system through a peak
98      * @param minDusdAmount Min DUSD amount to mint. Used to cap slippage
99      * @param account Account to mint DUSD to
100     * @return dusdAmount DUSD amount actually minted
101     */
102     function mint(
103         uint[] calldata delta,
104         uint minDusdAmount,
105         address account
106     )   external
107         checkAndNotifyDeficit
108         returns (uint dusdAmount)
109     {
110         Peak memory peak = peaks[msg.sender];
111         require(
112             peak.state == PeakState.Active,
113             "Peak is inactive"
114         );
```

```
        uint usdDelta;
        SystemCoin[] memory coins = systemCoins;
        for (uint i = 0; i < peak.systemCoinIds.length; i++) {
            SystemCoin memory coin = coins[peak.systemCoinIds[i]];
            usdDelta = usdDelta.add(
                delta[i].mul(coin.price).div(coin.precision)
            );
        }
        dusdAmount = usdToDusd(usdDelta);
        require(dusdAmount >= minDusdAmount, "They see you slippin");
        dusd.mint(account, dusdAmount);
        totalAssets = totalAssets.add(usdDelta);
        emit Mint(account, dusdAmount);
    }
```

Listing 3.4: Core.sol

Recommendation Ensure the freshness of price feeds. To mitigate possible gas cost, an alternative is to implement the `poke` mechanism in the oracle such that it dynamically notifies the arrival of a new price feed. With that, there is no need to always invoke gas-heavy `syncSystem()` routine before the calculation of `totalAssets`.

3.3 Improved Sanity Checks in whitelistTokens()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Core`
- Category: Error Conditions, Return Values, Status Codes [11]
- CWE subcategory: CWE-391 [4]

Description

In the `Core` contract, the `whitelistTokens()` function allows the owner to set up or append the supported `systemCoins`. Notice that this function has three arguments, i.e., `tokens`, `decimals`, and `initialPrices`, and they are all dynamic arrays. It is necessary to ensure that they have the same length. Moreover, it is also needed to ensure there is no duplicate token in the provided argument and each indicates a new new token for `systemCoins` inclusion.

```
    /**
    * @notice Whitelist new tokens supported by the peaks.
    * These are vanilla coins like DAI, USDC, USDT etc.
    * @dev onlyOwner ACL is provided by the whitelistToken call
    * @param tokens Token addresses to whitelist
    * @param decimals Token Precision
    * @param initialPrices Intialize prices akin to retieving from an oracle
```

```
    */
    function whitelistTokens(
        address[] calldata tokens,
        uint[] calldata decimals,
        uint[] calldata initialPrices
    ) external
        onlyOwner
    {
        for (uint i = 0; i < tokens.length; i++) {
            _whitelistToken(tokens[i], decimals[i], initialPrices[i]);
        }
    }
```

Listing 3.5: Core.sol

Recommendation Add necessary sanity checks to the `whitelistTokens()` routine. It is also suggested to ensure the provided `decimals` no larger than 18 – a restriction inherently assumed by the Curve protocol.

```
    /**
    * @notice Whitelist new tokens supported by the peaks.
    * These are vanilla coins like DAI, USDC, USDT etc.
    * @dev onlyOwner ACL is provided by the whitelistToken call
    * @param tokens Token addresses to whitelist
    * @param decimals Token Precision
    * @param initialPrices Intialize prices akin to retieving from an oracle
    */
    function whitelistTokens(
        address[] calldata tokens,
        uint[] calldata decimals,
        uint[] calldata initialPrices
    ) external
        onlyOwner
    {
        require(
            tokens.length == decimals.length && tokens.length == initialPrices.length,
            "Invalid system coins"
        );

        for (uint i = 0; i < tokens.length; i++) {
            _checkExistToken(tokens[i]);
            _whitelistToken(tokens[i], decimals[i], initialPrices[i]);
        }
    }

    function _checkExistToken(address token)
        internal
    {
        SystemCoin[] memory coins = systemCoins;
        for(uint i = 0; i < coins.length; i++) {
            SystemCoin memory coin = coins[i];
            require(token != coin.token, "token already existed!");
```

```
        }
    }

    function _whitelistToken(address token, uint decimals, uint initialPrice)
        internal
    {
        require(decimals > 0 && decimals <= 18, "Using a 0 decimal coin can break the
            system");
        systemCoins.push(SystemCoin(token, 10 ** decimals, initialPrice));
        emit TokenWhiteListed(token);
    }
```

Listing 3.6: Core.sol

3.4 Simplified Execution Logic in mint()

- ID: PVE-004
- Severity: Infomational
- Likelihood: N/A
- Impact: N/A
- Target: Core
- Category: Error Conditions, Return Values, Status Codes [11]
- CWE subcategory: CWE-391 [4]

Description

The whitelisted `peaks` (e.g., `CurveSusdPeak`) can interact with the `Core` contract to mint the stable coin index, i.e., `DUSD`. The `mint` can be slightly optimized to speed up the internal `loop` when the condition `delta[i] == 0` is satisfied. This brings a simplified logic with slightly improved performance benefit.

```
    /**
    * @notice Mint DUSD
    * @dev Only whitelisted peaks can call this function
    * @param delta Delta of system coins added to the system through a peak
    * @param minDusdAmount Min DUSD amount to mint. Used to cap slippage
    * @param account Account to mint DUSD to
    * @return dusdAmount DUSD amount actually minted
    */
    function mint(
        uint[] calldata delta,
        uint minDusdAmount,
        address account
    )   external
        checkAndNotifyDeficit
        returns (uint dusdAmount)
    {
        Peak memory peak = peaks[msg.sender];
        require(
```

```
            peak.state == PeakState.Active,
            "Peak is inactive"
        );
        uint usdDelta;
        SystemCoin[] memory coins = systemCoins;
        for (uint i = 0; i < peak.systemCoinIds.length; i++) {
            SystemCoin memory coin = coins[peak.systemCoinIds[i]];
            usdDelta = usdDelta.add(
                delta[i].mul(coin.price).div(coin.precision)
            );
        }
        dusdAmount = usdToDusd(usdDelta);
        require(dusdAmount >= minDusdAmount, "They see you slippin");
        dusd.mint(account, dusdAmount);
        totalAssets = totalAssets.add(usdDelta);
        emit Mint(account, dusdAmount);
    }
```

Listing 3.7: Core.sol

Recommendation Simplify the above `loop` logic by checking `delta[i] == 0`.

```
    /**
    * @notice Mint DUSD
    * @dev Only whitelisted peaks can call this function
    * @param delta Delta of system coins added to the system through a peak
    * @param minDusdAmount Min DUSD amount to mint. Used to cap slippage
    * @param account Account to mint DUSD to
    * @return dusdAmount DUSD amount actually minted
    */
    function mint(
        uint[] calldata delta,
        uint minDusdAmount,
        address account
    )   external
        checkAndNotifyDeficit
        returns (uint dusdAmount)
    {
        Peak memory peak = peaks[msg.sender];
        require(
            peak.state == PeakState.Active,
            "Peak is inactive"
        );
        uint usdDelta;
        SystemCoin[] memory coins = systemCoins;
        for (uint i = 0; i < peak.systemCoinIds.length; i++) {
            if (delta[i] == 0) continue;
            SystemCoin memory coin = coins[peak.systemCoinIds[i]];
            usdDelta = usdDelta.add(
                delta[i].mul(coin.price).div(coin.precision)
            );
        }
        dusdAmount = usdToDusd(usdDelta);
```

```
        require(dusdAmount >= minDusdAmount, "They see you slippin");
        dusd.mint(account, dusdAmount);
        totalAssets = totalAssets.add(usdDelta);
        emit Mint(account, dusdAmount);
    }
```

Listing 3.8: Core.sol

3.5 Possible Unaccounted Staking Income

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `StakeLPToken`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

Staking provided the last line of assurance in ensuring the stable coin index `DUSD` does not deviate wildly from the intended peg. For that, staking users are rewarded by potential incomes from the integrated `peaks`. The staking logic is implemented in the `StakeLPToken` contract and one important functionality is to update protocol income from surrounding peaks.

The logic of updating protocol income is implemented in `updateProtocolIncome()`. Within the function, there is a critical variable named `rewardPerTokenStored`. This variable keeps track of the average reward for each staked `DUSD` token. We note that the variable is only updated via the `updateProtocolIncome()` function: It first calculates the last-period income and then updates `rewardPerTokenStored`.

```
    function updateProtocolIncome() public {
        uint income = core.rewardDistributionCheckpoint();
        rewardPerTokenStored = rewardPerToken(income);
    }
```

Listing 3.9: StakeLPToken.sol

The last-period income is calculated through an exported interface `rewardDistributionCheckpoint()` from the `Core` contract. We notice that it is exported, and is available for anyone to call. Unfortunately, if it is called not by the `StakeLPToken` contract, the last-period income becomes unaccounted to update the variable `rewardPerTokenStored`, leading to unexpected staking loss for all staking users!

```
    function rewardDistributionCheckpoint()
        external
        checkAndNotifyDeficit
        returns(uint)
```

```
    {
        uint supply = dusd.totalSupply();
        totalAssets = totalSystemAssets();
        uint unclaimedRewards;
        if (totalRewards > claimedRewards) {
            unclaimedRewards = totalRewards.sub(claimedRewards);
        }
        uint _totalAssets = totalAssets.sub(unclaimedRewards);
        uint periodIncome;
        if (_totalAssets > supply) {
            periodIncome = _totalAssets.sub(supply);
            totalRewards = totalRewards.add(periodIncome);
        }
        return periodIncome;
    }
```

Listing 3.10: Core.sol

Recommendation Ensure `rewardDistributionCheckpoint()` in the `Core` contract can only be invoked by `StakeLPToken`.

```
    function rewardDistributionCheckpoint()
        external
        onlyStakeLPToken
        checkAndNotifyDeficit
        returns(uint)
    {
        uint supply = dusd.totalSupply();
        totalAssets = totalSystemAssets();
        uint unclaimedRewards;
        if (totalRewards > claimedRewards) {
            unclaimedRewards = totalRewards.sub(claimedRewards);
        }
        uint _totalAssets = totalAssets.sub(unclaimedRewards);
        uint periodIncome;
        if (_totalAssets > supply) {
            periodIncome = _totalAssets.sub(supply);
            totalRewards = totalRewards.add(periodIncome);
        }
        return periodIncome;
    }
```

Listing 3.11: Core.sol (revised)

3.6 Locked Non-SystemCoins-Assets From Yield-Farming

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakeLPToken`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

By design, the `peak` refers to a yield generating protocol through which certain yields are expected after supported `systemCoins` are deposited. Using `CurveSusdPeak` as example, the contract supports a number of `systemCoins`, such as `DAI`, `USDT`, `USDC`, `TUSD`, and `sUSD`. Any user can deposit the above assets and then expect to receive certain rewards.

DefiDollar supports staking to provide the opportunity for staking users to receive all the underlying yield income while taking the risking of ensuring `DUSD`'s peg. We notice that yield-farming may lead to gaining additional tokens that may not necessarily be part of supported `SystemCoins`, such as `CRV` or `COMP`.

From the currently implemented logic, these rewarded yields are locked in `peaks`, and stakers do not get their shares on these non-`systemCoins` yields. The discussion with the DefiDollar team indicates that the `UpgradableProxy`-based architecture allows for flexible new logic contract to be introduced and the built-in `execute()` function in the `UpgradableProxy` contract guarantees the purpose. However, this built-in function is considered omnipotent and could undermine the confidence on the logic implemented in DefiDollar, an issue we will further elaborate in Section ??.

```
    /**
    * Owner can execute operations such as claiming yield farming benefits
    */
    function execute(address _target, bytes memory data) public onlyOwner {
        (bool success, bytes memory returnData) = _target.call(data);
        require(success, string(returnData));
    }
```

Listing 3.12: UpgradableProxy.sol

Recommendation Add necessary helper routines to collect possible yield farming rewards. The distribution of these rewards can be achieved either on-chain or off-chain.

3.7 Possible Front-Running in syncSystem()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CurveSusdPeak`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [3]

Description

In the `Core` contract, the `syncSystem()` function is responsible for updating the latest price feeds from the oracle. The price feeds ensure the accurate measurement and calculation of system-wide assets, i.e., `totalAssets`. This function is designed to be callable by anyone.

The market fluctuations likely introduce dynamics on the current prices of various `systemCoins` and accordingly bring economic implications for staking users. In addition, as mentioned earlier in Section 3.2, multiple functions in the `Core` contract do not use the latest price feeds and are thus susceptible to front-running by a malicious party. In particular, if a recent price update transaction, if submitted but not confirmed/mined yet, likely devalues the state into deficit, front-running can be launched to avoid upcoming staking loss due to the downward price update. In this case, the eventual staking loss, if any, may be inflicted on other innocent staking users.

Recommendation As suggested in Section 3.2, ensure the freshness of price feeds in the measurement and calculation of system-wide assets. And apply common wisdoms in known mitigation schemes, including various restrictions on the slippage limits (already used), allowed gas price ranges, and expiration times etc.

3.8 Code Optimization in getReward()

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StakeLPToken`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [2]

Description

In the `StakeLPToken` contract, the `getReward()` routine is intended to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 94).

```
94     function getReward() public updateReward(msg.sender) {
95         uint reward = earned(msg.sender);
96         if (reward > 0) {
97             rewards[msg.sender] = 0;
98             core.mintReward(msg.sender, reward);
99             emit RewardPaid(msg.sender, reward);
100        }
101    }
```

Listing 3.13: StakeLPToken.sol

Having the modifier `updateReward()`, there is no need to recalculate the earned reward for the caller `msg.sender`. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the `reward` variable (line 95).

```
62     modifier updateReward(address account) {
63         updateProtocolIncome();
64         if (account != address(0)) {
65             rewards[account] = _earned(rewardPerTokenStored, account);
66             userRewardPerTokenPaid[account] = rewardPerTokenStored;
67         }
68         _;
69     }
```

Listing 3.14: StakeLPToken.sol

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to additional gas cost reduction.

```
94     function getReward() public updateReward(msg.sender) {
95         uint reward = rewards[msg.sender];
96         if (reward > 0) {
97             rewards[msg.sender] = 0;
98             core.mintReward(msg.sender, reward);
99             emit RewardPaid(msg.sender, reward);
100        }
101    }
```

Listing 3.15: StakeLPToken.sol

3.9 Redemption Fee Miscalculation in dusdToUsd()

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Core
- Category: Business Logics [10]
- CWE subcategory: CWE-837 [6]

Description

The redemption of the stable coin index DUSD requires accurate conversion from the redemption DUSD amount to the corresponding amount of dollars in USD. This conversion supports the collection of protocol-wide redemption fee that is specified in the system parameter redeemFee in basis points. Specifically, if there is the dollar amount usd for redemption, the final dollar amount the user can receive should be calculated as: usd.mul(redeemFee).div(10000), not usd.mul(10000).div(redeemFee).

However, our analysis shows the redemption fee is miscalculated in the wrong direction. Instead of charging the user for the redemption fee, the user is freely given the redemption fee as bonus! If exploited, a malicious actor can continuously call mint and then redeem to receive the "bonus" of the redemption fee until all assets hold in DefiDollar are eventually drained.

```
    function dusdToUsd(uint _dusd, bool fee)
        public
        view
        returns(uint usd)
    {
        // system is healthy. Pegged at $1
        if (!inDeficit) {
            usd = _dusd;
        } else {
        // system is in deficit, see if staked funds can make up for it
            uint supply = dusd.totalSupply();
            uint perceivedSupply = supply.sub(stakeLPToken.totalSupply());
            // staked funds make up for the deficit
            if (perceivedSupply <= totalAssets) {
                usd = _dusd;
            } else {
                usd = _dusd.mul(totalAssets).div(perceivedSupply);
            }
        }
        if (fee) {
            usd = usd.mul(10000).div(redeemFee);
        }
        return usd;
    }
```

Listing 3.16: Core.sol

Recommendation Properly calculate the due charge when DUSD is being redeemed.

```
function dusdToUsd(uint _dusd, bool fee)
    public
    view
    returns(uint usd)
{
    // system is healthy. Pegged at $1
    if (!inDeficit) {
        usd = _dusd;
    } else {
    // system is in deficit, see if staked funds can make up for it
        uint supply = dusd.totalSupply();
        uint perceivedSupply = supply.sub(stakeLPToken.totalSupply());
        // staked funds make up for the deficit
        if (perceivedSupply <= totalAssets) {
            usd = _dusd;
        } else {
            usd = _dusd.mul(totalAssets).div(perceivedSupply);
        }
    }
    if (fee) {
        usd = usd.mul(redeemFee).div(10000);
    }
    return usd;
}
```

Listing 3.17: Core.sol (revised)

3.10 Suggested Padding in Logic Contracts

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `multiple contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [5]

Description

As mentioned earlier, DefiDollar adopts a proxy-based approach to enable flexible updates of new versions of logic contracts. However, the proxy-based approach also comes with a few caveats. Since DefiDollar's upgradeability is achieved using inherited storage, one particular caveat is that this approach relies on making the logic contract to incorporate the storage structure required by the proxy. In other words, both the proxy (i.e., `UpgradableProxy`) and the logic contract inherit the same storage structure to ensure that both adhere to storing the necessary proxy state variables. In the case of multiple inheritance, the order in which parent contracts are declared on the child contracts also determines how their storages are laid out.

To ensure the consistency of inherited storage layouts while still maintaining necessary flexibility of extending logic contracts (including both parent and child contracts), it is suggested to add necessary padding at the end of local variables for each contract.

Using the `Initializable` contract as an example, current prototype does not reserve necessary padding space for future extensions.

```
pragma solidity 0.5.17;

contract Initializable {
    bool initialized = false;

    modifier notInitialized() {
        require(!initialized, "already initialized");
        initialized = true;
        _;
    }
}
```

Listing 3.18: Initializable.sol

A suggested approach is to reserve padding storage (e.g., `______gap` as shown in the code snippets below) at the end of contract. The reserved padding space can be used by future upgrades and its size depends on estimated need in the new versions of logic contracts. A typical size of 50 might suffice to meet most needs for relatively stable logic contracts.

```
pragma solidity 0.5.17;
```

```
contract Initializable {
    bool initialized = false;

    modifier notInitialized() {
        require(!initialized, "already initialized");
        initialized = true;
        _;
    }

    // Reserved storage space to allow for layout changes in the future.
    uint256[50] private ______gap;
}
```

Listing 3.19: Initializable.sol (revised)

Recommendation Reserve necessary storage space in logic contracts to allow for their future layout changes.

3.11 Inaccurate Delta Calculation During mint()/redeem()

- ID: PVE-011
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `CurveSusdPeak`
- Category: Business Logics [10]
- CWE subcategory: CWE-837 [6]

Description

The minting of the stable coin index `DUSD` requires accurate calculation of `Curve`'s pool balance difference due to the deposit. However, the calculation of `Curve`'s balance is rather complicated as it is governed by the pricing curve behind `Curve`. Moreover, the trading fee as well as protocol-related administration fee (currently 0) also play a role in the balance calculation formula.

The formula is delicate and intrinsic to the `Curve`. It is our suggestion not to mock its execution for external balance calculation. Instead, take necessary steps to read the balances before and after the deposits and then calculate the balance difference (due to the deposit) according to the pricing curve.

Our analysis shows that current calculation in `_calcDepositDelta()` to derive the balance difference does not reflect the actual difference inside the `Curve` protocol. The simple addition between `old_pool_size` and `amount` (at line 194) does not take into consideration other factors, such as the protocol fee in `Curve` .

```
181    function _calcDepositDelta(
```

```
        LPShareInfo memory info,
        uint old_pool_size,
        uint amount
    )
        internal
        pure
        returns (uint /* delta */)
    {
        uint old_balance;
        if (info.old_lp_supply > 0) {
            old_balance = old_pool_size.mul(info.old_lp_amount).div(info.old_lp_supply);
        }
        uint new_balance = old_pool_size.add(amount).mul(info.new_lp_amount).div(info.
            new_lp_supply);
        return new_balance.sub(old_balance);
    }
```

Listing 3.20: CurveSusdPeak.sol

If we examine the logic in `Curve`, especially the relevant `add_liquidity()` routine, the new balance is calculated on `self.balances[i] = new_balances[i] - fees[i] * _admin_fee / FEE_DENOMINATOR` (line 260), where both `fees` and `_admin_fee` are configurable parameters during the pool initialization.

```
216 @public
217 @nonreentrant('lock')
218 def add_liquidity(amounts: uint256[N_COINS], min_mint_amount: uint256):
219     # Amounts is amounts of c-tokens
220     assert not self.is_killed

222     tethered: bool[N_COINS] = TETHERED
223     use_lending: bool[N_COINS] = USE_LENDING
224     fees: uint256[N_COINS] = ZEROS
225     _fee: uint256 = self.fee * N_COINS / (4 * (N_COINS - 1))
226     _admin_fee: uint256 = self.admin_fee

228     token_supply: uint256 = self.token.totalSupply()
229     rates: uint256[N_COINS] = self._current_rates()
230     # Initial invariant
231     D0: uint256 = 0
232     old_balances: uint256[N_COINS] = self.balances
233     if token_supply > 0:
234         D0 = self.get_D_mem(rates, old_balances)
235     new_balances: uint256[N_COINS] = old_balances

237     for i in range(N_COINS):
238         if token_supply == 0:
239             assert amounts[i] > 0
240         # balances store amounts of c-tokens
241         new_balances[i] = old_balances[i] + amounts[i]

243     # Invariant after change
244     D1: uint256 = self.get_D_mem(rates, new_balances)
```

```
    assert D1 > D0

    # We need to recalculate the invariant accounting for fees
    # to calculate fair user's share
    D2: uint256 = D1
    if token_supply > 0:
        # Only account for fees if we are not the first to deposit
        for i in range(N_COINS):
            ideal_balance: uint256 = D1 * old_balances[i] / D0
            difference: uint256 = 0
            if ideal_balance > new_balances[i]:
                difference = ideal_balance - new_balances[i]
            else:
                difference = new_balances[i] - ideal_balance
            fees[i] = _fee * difference / FEE_DENOMINATOR
            self.balances[i] = new_balances[i] - fees[i] * _admin_fee / FEE_DENOMINATOR
            new_balances[i] -= fees[i]
        D2 = self.get_D_mem(rates, new_balances)
    else:
        self.balances = new_balances

    # Calculate, how much pool tokens to mint
    mint_amount: uint256 = 0
    if token_supply == 0:
        mint_amount = D1  # Take the dust if there was any
    else:
        mint_amount = token_supply * (D2 - D0) / D0

    assert mint_amount >= min_mint_amount, "Slippage screwed you"

    # Take coins from the sender
    for i in range(N_COINS):
        if tethered[i] and not use_lending[i]:
            USDT(self.coins[i]).transferFrom(msg.sender, self, amounts[i])
        else:
            assert_modifiable(
                cERC20(self.coins[i]).transferFrom(msg.sender, self, amounts[i]))

    # Mint pool tokens
    self.token.mint(msg.sender, mint_amount)

    log.AddLiquidity(msg.sender, amounts, fees, D1, token_supply + mint_amount)
```

Listing 3.21: stableswap.vy

Note that the mint counterpart, i.e., `redeem()`, also shares the very same issue.

Recommendation Read the pool balances before and after the `mint` and `redeem` and correctly calculate their difference.

3.12 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 0.5.12;` instead of `pragma solidity ^0.5.12;`.

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 DefiDollar design and implementation. The proposed system for stable coin index presents a unique innovation and we are really impressed by the overall 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 [14, 15, 16, 17, 19].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [20] 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[4] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

[5] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[6] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

[11] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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

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

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

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

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