SMART CONTRACT AUDIT REPORT

for

DCNTRL Network

Prepared By: Xiaomi Huang

PeckShield

July 30, 2023

Document Properties

Client	DCNTRL Network
Title	Smart Contract Audit Report
Target	DCNTRL Network
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 30, 2023	Xuxian Jiang	Final Release
1.0-rc1	July 23, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the DCNTRL Network protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About DCNTRL Network

DCNTRL Network is a decentralized borrowing protocol that allows you to draw low-interest loans against the native asset used as collateral. Loans are paid out in USDEFI (a USD pegged stablecoin) and need to maintain a minimum (configurable) collateral ratio. In addition to the collateral, the loans are secured by a stability pool containing USDEFI and by borrowers collectively acting as guarantors of last resort. Initially forked from Liquity, DCNTRL Network makes a number of extensions by supporting customized tokenomics and fee structure, and allowing for governance-configurable risk parameters. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of DCNTRL Network

Item	Description
Name	DCNTRL Network
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 30, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that DCNTRL Network assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/tenfinance/Decntral-contracts.git (a10e877)

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

- https://github.com/tenfinance/Decntral-contracts.git (TBD)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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 [9]:

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

Table 1.3: The Full Audit Checklist

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

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

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

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

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

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

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 Logic	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.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the DCNTRL Network protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	4	■■■■
Informational	2	■■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key DCNTRL Network Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect TroveLiquidated Event in TroveManager	Business Logic	Confirmed
PVE-002	Low	Revisited Caller Validation in SortedTroves::insert()	Security Features	Confirmed
PVE-003	Informational	Enhanced Oracle Status in PriceFeed::_fetchPrice()	Business Logic	Confirmed
PVE-004	Low	Improved Trove Close Logic in TroveManager	Business Logic	Confirmed
PVE-005	Low	Improved Validation in USDEFIToken/DCNXToken::permit()	Coding Practices	Confirmed
PVE-006	Informational	Simplified Logic in Unipool::claimReward()	Business Logic	Confirmed

Besides the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Incorrect TroveLiquidated Event in TroveManager

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

Description

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

In the following, we use the TroveManager contract as an example. This contract has public functions that are used to manage current troves. While examining the TroveLiquidated events, we notice the emitted information needs to be improved.

Specifically, when a trove is liquidated during the recovery mode, the _liquidateRecoveryMode() routine will be invoked. By design, if there is USDEFI in the stability pool, the liquidation will only offset, with no redistribution, but at a capped rate of 1.1 and only if the whole debt can be liquidated. In the meantime, the remainder due to the capped rate will be claimable as collateral surplus. With that, the TroveLiquidated event needs to reflect the actual debt/collateral being liquidated. The current event logic shows the right debt amount (singleLiquidation.entireTroveDebt), but not the collateral amount (singleLiquidation.collToSendToSP). The exact collateral amount being liquidated is singleLiquidation.entireTroveColl - singleLiquidation.collSurplus (line 417).

```
404         ...
405         if ((_ICR >= MCR) && (_ICR < _TCR) && (singleLiquidation.entireTroveDebt <=
                _USDEFIInStabPool)) {
```

```
            _movePendingTroveRewardsToActivePool(_activePool, _defaultPool, vars.pendingDebtReward, vars.pendingCollReward);
            assert(_USDEFIInStabPool != 0);

            _removeStake(_borrower);
            singleLiquidation = _getCappedOffsetVals(singleLiquidation.entireTroveDebt, singleLiquidation.entireTroveColl, _price);

            _closeTrove(_borrower, Status.closedByLiquidation);
            if (singleLiquidation.collSurplus > 0) {
                collSurplusPool.accountSurplus(_borrower, singleLiquidation.collSurplus);
            }

            emit TroveLiquidated(_borrower, singleLiquidation.entireTroveDebt, singleLiquidation.collToSendToSP, TroveManagerOperation.liquidateInRecoveryMode);
            emit TroveUpdated(_borrower, 0, 0, 0, TroveManagerOperation.liquidateInRecoveryMode);

        }
```

Listing 3.1: `TroveManager::_liquidateRecoveryMode()`

Recommendation Properly emit the above `TroveLiquidated` event with the right debt/collateral amount.

Status This issue has been confirmed.

3.2 Revisited Caller Validation in SortedTroves::insert()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SortedTroves`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

The `DCNTRL Network` protocol has a core `SortedTroves` contract to maintain a sorted doubly linked list of active `troves` in descending order accordingly to their nominal individual collateral ratios (`NICR`). Our analysis shows that the key `insert()` operation is expected to be called only from the `borrowerOperations` contract.

To elaborate, we show below the related `insert()` routine, which has a rather straightforward logic in inserting a `trove` node into the list while maintaining the proper descending list based on its `NICR`. It comes to our attention that the caller is validated to be from either `borrowerOperations` or `TroveManager`. However, the current `TroveManager` logic will only call the `reInsert()` function to re-insert the node at a new position (based on its new `NICR`), not the `insert()` routine.

```
    function insert (address _id, uint256 _NICR, address _prevId, address _nextId)
        external override {
        ITroveManager troveManagerCached = troveManager;

        _requireCallerIsBOorTroveM(troveManagerCached);
        _insert(troveManagerCached, _id, _NICR, _prevId, _nextId);
    }
```

Listing 3.2: `SortedTroves::insert()`

Recommendation Revise the above caller-validating logic inside the `insert()` routine.

Status This issue has been confirmed.

3.3 Enhanced Oracle Status in PriceFeed::_fetchPrice()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PriceFeed`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

The `DCNTRL Network` protocol is unique in supporting dual oracles, which necessitate the examination of current oracle states. In total, there are five different oracle states, i.e., `chainlinkWorking`, `usingTellorChainlinkUntrusted`, `bothOraclesUntrusted`, `usingTellorChainlinkFrozen`, and `usingChainlinkTellorUntrusted`. While examining possible transition from the fourth state, we notice the transition logic can be revisited.

To elaborate, we show below the code snippet from the `_fetchPrice()` function. This function is designed to fetch the current price and adjust the current oracle state accordingly. Starting from the fourth state `usingTellorChainlinkFrozen`, the current logic considers the conditions of `!_chainlinkIsFrozen(chainlinkResponse)` (line 269) and `_tellorIsBroken(bandResponse)` (line 284) to still yield `usingTellorChainlinkFrozen` as the next state, which in fact can be better adjusted as `usingChainlinkTellorFrozen`.

```
251         // --- CASE 4: Using Tellor, and Chainlink is frozen ---
252         if (status == Status.usingTellorChainlinkFrozen) {
253             if (_chainlinkIsBroken(chainlinkResponse, prevChainlinkResponse)) {
254                 // If both Oracles are broken, return last good price
```

```
            if (_tellorIsBroken(tellorResponse)) {
                _changeStatus(Status.bothOraclesUntrusted);
                return lastGoodPrice;
            }

            // If Chainlink is broken, remember it and switch to using Tellor
            _changeStatus(Status.usingTellorChainlinkUntrusted);

            if (_tellorIsFrozen(tellorResponse)) {return lastGoodPrice;}

            // If Tellor is working, return Tellor current price
            return _storeTellorPrice(tellorResponse);
        }

        if (_chainlinkIsFrozen(chainlinkResponse)) {
            // if Chainlink is frozen and Tellor is broken, remember Tellor broke, and return last good price
            if (_tellorIsBroken(tellorResponse)) {
                _changeStatus(Status.usingChainlinkTellorUntrusted);
                return lastGoodPrice;
            }

            // If both are frozen, just use lastGoodPrice
            if (_tellorIsFrozen(tellorResponse)) {return lastGoodPrice;}

            // if Chainlink is frozen and Tellor is working, keep using Tellor (no status change)
            return _storeTellorPrice(tellorResponse);
        }

        // if Chainlink is live and Tellor is broken, remember Tellor broke, and return Chainlink price
        if (_tellorIsBroken(tellorResponse)) {
            _changeStatus(Status.usingChainlinkTellorUntrusted);
            return _storeChainlinkPrice(chainlinkResponse);
        }

         // If Chainlink is live and Tellor is frozen, just use last good price (no status change) since we have no basis for comparison
        if (_tellorIsFrozen(tellorResponse)) {return lastGoodPrice;}

        // If Chainlink is live and Tellor is working, compare prices. Switch to Chainlink
        // if prices are within 5%, and return Chainlink price.
        if (_bothOraclesSimilarPrice(chainlinkResponse, tellorResponse)) {
            _changeStatus(Status.chainlinkWorking);
            return _storeChainlinkPrice(chainlinkResponse);
        }

        // Otherwise if Chainlink is live but price not within 5% of Tellor, distrust Chainlink, and return Tellor price
        _changeStatus(Status.usingTellorChainlinkUntrusted);
```

```
            return _storeTellorPrice(tellorResponse);
        }
```

Listing 3.3: `PriceFeed::_fetchPrice()`

Recommendation Apply the proper state-transition logic in `_fetchPrice()` as elaborated earlier.

Status This issue has been confirmed.

3.4 Improved Trove Close Logic in TroveManager

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TroveManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

At the core of `DCNTRL Network` is the `TroveManager` contract which contains the logic to open, adjust and close various troves. Note each trove is in essence an individual collateralized debt position for borrowing users. While reviewing the current trove-closing logic, we notice the current implementation can be improved.

To elaborate, we show below the related `_closeTrove()` routine. The current logic properly releases unused states, including the trove `coll`, `debt`, as well as the associated `rewardSnapshots`. However, it does not release the trove index in the global owners, i.e., `TroveOwners`. The release of `arrayIndex` needs to be performed after the call `_removeTroveOwner()` is completed.

```
    function _closeTrove(address _borrower, Status closedStatus) internal {
        assert(closedStatus != Status.nonExistent && closedStatus != Status.active);

        uint TroveOwnersArrayLength = TroveOwners.length;
        _requireMoreThanOneTroveInSystem(TroveOwnersArrayLength);

        Troves[_borrower].status = closedStatus;
        Troves[_borrower].coll = 0;
        Troves[_borrower].debt = 0;

        rewardSnapshots[_borrower].ETH = 0;
        rewardSnapshots[_borrower].USDEFIDebt = 0;

        _removeTroveOwner(_borrower, TroveOwnersArrayLength);
        sortedTroves.remove(_borrower);
    }
```

Listing 3.4: `TroveManager::_closeTrove()`

Recommendation Release all unused states once a trove is closed. An example revision is shown below:

```
    function _closeTrove(address _borrower, Status closedStatus) internal {
        assert(closedStatus != Status.nonExistent && closedStatus != Status.active);

        uint TroveOwnersArrayLength = TroveOwners.length;
        _requireMoreThanOneTroveInSystem(TroveOwnersArrayLength);

        Troves[_borrower].status = closedStatus;
        Troves[_borrower].coll = 0;
        Troves[_borrower].debt = 0;

        rewardSnapshots[_borrower].ETH = 0;
        rewardSnapshots[_borrower].LUSDDebt = 0;

        _removeTroveOwner(_borrower, TroveOwnersArrayLength);
        sortedTroves.remove(_borrower);
        Troves[_borrower].arrayIndex = 0;
    }
```

Listing 3.5: `TroveManager::_closeTrove()`

Status This issue has been confirmed.

3.5 Improved Validation in USDEFIToken/DCNXToken::permit()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `USDEFIToken`, `DCNXToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

The `DCNTRL Network` protocol has two tokens `USDEFIToken` and `DCNXToken`, each supporting the `EIP2612` functionality. In particular, the `permit()` function is introduced to simplify the token transfer process.

To elaborate, we show below this helper routine from the `USDEFIToken` contract. This routine ensures that the given `owner` is indeed the one who signs the approve request. Note that the internal implementation makes use of the `ecrecover()` precompile for validation. It comes to our attention that the precompile-based validation needs to properly ensure the signer, i.e., `owner`, is not equal to `address(0)`. This issue is also applicable to the `DCNXToken` token contract.

```
    function permit
    (
        address owner,
        address spender,
        uint amount,
        uint deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
        override
    {
        require(deadline >= now, 'USDEFI: expired deadline');
        bytes32 digest = keccak256(abi.encodePacked('\x19\x01',
                                  domainSeparator(), keccak256(abi.encode(
                                  _PERMIT_TYPEHASH, owner, spender, amount,
                                  _nonces[owner]++, deadline))));
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress == owner, 'USDEFI: invalid signature');
        _approve(owner, spender, amount);
    }
```

Listing 3.6: `USDEFIToken::permit()`

Recommendation Strengthen the `permit()` routine to ensure the `owner` is not equal to `address(0)`.

Status This issue has been confirmed.

3.6 Simplified Logic in Unipool::claimReward()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Unipool`
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [3]

Description

In the `Unipool` contract, the `claimReward()` 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 `claimReward()` routine has internally invoked `_updateAccountReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 182).

```
178     function claimReward() public override {
179         require(address(uniToken) != address(0), "Liquidity Pool Token has not been set
            yet");

181         _updatePeriodFinish();
182         _updateAccountReward(msg.sender);

184         uint256 reward = earned(msg.sender);

186         require(reward > 0, "Nothing to claim");

188         rewards[msg.sender] = 0;
189         DCNXToken.transfer(msg.sender, reward);
190         emit RewardPaid(msg.sender, reward);
191     }
```

Listing 3.7: Unipool::claimReward()

```
    function _updateAccountReward(address account) internal {
        _updateReward();

        assert(account != address(0));

        rewards[account] = earned(account);
        userRewardPerTokenPaid[account] = rewardPerTokenStored;
    }
```

Listing 3.8: Unipool::_updateAccountReward()

Having the internal routine `_updateAccountReward()`, there is no need to re-calculate 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 184).

Recommendation Avoid the duplicated calculation of the caller's reward in `claimReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```
    function claimReward() public override {
        require(address(uniToken) != address(0), "Liquidity Pool Token has not been set
            yet");

        _updatePeriodFinish();
        _updateAccountReward(msg.sender);

        uint256 reward = rewards[msg.sender];

        require(reward > 0, "Nothing to claim");

        rewards[msg.sender] = 0;
        lqtyToken.transfer(msg.sender, reward);
        emit RewardPaid(msg.sender, reward);
    }
```

Listing 3.9: Revised Unipool::claimReward()

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the DCNTRL Network protocol, which is a decentralized borrowing protocol that allows to draw low-interest loans against the native asset used as collateral. Loans are paid out in USDEFI (a USD pegged stablecoin) and need to maintain a minimum (configurable) collateral ratio. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

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

[3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

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

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

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