SMART CONTRACT AUDIT REPORT

for

Demeter Protocol

Prepared By: Yiqun Chen

PeckShield

September 12, 2021

Document Properties

Client	Demeter
Title	Smart Contract Audit Report
Target	Demeter
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 12, 2021	Xuxian Jiang	Final Release
1.0-rc1	September 10, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Demeter** design document and related smart contract source code, 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 Demeter

The `Demeter` protocol is designed to enable a complete algorithmic money market protocol on `Heco`. The protocol designs are architected and inspired based on `Compound` and `MakerDAO` and synced into the `Demeter` platform to capitalize the benefits of both systems. `Demeter` enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by staking over-collateralized cryptocurrencies. It also features a synthetic stablecoin (`DUSD`) that is not backed by a basket of fiat currencies but by a basket of cryptocurrencies. `Demeter` utilizes the `Heco` for fast, low-cost transactions while accessing a deep network of wrapped tokens and liquidity.

The basic information of Demeter is as follows:

Table 1.1: Basic Information of Demeter

Item	Description
Issuer	Demeter
Website	https://demeter.vip/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 12, 2021

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

- https://github.com/Demetervip/demeter_contract.git (f4682bc)

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

- https://github.com/Demetervip/demeter_contract.git (1746485)

1.2 About PeckShield

PeckShield Inc. [14] 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.

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) [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. 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 `Demeter` 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	2	
Low	6	
Informational	1	
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 2 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Demeter Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper StakingEIP20::deposit() Logic	Business Logic	Fixed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time And State	Mitigated
PVE-003	Low	Proper totalWithdrawQuantity Accounting in StakingDAO::withdraw()	Business Logic	Fixed
PVE-004	Low	Redundant State/Code Removal	Coding Practice	Fixed
PVE-005	Low	Lack Of Payment Source In executeTransaction()	Coding Practice	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Fixed
PVE-008	Medium	Non ERC20-Compliance Of VToken	Coding Practices	Fixed
PVE-009	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed

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 Proper StakingEIP20::deposit() Logic

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: StakingEIP20
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `Demeter` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating specific staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool. While examining the supported `StakingEIP20` implementation, we notice its deposit logic can be improved.

To elaborate, we show below the `deposit()` function. It stores the user-specific reward accrual height in the `accountHeights` state. However, when the user makes the first deposit globally, this state is not initialized properly. Specifically, the first global deposit initializes `accountHeights[msg.sender] = 1` (line 156), which should not be the case. Instead, the user-specific height should be synchronized with the global, which is currently 0.

```
149     function deposit(uint quantity) external _updateReward(msg.sender) {
150         require(quantity > 0, "quantity less then zero");
151         EIP20Interface(stakingTokenAddr).transferFrom(msg.sender, address(this),
                quantity);
152         accountBalances[msg.sender] = add_(accountBalances[msg.sender], quantity);
153         totalBalance = add_(totalBalance, quantity);
154         totalDepositQuantity = add_(totalDepositQuantity, quantity);
155         if (totalDepositQuantity == quantity && accruedRewardHeight == 0) //
156             accountHeights[msg.sender] = 1;
157         else if (accountBalances[msg.sender] == quantity)
158             accountHeights[msg.sender] = accruedRewardHeight;
159
```

```
        emit Deposit(msg.sender, quantity);
    }
```

Listing 3.1: StakingEIP20::deposit()

Recommendation Revise the above `deposit()` function to properly apply the global `accruedRewardHeight` to the user-specific state.

Status The issue has been fixed by this commit: `70cbb6e`.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [6]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `VBep20Delegate0` as an example, the `_borrowFresh()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 487) start before effecting the update on internal states (lines 490 − 492), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
434     function _borrowFresh(address payable borrower, uint borrowAmount) internal returns
            (uint) {
435         /* Fail if borrow not allowed */
436         uint allowed = ComptrollerInterface(_getComptroller()).borrowAllowed(address(
                this), borrower, borrowAmount);
```

```
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.BORROW_COMPTROLLER_REJECTION, allowed);
        }

        if (!allowBorrow  crossAsset) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.NOT_ALLOW_BORROW);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != block.number) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
        }

        /* Fail gracefully if protocol has insufficient underlying cash */
        if (_getCashPrior() < borrowAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.BORROW_CASH_NOT_AVAILABLE);
        }

        BorrowLocalVars memory vars;

        /*
         * We calculate the new borrower and total borrow balances, failing on overflow:
         *  accountBorrowsNew = accountBorrows + borrowAmount
         *  totalBorrowsNew = totalBorrows + borrowAmount
         */
        (vars.mathErr, vars.accountBorrows) = _borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
```

```
         * We invoke doTransferOut for the borrower and the borrowAmount.
         *  Note: The vToken must handle variations between BEP-20 and BNB underlying.
         *  On success, the vToken borrowAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
         */
        _doTransferOut(borrower, borrowAmount);

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a Borrow event */
        emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);

        /* We call the defense hook */
        ComptrollerInterface(_getComptroller()).borrowVerify(address(this), borrower, borrowAmount);

        return uint(Error.NO_ERROR);
    }
```

Listing 3.2: `VBep20Delegate0::_borrowFresh()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`. The similar issue is also present in other functions, including `_redeemFresh()/_repayBorrowFresh()/_mintFresh()` in other contracts, and the adherence of the `checks-effects-interactions` best practice is strongly recommended. We highlight that the very same issue has been exploited in a recent `Cream` incident [1] and therefore deserves special attention.

From another perspective, the current mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the `Comptroller`-level. In addition, each individual function can be self-strengthened by following the `checks-effects-interactions` principle

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status The issue has been partially fixed by this commit: `70cbb6e`.

3.3 Proper totalWithdrawQuantity Accounting in StakingDAO::withdraw()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `StakingDAO`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

As mentioned in Section 3.1, the `Demeter` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating specific staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool. While examining the supported `StakingDAO` implementation, we notice its withdraw logic can be improved.

To elaborate, we show below the core routine `withdraw()` that actually implements the main logic behind the withdrawal of staked assets. It comes to our attention the current implementation does not properly update the global state `totalWithdrawQuantity` when the staked assets are withdrawn from the first lock pool (lines 255-269). Note the withdraw logic on other lock pools properly keeps track of the global state `totalWithdrawQuantity`.

```
251     function withdraw(uint lockIdx, uint quantity) external _updateReward(msg.sender) {
252         require(accountRewardIndexes[msg.sender] == rewardInfos.length, "need claim");
253         AccountLockInfo[] storage actLockInfos = accountLockInfoLists[msg.sender];
254         require(lockIdx >= 0 && lockIdx < actLockInfos.length, "lock idx overflow");
255         if (lockIdx == 0) {
256             AccountLockInfo storage actLockInfo = actLockInfos[lockIdx];
257             require(quantity > 0 && quantity <= actLockInfo.quantity, "quantity overflow
                    ");
258             uint weightQuantity = actLockInfo.weightQuantity;
259             if (quantity != actLockInfo.quantity)
260                 weightQuantity = div_(mul_(quantity, actLockInfo.weightQuantity),
                        actLockInfo.quantity);
261             totalWeight = sub_(totalWeight, weightQuantity);
262             totalBalance = sub_(totalBalance, quantity);
263             actLockInfo.quantity = sub_(actLockInfo.quantity, quantity);
264             actLockInfo.weightQuantity = sub_(actLockInfo.weightQuantity, weightQuantity
                    );

266             emit Withdraw(msg.sender, lockIdx, quantity, weightQuantity);

268             EIP20Interface(stakingTokenAddr).transfer(msg.sender, quantity);
269         } else {
270             AccountLockInfo storage actLockInfo = actLockInfos[lockIdx];
```

```
            totalBalance = sub_(totalBalance, actLockInfo.quantity);
            totalWeight = sub_(totalWeight, actLockInfo.weightQuantity);

            totalWithdrawQuantity = add_(totalWithdrawQuantity, actLockInfo.
                weightQuantity);

            EIP20Interface(stakingTokenAddr).transfer(msg.sender, actLockInfo.quantity);

            emit Withdraw(msg.sender, lockIdx, quantity, actLockInfo.weightQuantity);

            uint endIdx = actLockInfos.length - 1;
            if (lockIdx != endIdx) {
                actLockInfos[lockIdx] = actLockInfos[endIdx];
                delete actLockInfos[endIdx];
                actLockInfos.length--;
            } else {
                delete actLockInfos[endIdx];
                actLockInfos.length--;
            }
        }

    }
```

Listing 3.3: StakingDAO::withdraw()

Recommendation Revisit the above `withdraw()` logic to keep track of the global state `totalWithdrawQuantity` in all lock pools.

Status The issue has been fixed by this commit: `70cbb6e`.

3.4 Redundant State/Code Removal

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MdexSwapProxy`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [3]

Description

The `Demeter` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeBEP20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `Comptroller` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `VAIController` contract, there are a number of local variables that are defined, but not used. Examples include the `err` field in the defined `MintLocalVars` structures. Note another structure `AccountAmountLocalVars` also contains unused member fields `collateralFactor` and `totalSupplyAmount`.

```
    /*** Main Actions ***/
    struct MintLocalVars {
        Error err;
        MathError mathErr;
        uint mintAmount;
        uint accountMint;
        uint accountMintNew;
        uint originalMintNew;
        uint totalMintNew;
        uint totalMintOriginalPrincipalNew;
    }
```

Listing 3.4: VAIController :: MintLocalVars

```
    struct AccountAmountLocalVars {
        uint totalSupplyAmount;
        uint sumSupply;
        uint sumBorrowPlusEffects;
        uint vTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        uint vaiMinterAmount;
        Exp collateralFactor;
        Exp exchangeRate;
        Exp oraclePrice;
        Exp tokensToDenom;
    }
```

Listing 3.5: VAIController :: AccountAmountLocalVars

```
    modifier onlyMarketVToken() {
        address comptroller = _getComptroller();
        require(comptroller != address(0)  !ComptrollerInterface(comptroller).
            isComptroller(), "invalid comptroller");
        bool isListed = ComptrollerInterface(comptroller).isListedMarket(msg.sender);
        require(isListed, "not listed market");
        _;
    }
```

Listing 3.6: MdexSwapProxy::onlyMarketVToken()

Moreover, the `onlyMarketVToken()` routine from the `MdexSwapProxy` contract can be simplified by removing the requirement on `comptroller != address(0)`, which is already implicitly enforced by the subsequent calls. The requirement on `require(!manager.protocolPaused())` can also be removed in the `VAIController::liquidateVAI()` function.

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by this commit: 70cbb6e.

3.5 Lack Of Payment Source In executeTransaction()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MasterMultiSig
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [5]

Description

The Demeter protocol has a built-in MasterMultiSig contract that contains a rather standard multi-signature implementation. The multi-signature support implements the well-established APIs, including submitTransaction(), confirmTransaction(), revokeConfirmation(), and executeTransaction(). While examining the actual transaction execution, we notice the invoked external_call() allows for the transfer of native tokens, e.g., HT in Heco.

To elaborate, we show below the executeTransaction() routine as well as the helper external_call() routine. It comes to our attention that the executeTransaction() can be invoked when the transaction is being confirmed via confirmTransaction(). However, this confirmTransaction() function does not have the payable modifier, which makes the external_call() with native tokens inconvenient. For gas efficiency by avoiding explicit calls to deposit native tokens, it is suggested to make executeTransaction() callers payable, including confirmTransaction(), and submitTransaction().

```
    /// @dev Allows anyone to execute a confirmed transaction.
    /// @param transactionId Transaction ID.
    function executeTransaction(uint transactionId)
        public
        ownerExists(msg.sender)
        confirmed(transactionId, msg.sender)
        notExecuted(transactionId)
    {
        if (isConfirmed(transactionId)) {
            Transaction storage txn = transactions[transactionId];
            txn.executed = true;
            if (external_call(txn.destination, txn.value, txn.data.length, txn.data))
                emit Execution(transactionId);
            else {
                emit ExecutionFailure(transactionId);
                txn.executed = false;
            }
```

```
        }
    }

    // call has been separated into its own function in order to take advantage
    // of the Solidity's code generator to produce a loop that copies tx.data into memory.
    function external_call(address destination, uint value, uint dataLength, bytes memory data) internal returns (bool) {
        bool result;
        assembly {
            let x := mload(0x40)   // "Allocate" memory for output (0x40 is where "free memory" pointer is stored by convention)
            let d := add(data, 32) // First 32 bytes are the padded length of data, so exclude that
            result := call(
                sub(gas(), 34710),   // 34710 is the value that solidity is currently emitting
                                   // It includes callGas (700) + callVeryLow (3, to pay for SUB) + callValueTransferGas (9000) +
                                   // callNewAccountGas (25000, in case the destination address does not exist and needs creating)
                destination,
                ...
            )
        }
        return result;
    }
```

Listing 3.7: MasterMultiSig::executeTransaction()

Recommendation Add the `payable` modifier to the above functions: `confirmTransaction()` and`submitTransaction()`.

Status This issue has been resolved as the team considers the inclusion of the default fallback function allows to deposit `ether`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

In the Demeter protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setMember(string calldata name, address member) external onlyOwner {
        members[name] = member;
    }

    function setValue(string calldata name, uint value) external onlyOwner {
        values[name] = value;
    }

    function setUserPermit(address user, string calldata permit, bool enable) external
        onlyOwner {
        userPermits[user][permit] = enable;
    }

    function setProtocolPaused(bool state) external onlyOwnerOrGuardian {
        bool oldState = protocolPaused;
        protocolPaused = state;
        emit NewProtocolState(oldState, state);
    }

    function setRedeemPaused(bool state) external onlyOwnerOrGuardian {
        bool oldState = redeemPaused;
        redeemPaused = state;
        emit NewRedeemState(oldState, state);
    }
```

Listing 3.8: Example Setters in the Manager Contract

We should highlight that any account with the DUSDMinter role is authorized to arbitrarily mint the synthetic stablecoin (DUSD). Apparently, if the privileged owner account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

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

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

```
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

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

Listing 3.9: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `claimReward()` routine in the `StakingDAO` contract. If the `USDT` token is supported as `rewardTokenAddr`, the unsafe version of `EIP20Interface(rewardTokenAddr).transfer(msg.sender, quantity)` (line 302) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
294     function claimReward() _updateReward(msg.sender) external {
295         uint quantity = accountUnclaimedReward[msg.sender];
296         if (quantity > 0) {
297             accountUnclaimedReward[msg.sender] = 0;
298             accountClaimedReward[msg.sender] = add_(accountClaimedReward[msg.sender],
                    quantity);

300             totalClaimedRewardQuantity = add_(totalClaimedRewardQuantity, quantity);

302             EIP20Interface(rewardTokenAddr).transfer(msg.sender, quantity);

304             emit ClaimReward(msg.sender, quantity);
305         }
306     }
```

Listing 3.10: StakingDAO::claimReward()

The same issue is also present in other routines, including `deposit()/withdraw()/addReward()/claimReward()` from `StakingDAO`, `swap()/withdraw()` from `MdexSwapProxy`, `drip()/swap()` from `Reservoir`, and `_doApprove()` in `VBep20/VToken`. We highlight that he `approve()`-related idiosyncrasy needs to be addressed by applying `safeApprove()` twice: the first one reduces the allowance to 0 and the second one sets the new intended allowance.

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

Status The issue has been fixed by this commit: `70cbb6e`.

3.8 Non ERC20-Compliance Of VToken

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VToken`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [3]

Description

Each asset supported by the `Demeter` protocol is integrated through a so-called `vToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `vTokens`, users can earn interest through the `vToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `vTokens` as collateral. In the following, we examine the ERC20 compliance of these `vTokens`.

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `vToken` contract. Specifically, the current `transfer()` function simply returns the related error code if the sender does not have sufficient balance to spend. A similar issue is also present in the `transferFrom()` function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `VToken` implementation to ensure its ERC20-compliance.

Status The issue has been fixed by this commit: `70cbb6e`.

3.9 Inconsistency Between Document and Implementation

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [2]

Description

There is a misleading comment embedded among lines of solidity code, which brings unnecessary hurdles to understand and/or maintain the software.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

Specifically, if we examine the `VAIController::repayVAIFresh()` routine, the preceding comment indicates that the intermediate variable `accountTotalInterestRate` is calculated as `accountTotalInterestRate = (vaiBalanceBorrower-originalPrincipal)/originalPrincipal`. However, the implemented logic (line 353) shows it is computed as `accountTotalInterestRate = (vaiBalanceBorrower-originalPrincipal)/ vaiBalanceBorrower`.

```
342        // accountTotalInterestRate = (vaiBalanceBorrower-originalPrincipal)/
               originalPrincipal
343        (vars.mErr, vars.accountTotalInterest) = subUInt(vars.vaiBalanceBorrower, vars.
               originalPrincipal);
344        if (vars.mErr != MathError.NO_ERROR) {
345            return (uint(vars.mErr), 0);
346        }
347
348        (vars.mErr, vars.accountTotalInterestMulti) = mulUInt(vars.accountTotalInterest,
                1e18);
349        if (vars.mErr != MathError.NO_ERROR) {
350            return (uint(vars.mErr), 0);
351        }
352
353        (vars.mErr, vars.accountTotalInterestRate) = divUInt(vars.
               accountTotalInterestMulti, vars.vaiBalanceBorrower);
354        if (vars.mErr != MathError.NO_ERROR) {
355            return (uint(vars.mErr), 0);
356        }
```

Listing 3.11: `VAIController::repayVAIFresh()`

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been fixed by this commit: `70cbb6e`.

4 | Conclusion

In this audit, we have analyzed the `Demeter` design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and synthetic stablecoins. The protocol designs are architected and forked based on `Compound` and `MakerDAO` and synced into the `Demeter` platform to capitalize the benefits of both systems. 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] Aislinn Keely. Cream Finance Exploited in $18.8 million Flash Loan Attack. https://www.theblockcrypto.com/linked/116055/creamfinance-exploited-in-18-8-million-flash-loan-attack.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

[5] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[8] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.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: Concurrency. https://cwe.mitre.org/data/definitions/557.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. PeckShield Inc. https://www.peckshield.com.

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

[16] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.