SMART CONTRACT AUDIT REPORT

for

ERD Protocol

Prepared By: Xiaomi Huang

PeckShield

June 12, 2023

Document Properties

Client	Ethereum Reserve Dollar
Title	Smart Contract Audit Report
Target	ERD Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 12, 2023	Luck Hu	Final Release
1.0-rc	June 2, 2023	Luck Hu	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 ERD 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 ERD Protocol

ERD is a decentralized lending protocol that allows Ether/LSDs(Liquid Staking Derivatives) holders to obtain maximum liquidity against their collateral with paying low interest. After locking up ETH/LSDs as collateral in a smart contract and creating an individual position called a trove, the user can get instant liquidity by minting USDE, an USD-pegged stablecoin. The benefits of ERD include low interest rates, high capital efficiency, direct redemption, and decentralization. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ERD Protocol

Item	Description
Name	Ethereum Reserve Dollar
Website	https://www.erd.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 12, 2023

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

- https://github.com/Ethereum-ERD/dev-upgradeable (37244ee7)

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

- https://github.com/Ethereum-ERD/dev-upgradeable (c46e664f)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 accordingly classified into four categories, 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) [7], 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 comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the ERD protocol implementations. 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	6	■■■■■■
Low	0	
Informational	0	
Total	8	

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 high-severity vulnerabilities and 6 medium-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved WETH Input Amount in _adjustArray()	Business Logic	Fixed
PVE-002	Medium	Revised Collateral Priority Update in setCollateralPriority()	Business Logic	Fixed
PVE-003	High	Revised Transfer Validation in EToken::transferFrom()	Business Logic	Fixed
PVE-004	High	Revised Shares Update in EToken::transfer()	Business Logic	Fixed
PVE-005	Medium	Timely Price Update in validAdjustment()	Coding Practices	Fixed
PVE-006	Medium	Revised EUSD Amount to Mint in _mintToTreasury()	Business Logic	Fixed
PVE-007	Medium	Potential Reentrancy Risk in sendCollateral()	Coding Practices	Fixed
PVE-008	Medium	Trust Issue on Admin Keys	Security Features	Mitigated

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

3 | Detailed Results

3.1 Improved WETH Input Amount in _adjustArray()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: BorrowerOperations
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

ERD allows a user to open a Trove to borrow USDE by supplying required collateral. The first token to be supported as collateral is WETH. To facilitate users supply WETH, the protocol allows users to take ETH (msg.value) within the transactions to open troves. The ETH will be deposited to WETH by ERD on behalf of the borrower. While examining the calculation of the WETH amount to be supplied, we notice the WETH amount may not be correctly counted.

In the following, we show the code snippet of the _adjustArray() routine which is used to count the input WETH amount in all the input collaterals. By design, if msg.value > 0, it shall check if WETH exists in the input collateral list or not. If WETH doesn't exist, it adds WETH into the collateral list with msg.value as its amount. Otherwise, it adds msg.value to the amount of WETH.

However, it comes to our attention that, when WETH exists in the input collateral list, the current implementation tries to update the corresponding collateral address from WETH to a wrong one (line 323). As a result, the input collaterals are messed up. Based on this, we suggest to update the input amount of WETH only when WETH exists in the input collateral list.

```
296     function _adjustArray(
297       address[] memory _collaterals,
298       uint256[] memory _amounts,
299       uint256 _amount
300   ) public view returns (address[] memory, uint256[] memory) {
301       uint256 collLen = _collaterals.length;
302       if (collLen == 0 && _amount > 0) {...}
303       if (_amount > 0) {
```

```
            address[] memory collaterals = new address[](collLen + 1);
            uint256[] memory amounts = new uint256[](collLen + 1);
            collaterals[0] = address(WETH);
            amounts[0] = _amount;
            address collateral;
            bool hasWETH;
            uint256 index;
            for (uint256 i = 0; i < collLen; i++) {
                collateral = _collaterals[i];
                if (collateral != address(WETH)) {
                    collaterals[i + 1] = collateral;
                    amounts[i + 1] = _amounts[i];
                } else {
                    hasWETH = true;
                    index = i;
                    break;
                }
            }
            if (hasWETH) {
                _collaterals[index] = _collaterals[index.add(_amount)];
                return (_collaterals, _amounts);
            } else {
                return (collaterals, amounts);
            }
        } else {
            return (_collaterals, _amounts);
        }
    }
```

Listing 3.1: BorrowerOperations::_adjustArray()

Recommendation Revisit the above `_adjustArray()` routine and correctly update the input amount of `WETH`.

Status The issue has been fixed by these commits: `0725084e` and `ae03f1c7`.

3.2 Revised Collateral Priority Update in setCollateralPriority()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CollateralManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

In the `ERD` protocol, the `CollateralManager` contract maintains a list of the supported collaterals and their parameters. The collaterals are sorted in descending order per their priorities. The protocol owner has the right to add/remove collaterals and update their parameters. While examining the collateral priority update functionality, we notice a logical issue that may mess up the collaterals priority order and their parameters.

To elaborate, we show below the code snippet of the `setCollateralPriority()` routine. As the name indicates, it is used to update the priority of the input collateral. Specifically, if the new priority gets higher, the collateral is moved up from current index in the sorted collateral list. Similarly, when the new priority is lowered, the collateral is moved down from current index.

```
  function setCollateralPriority(
    address _collateral,
    uint256 _newIndex
) external override onlyOwner {
    _requireCollIsActive(_collateral);
    uint256 oldIndex = getIndex(_collateral);
    uint256 newIndex = _newIndex;
    assert(newIndex != oldIndex && newIndex < collateralsCount);
    if (newIndex < oldIndex) {
        uint256 tmpIndex = oldIndex;
        uint256 gap = oldIndex - newIndex;
        for (uint256 i = 0; i < gap; ) {
            tmpIndex = _up(tmpIndex);
            unchecked {
                i++;
            }
        }
    } else {
        uint256 tmpIndex = newIndex;
        uint256 gap = oldIndex - newIndex;
        for (uint256 i = 0; i < gap; ) {
            tmpIndex = _down(tmpIndex);
            unchecked {
                i++;
            }
        }
```

```
185         }
186         collateralParams[_collateral].index = _newIndex;
187     }
```

Listing 3.2: setCollateralPriority ()

However, it comes to our attention that, when the priority gets lower, i.e., new index is larger than the old index, it tries to move the collateral at the new index while not the given collateral at the old index (line 177). Our analysis shows that the `tmpIndex` shall be set to `oldIndex`, i.e., `uint256 tmpIndex = newIndex`. What's more, it may revert the function because of arithmetic underflow when calculating the gap between the old index and the new index (line 178), i.e., `uint256 gap = oldIndex - newIndex`, because `oldIndex <= newIndex` in this case.

In addition, during the process of moving the target collateral to the new index, the indexes of the collaterals on the moving path are also updated. However, it only links the `collateralParams[_collateral]` to the new index for the target collateral. As a result, the links in the `collateralParams[_collateral]` may be out-of-date for all the other affected collaterals. Based on this, we suggest to update the `collateralParams[_collateral].index` for all the collaterals whose indexes are updated.

Recommendation Revisit the collateral priority update in the `setCollateralPriority()` routine and properly update the positions of all the affected collaterals and update their `collateralParams[_collateral].index` accordingly.

Status The issue has been fixed by this commit: `0fa375e7`.

3.3 Revised Transfer Validation in EToken::transferFrom()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `EToken`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

In the `ERD` protocol, `EToken` is a wrapped token contract that certificates users deposit of collaterals. The `EToken` balance of a user is the collateral amount of the user in the protocol. A user can transfer its `EToken` in normal mode, but must ensure that after the transfer the protocol is still in normal mode and its new `ICR` is greater than the `CCR`. While reviewing the validation of the transfer, we notice the logic issue that may permit a transfer that should be forbidden or block a transfer that should be permitted.

To elaborate, we show below the related code snippet of the `EToken::transferFrom()` routine. At the beginning of the routine, it calls the `_requireValidAdjustment()` routine (line 105) to check whether the transfer is valid or not. However, in the `_requireValidAdjustment()` routine, we notice that it validates the trove for the `msg.sender` (line 115) while not the transfer `_sender` (line 100). As a result, a user can approve an operator to transfer all its `EToken` as long as the operator is valid for transfer. Based on this, we suggest to properly validate the trove of the transfer sender.

```
99    function transferFrom(
100     address _sender,
101     address _recipient,
102     uint256 _amount
103 ) public virtual override(IERC20Upgradeable, ERC20Upgradeable) returns (bool) {
104     uint256 share = getShare(_amount);
105     _requireValidAdjustment(_amount);
106     shares[_sender] = shares[_sender].sub(share);
107     _totalShares = _totalShares.sub(share);
108     super.transferFrom(_sender, _recipient, _amount);
109     return true;
110 }

112 function _requireValidAdjustment(uint256 _amount) internal view {
113   require(
114       collateralManager.validAdjustment(
115           msg.sender,
116           tokenAddress,
117           _amount
118       ),
119       "EToken: Invalid adjustment"
120   );
121 }
```

Listing 3.3: EToken::transferFrom()

What's more, in the `_requireValidAdjustment()` routine, it calls the `collateralManager.validAdjustment()` routine to validate for the transfer. While examining below the code of the `CollateralManager::validAdjustment()` routine, we notice it directly return `false` when the protocol is currently not in recovery mode (line 604). As a result, `EToken` transfer is forbidden in normal mode, though by design the transfer is permitted only in normal mode.

```
590 function validAdjustment(
591   address _account,
592   address _collateral,
593   uint256 _amount
594 ) external view override returns (bool) {
595   bool active = troveManager.getTroveStatus(_account) == 1;
596   if (!active) {
597       return true;
598   }
599   uint256 price = priceFeed.fetchPrice_view();
600   uint256 totalDebt = getEntireSystemDebt();
```

```
    (, , uint256 totalValue) = getEntireSystemColl(price);
    bool isRecoveryMode = _checkRecoveryMode(totalValue, totalDebt, CCR);
    if (!isRecoveryMode) {
        return false;
    }
    ...
    return newTCR >= CCR;
}
```

Listing 3.4: CollateralManager::validAdjustment()

Recommendation Properly validate the trove of the transfer sender in the `EToken::transferFrom()` routine and allow the transfer in normal mode only.

Status The issue has been fixed by this commit: `9a9be04a`.

3.4 Revised Shares Update in EToken::transfer()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `EToken`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `ERD` protocol targets to support `stETH` as collateral, which is a rebasing token. In order to refund the income generated by `stETH` to the user who deposits `stETH` as collateral, it records a share balance for the user in `EToken` contract. So the token balance of the `stETH` depositor can change constantly with the rebasing of `stETH`. In particular, if the collateral is not rebasing token, the share balance of a user is equal to its token balance. While reviewing the share balance update in `EToken` transfer, we notice there is a lack of update for the share balance of the recipient.

In the following, we show the related code snippet of the `EToken::transfer()` routine, which is used to transfer `EToken`. It gets the equivalent share amount for the input token amount to transfer (line 91), reduces the share amount from the sender's share balance (line 93) and the total shares (line 94), and transfer the tokens to the recipient (line 95) at last. However, we notice the routine does not add the share amount to the share balance of the recipient. As a result, it reduces the share amount from the sender's share balance but the recipient doesn't receive any share. Our analysis shows that we need to add the corresponding share amount to the recipient's share balance.

```
87  function transfer(
88    address _recipient,
89    uint256 _amount
```

```
) public virtual override(IERC20Upgradeable, ERC20Upgradeable) returns (bool) {
    uint256 share = getShare(_amount);
    _requireValidAdjustment(_amount);
    shares[msg.sender] = shares[msg.sender].sub(share);
    _totalShares = _totalShares.sub(share);
    _transfer(msg.sender, _recipient, _amount);
    return true;
}
```

Listing 3.5: `EToken::transfer()`

Note this issue is also applicable to the `EToken::transferFrom()` routine.

Recommendation Revisit the `EToken::transfer()/transferFrom()` routines and properly add the share amount to the recipient's share balance.

Status The issue has been fixed by this commit: `9a9be04a`.

3.5 Timely Price Update in validAdjustment()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.3, the `collateralManager.validAdjustment()` routine is used to validate if a `EToken` transfer from the sender is permitted or not. The routine basically checks three conditions: 1) if the protocol is currently in normal mode (`TCR >= CCR`); 2) if the new `ICR` of the sender is larger than the `MCR`; 3) if the protocol can be in normal mode after the transfer.

In the following, we show the code snippet of the `CollateralManager::validAdjustment()` routine. In order to calculate the `TCR/ICR`, there is a need to fetch the latest collaterals prices from the price oracle. However, it comes to our attention that it simply fetch the last good prices from the `priceFeed` (line 599). The last good price is stored in the `priceFeed` for each collateral when the price is successfully obtained from the oracle. As a result, the last good price may be out of date without an initiative price update request to the oracle. Based on this, we suggest to update the prices via the `priceFeed` for all collaterals before using the last good prices.

```
590 function validAdjustment(
591   address _account,
592   address _collateral,
593   uint256 _amount
```

```
) external view override returns (bool) {
    bool active = troveManager.getTroveStatus(_account) == 1;
    if (!active) {
        return true;
    }
    uint256 price = priceFeed.fetchPrice_view();
    uint256 totalDebt = getEntireSystemDebt();
    (, , uint256 totalValue) = getEntireSystemColl(price);
    bool isRecoveryMode = _checkRecoveryMode(totalValue, totalDebt, CCR);
    if (!isRecoveryMode) {
        return false;
    }
    (uint256[] memory colls, , ) = getTroveColls(_account);
    uint256 debt = troveManager.getTroveDebt(_account);
    (uint256 currValue, ) = getValue(collateralSupport, colls, price);
    uint256 value = _calcValue(_collateral, _amount, price);
    uint256 newICR = ERDMath._computeCR(currValue.sub(value), debt);
    if (newICR < MCR) {
        return false;
    }
    uint256 newTCR = ERDMath._computeCR(totalValue.sub(value), totalDebt);
    return newTCR >= CCR;
}
```

Listing 3.6: `CollateralManager::validAdjustment()`

Note this issue is also applicable to the `CollateralManager::getTotalValue()/TroveManagerRedemptions.updateTroves()/BorrowerWrappersScript::_getNetEUSDAmount()`, etc.

Recommendation Revisit all the routines that fetch the last good prices add ensure the prices are updated to date.

Status The issue has been fixed by these commits: `8c953e45` and `832b74e8`.

3.6 Revised EUSD Amount to Mint in _mintToTreasury()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TroveLogic`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `ERD` protocol, the trove owner has to pay a low interest for the trove debt. The interest that is accumulated by a trove from the last operation to the present will be minted to the treasury. The interest is charged based on the borrowing rate. While reviewing the calculation of the repaid interest that will be minted to the treasury, we notice it is wrongly divided by the `newLiquidityIndex`.

In the following, we show the related code snippet of the `TroveLogic::_mintToTreasury()` routine, which is used to calculate the accumulated interest amount for the `scaledDebt` when the borrow index changes from `previousBorrowIndex` to `newBorrowIndex`. Then the accumulated interest amount is scaled by dividing the `newLiquidityIndex` (line 185), and the result is used as the `USDE` amount to be minted to the treasury. However, we notice the `EUSDToken` contract implements a standard `ERC20` which has no special processing for the `liquidityIndex`. As a result, the treasury actually receives a scaled token balance which may be less than the minted amount.

Our analysis shows that we can directly use the accumulated interest amount as the `USDE` amount to be minted to the treasury.

```
167   function _mintToTreasury(
168     DataTypes.TroveData storage trove,
169     uint256 scaledDebt,
170     uint256 previousBorrowIndex,
171     uint256 newLiquidityIndex,
172     uint256 newBorrowIndex
173 ) internal {
174     MintToTreasuryLocalVars memory vars;
175
176     //calculate the last principal variable debt
177     vars.previousDebt = scaledDebt.rayMul(previousBorrowIndex);
178
179     //calculate the new total supply after accumulation of the index
180     vars.currentDebt = scaledDebt.rayMul(newBorrowIndex);
181
182     //debt accrued is the sum of the current debt minus the sum of the debt at the last
          update
183     vars.totalDebtAccrued = vars.currentDebt.sub(vars.previousDebt);
184
185     vars.amountToMint = vars.totalDebtAccrued.rayDiv(newLiquidityIndex);
186
187     if (vars.amountToMint != 0) {
188         IEUSDToken(trove.eusdTokenAddress).mintToTreasury(
189             vars.amountToMint,
190             trove.factor
191         );
192     }
193 }
```

Listing 3.7: `TroveLogic::_mintToTreasury()`

Recommendation Remove the division by the `newLiquidityIndex` and use the accumulated interest amount as the `USDE` amount to be minted to the treasury.

Status The issue has been fixed by this commit: `587e3a7a`.

3.7 Potential Reentrancy Risk in sendCollateral()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. In the `ActivePool` contract, the `sendCollateral()` function (see the code snippet below) is provided to transfer the given amounts of collaterals to the given account by externally calling the collaterals contracts. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. Apparently, the interaction with the external contract (line 163) may start before the transferring of other collaterals that will update the token balances of the contract, hence violating the principle.

```
143     function sendCollateral(
144         address _account,
145         address[] memory _collaterals,
146         uint256[] memory _amounts
147     ) external override {
148         _requireCallerIsBOorTroveMorSP();
149         uint256 collLen = _collaterals.length;
150         address collateral;
151         uint256 amount;
152         bool flag = _notNeedsToSwitchWETH(_account);
153         for (uint256 i = 0; i < collLen; ) {
154             collateral = _collaterals[i];
155             amount = _amounts[i];
156             if (amount != 0) {
157                 if (collateral != address(WETH)) {
158                     _sendCollateral(_account, collateral, amount);
159                 } else {
160                     if (flag) {
161                         _sendCollateral(_account, collateral, amount);
```

```
                } else {
                    _sendETH(_account, amount);
                }
            }
        }
        unchecked {
            i++;
        }
    }
}
```

Listing 3.8: ActivePool :: sendCollateral ()

Specifically, in the case when the recipient of `_sendETH()` is a contract, it could hijack a call to the protocol before the transferring of other collaterals. Within the `receive()/fallback()` functions of the recipient contract, it could call the functions in the protocol that will fetch the collaterals balances of the `ActivePool` contract, e.g, the `ActivePool::getTotalCollateral()` routine as the code shown below. Since the other collaterals except for `ETH` are not transferred out yet, the `IERC20Upgradeable(collaterals[i]).balanceOf(address(this))` (line 114) will return a larger value. Generally the collaterals balances are used to calculate the total collateral ratio `TCR`, which is further used to check if the protocol is in recovery mode or normal mode. Similarly, if some collateral has a callback function, e.g., `ERC777`, it can also hijack a call to the protocol from the callback function.

Based on this, we suggest to properly protect the key functions with the `nonReentrant` modifier. Another option is to always transfer `ETH` at last if `ETH` is the only collateral that has callback function.

```
100 function getTotalCollateral()
101     public
102     view
103     override
104     returns (
105         uint256 total,
106         address[] memory collaterals,
107         uint256[] memory amounts
108     )
109 {
110     collaterals = ITroveManager(troveManagerAddress).getCollateralSupport();
111     uint256 collLen = collaterals.length;
112     amounts = new uint256[](collLen);
113     for (uint256 i = 0; i < collLen; ) {
114         amounts[i] = IERC20Upgradeable(collaterals[i]).balanceOf(
115             address(this)
116         );
117         total = total.add(amounts[i]);
118         unchecked {
119             i++;
120         }
121     }
```

```
    }
```

Listing 3.9: ActivePool :: getTotalCollateral ()

Note the same issue is also applicable to the `StabilityPool::_sendCollateralGainToDepositor()/CollSurplusPool::claimColl()` routines, etc.

Recommendation Apply the `checks-effects-interactions` design pattern or add the `nonReentrant` guard modifier.

Status The issue has been fixed by this commit: `1e314739`.

3.8 Trust Issue on Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `ERD` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in regulating the protocol-wide operations (e.g., add/remove collaterals, update collateral priority). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `CollateralManager` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `CollateralManager` allow for the `owner` to set protocol contracts addresses, e.g., the `activePool/defaultPool` addresses, add/remove collateral, set collateral priority, pause/unpause a collateral, set the `eToken` address for a collateral, set the value ratio for a collateral, etc.

```
    function setAddresses(
        address _activePoolAddress,
        address _borrowerOperationsAddress,
        address _defaultPoolAddress,
        address _priceFeedAddress,
        address _troveManagerAddress,
        address _troveManagerRedemptionsAddress,
        address _wethAddress
    ) external override onlyOwner {
        _requireIsContract(_activePoolAddress);
        _requireIsContract(_borrowerOperationsAddress);
        _requireIsContract(_defaultPoolAddress);
        _requireIsContract(_priceFeedAddress);
```

```
        _requireIsContract(_wethAddress);
        _requireIsContract(_troveManagerAddress);

        borrowerOperationsAddress = _borrowerOperationsAddress;
        activePool = IActivePool(_activePoolAddress);
        defaultPool = IDefaultPool(_defaultPoolAddress);
        priceFeed = IPriceFeed(_priceFeedAddress);
        wethAddress = _wethAddress;

        troveManager = ITroveManager(_troveManagerAddress);
        troveManagerRedemptionsAddress = _troveManagerRedemptionsAddress;
        ...
    }

    function addCollateral(
        address _collateral,
        address _oracle,
        address _eTokenAddress,
        uint256 _ratio
    ) external override onlyOwner {
        require(
            !getIsSupport(_collateral),
            "CollateralManager: Collateral already exists"
        );
        _requireRatioLegal(_ratio);

        collateralParams[_collateral] = DataTypes.CollateralParams(...);
        collateralSupport.push(_collateral);
        collateralsCount = collateralsCount.add(1);
    }

    function removeCollateral(address _collateral) external override onlyOwner {
        address collAddress = _collateral;
        require(
            getIsSupport(collAddress) && !getIsActive(collAddress),
            "CollateralManager: Collateral not pause"
        );
        require(
            collateralsCount > 1,
            "CollateralManager: Need at least one collateral support"
        );
        uint256 index = getIndex(collAddress);
        address collateral;
        for (uint256 i = index; i < collateralsCount - 1; ) {...}
        collateralSupport.pop();
        collateralsCount = collateralsCount.sub(1);
        delete collateralParams[collAddress];
    }

    function setCollateralPriority(
        address _collateral,
        uint256 _newIndex
```

```
    ) external override onlyOwner {
        _requireCollIsActive(_collateral);
        uint256 oldIndex = getIndex(_collateral);
        uint256 newIndex = _newIndex;
        assert(newIndex != oldIndex && newIndex < collateralsCount);
        if (newIndex < oldIndex) {
            uint256 tmpIndex = oldIndex;
            uint256 gap = oldIndex - newIndex;
            for (uint256 i = 0; i < gap; ) {
                tmpIndex = _up(tmpIndex);
                unchecked {
                    i++;
                }
            }
        } else {...}
        collateralParams[_collateral].index = _newIndex;
    }

    function pauseCollateral(address _collateral) external override onlyOwner {
        _setStatus(_collateral, 2);
    }

    function activeCollateral(address _collateral) external override onlyOwner {
        _setStatus(_collateral, 1);
    }

    function setOracle(
        address _collateral,
        address _oracle
    ) public override onlyOwner {
        collateralParams[_collateral].oracle = _oracle;
    }

    function setEToken(
        address _collateral,
        address _eTokenAddress
    ) public override onlyOwner {
        collateralParams[_collateral].eToken = _eTokenAddress;
    }

    function setRatio(
        address _collateral,
        uint256 _ratio
    ) public override onlyOwner {
        _requireCollIsActive(_collateral);
        _requireRatioLegal(_ratio);
        collateralParams[_collateral].ratio = _ratio;
    }
```

Listing 3.10: Example Privileged Operations in `CollateralManager`

We understand the need of the privileged functions for proper operations, but at the same time the extra power to the privileged account may also be a counter-party risk to the ERD users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

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

Status The issue has been mitigated as the team confirm they will transfer the ownership to a multi-sig account after deployment.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `ERD` protocol. `ERD` is a decentralized lending protocol that allows `Ether/LSDs(Liquid Staking Derivatives)` holders to obtain maximum liquidity against their collateral with paying low interest. The benefits of `ERD` include low interest rates, high capital efficiency, direct redemption, and decentralization. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

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

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

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

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

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

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

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

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

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

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