SMART CONTRACT AUDIT REPORT

for

HANDLE.FI

Prepared By: Yiqun Chen

PeckShield

October 11, 2021

Document Properties

Client	handle.fi
Title	Smart Contract Audit Report
Target	handle.fi
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	October 11, 2021	Xuxian Jiang	Final Release
1.0-rc2	October 2, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	August 22, 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 **handle.fi** 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 handle.fi

The `handle.fi` protocol is a decentralized multi-currency stablecoin creation and exchange protocol. The protocol enables users to create (borrow) and convert between multi-currency stablecoins, i.e., `fxTokens`. The multi-currency stablecoin exchange provided by `handle.fi` can settle transactions in local currency, thus reducing conversion fees and foreign currency risks and facilitating efficient conversion between multi-currency stablecoins. Initial rollout is targeted for `fxTokens` representing `Australian Dollar` (`fxAUD`), `Japanese Yen` (`fxJPY`), and others. Overall, it enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies.

The basic information of handle.fi is as follows:

Table 1.1: Basic Information of handle.fi

Item	Description
Name	handle.fi
Website	https://handle.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 11, 2021

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

- https://github.com/handle-fi/handle-vue.git (d95b64f)

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

- https://github.com/handle-fi/handle-vue.git (56a76ae)

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.

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 1.3: 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

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 design and implementation of the `handle.fi` 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	2	■■
Medium	3	■■■
Low	7	■■■■■■■
Informational	0	
Undetermined	1	■
Total	13	

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, 3 medium-severity vulnerabilities, 7 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key handle.fi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-002	Medium	Improved Logic in Handle::removeFxToken()	Business Logic	Fixed
PVE-003	Low	Improved Sanity Checks Of System/-Function Parameters	Coding Practices	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Lack Of Duplicate Checks To Add The Same FxToken	Coding Practice	Fixed
PVE-006	Low	Timely charge() Right Before Interest Rate Changes	Business Logic	Resolved
PVE-007	High	Incorrect Calculation of getNewMinimumRatio()	Business Logic	Fixed
PVE-008	Low	Improved Arithmetic Calculation	Numeric Errors	Fixed
PVE-009	Undetermined	Strengthened Reentrancy Prevention in Comptroller	Time And State	Fixed
PVE-010	Medium	Proper Debt Absorb in absorbDebt()	Business Logic	Fixed
PVE-011	Low	Timely Interest Rate Update In withdrawCollateral()	Business Logic	Fixed
PVE-012	High	Improved Logic in PCT::ensureUpperBoundLimit()	Business Logic	Fixed
PVE-013	Low	Inconsistent Deposit Fee Calculation And Collection	Business Logic	Fixed

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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

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 this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.1: USDT Token Contract

Because of that, a normal call to approve() with a currently non-zero allowance may fail. Also, the IERC20 interface has defined the approve() interface with a bool return value, but the above implementation does not have the return value. As a result, a regular IERC20-based approve() with a non-compliant token may unfortunately revert the transaction. In the following, we use the Comptroller::_mintAndDeposit() routine as an example. This routine is designed to initialize various states. To accommodate the specific idiosyncrasy, there is a need to use safeApprove(), instead of current approve() (line 170). Moreover, to accommodate possible non-zero allowance, it is suggested to apply the safeApprove() twice: The first one resets the allowance to zero, while the second time sets the intended allowance amount.

```
    function _mintAndDeposit(
        uint256 tokenAmount,
        address token,
        address collateralToken,
        uint256 collateralAmount,
        address referral
    ) private {
        assert(
            IERC20(collateralToken).approve(address(treasury), collateralAmount)
        );

        // Calculate fee with current amount and increase token amount to include fee.
        uint256 feeTokens = tokenAmount.mul(handle.mintFeePerMille()).div(1000);
        ...
  }
```

Listing 3.2: Comptroller::_mintAndDeposit()

Similarly, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the transfer() function does not have a return value. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted.

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 approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom().

Note that a number of functions share the same issue, including _forceWithdrawCollateral(), _depositCollateral(), and requestFundsPCT() in Treasury as well as mint() and _mintAndDeposit() in

Comptroller.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom().

Status The issue has been fixed by the following commits: 40f8048 and da6c546.

3.2 Improved Logic in Handle::removeFxToken()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Handle
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

The handle.fi protocol has an essential contract Handle that stores the main protocol data and configurations. This contract provides public functions that allow authorized admin to dynamically adjust the list of fxTokens supported in the protocol. Accordingly, the setFxToken() and removeFxToken() are these two functions to list and delist a fxToken from the protocol.

To elaborate, we show below the removeFxToken() function. This function has a rather straightforward logic in firstly locating the current index of the to-be-removed fxToken, then removing the index from the array by swapping with the last item in the validFxTokens array, and finally emit a related event for off-chain reporting and monitoring.

```
    /** @dev Invalidate an existing fxToken and remove it from the protocol */
    function removeFxToken(address token) external override onlyAdmin {
        uint256 tokenIndex = validFxTokens.length;
        for (uint256 i = 0; i < tokenIndex; i++) {
            if (validFxTokens[i] == token) {
                tokenIndex = i;
                break;
            }
        }
        // Assert that token was found.
        assert(tokenIndex < validFxTokens.length);
        delete isFxTokenValid[token];
        if (tokenIndex < validFxTokens.length - 1) {
            delete validFxTokens[tokenIndex];
            // Replace to-be-deleted item with last element and then pop array.
            validFxTokens[tokenIndex] = validFxTokens[validFxTokens.length - 1];
            validFxTokens.pop();
        } else {
            // Token index is last element, so no need to pop array.
```

```
138            delete validFxTokens[tokenIndex];
139        }
140        emit ConfigureFxToken(token);
141    }
```

Listing 3.3: `Handle::removeFxToken()`

Our analysis shows that the above routine can be improved in three aspects. Firstly, the operation on `delete validFxTokens[tokenIndex]` (line 132) is not necessary as it is immediately overwritten by the following statement (line 134). Secondly, when the to-be-removed `fxToken` is the last element, the operation on `delete validFxTokens[tokenIndex]` (line 138) needs to be replaced as `validFxTokens.pop()`. Thirdly, the final event `emit ConfigureFxToken(token)` (line 140) is the same as the one emitted when a `fxToken` is added via `setFxToken()`. It is helpful to emit different events to differentiate these two actions.

Recommendation Improve the above `removeFxToken()` to properly maintain the list of supported `fxToken`s in `validFxTokens`.

Status The issue has been fixed by this commit: `5ccfc63`.

3.3 Improved Sanity Checks For System/Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Handle`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `handle.fi` protocol is no exception. Specifically, if we examine the `Handle` contract, it has defined a number of protocol-wide risk parameters, such as `withdrawFeePerMille` and `depositFeePerMille`. In the following, we show the corresponding routines that allow for their changes.

```
    /** @dev Setter for all protocol transaction fees */
    function setFees(
        uint256 _withdrawFeePerMille,
        uint256 _depositFeePerMille,
        uint256 _mintFeePerMille,
        uint256 _burnFeePerMille
    ) external override onlyAdmin {
        withdrawFeePerMille = _withdrawFeePerMille;
        depositFeePerMille = _depositFeePerMille;
        burnFeePerMille = _burnFeePerMille;
```

```
        mintFeePerMille = _mintFeePerMille;
    }
```

Listing 3.4: `Handle::setFees()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `depositFeePerMille` may charge unreasonably high fee in the `depositCollateral()` operation, hence incurring cost to users or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been confirmed. The team clarifies the sanity checks on admin functions remain unchanged in favor of smaller contract sizes. Additional checks and validations on any parameter changes and ranges will be implemented via `handleDAO` operation.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Handle`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [4]

Description

In the `handle.fi` protocol, the `admin` account plays a critical role in governing and regulating the system-wide operations (e.g., `oracle` addition, fee adjustment, and parameter setting). It also has the privilege to regulate or govern the flow of assets for borrowing and lending among the involved components, i.e., `Comptroller`, `fxKeeperPool`, and `Treasury`.

With great privilege comes great responsibility. Our analysis shows that the governance account is indeed privileged. In the following, we show representative privileged operations in the `handle.fi` protocol.

```
    function setComponents(address[] memory components)
        external
        override
        onlyAdmin
    {
        uint256 l = components.length;
```

```
        for (uint256 i = 0; i < 1; i++) {
            require(components[i] != address(0), "Invalid address");
        }
        treasury = payable(components[0]);
        comptroller = components[1];
        vaultLibrary = components[2];
        fxKeeperPool = components[3];
        pct = components[4];
        liquidator = components[5];
        interest = components[6];
        referral = components[7];
        for (uint256 i = 0; i < 7; i++) {
            // Skip VaultLibrary, fxKeeperPool and referral (i != 7).
            if (i == 2  i == 3) continue;
            // Grant operator roles if needed.
            if (!hasRole(OPERATOR_ROLE, components[i]))
                grantRole(OPERATOR_ROLE, components[i]);
        }
    }
```

Listing 3.5: Various `Setters` in `Handle`

We emphasize that the privilege assignment with various contracts is necessary and required for proper protocol operations. However, it is worrisome if the `admin` is not governed by a `DAO`-like structure. Meanwhile, we point out that a compromised `admin` account would allow the attacker to add a malicious contract or change other settings to steal funds in current protocol, which directly undermines the assumption of the `handle.fi` protocol.

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 The issue has been confirmed. The team clarifies that the `admin` is governed by the `handleDAO` (that is not part of current audit). Current admin privileges are for initial deployment and guarded launch phase. Subsequent to full protocol launch the normal on-chain community-based governance life-cycle will be activated.

3.5 Lack Of Duplicate Checks To Add The Same FxToken

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Handle`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.2, the `handle.fi` protocol has an essential contract `Handle` that stores the main protocol data and configurations. Specifically, this contract provides two public functions `setFxToken()` and `removeFxToken()` to list and delist a `fxToken` from the protocol.

To elaborate, we show below the `setFxToken()` routine. This routine is designed to add a new `fxToken` into the protocol. It comes to our attention this routine does not validate whether the given `fxToken` may exist in the current list of `validFxTokens`. As a result, a duplicate `fxToken` may be accidentally added into the list.

```
    /** @dev Configure an ERC20 as a valid fxToken */
    function setFxToken(address token) public override onlyAdmin {
        validFxTokens.push(token);
        isFxTokenValid[token] = true;
        emit ConfigureFxToken(token);
    }
```

Listing 3.6: `Handle::setFxToken()`

Recommendation Ensure the given `fxToken` for addition does not exist in current `validFxTokens`.

Status The issue has been fixed by this commit: `6a24a39`.

3.6 Timely charge() Right Before Interest Rate Changes

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Interest`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

As a decentralized multi-currency stablecoin creation and exchange protocol, the `handle.fi` protocol is designed to collect interests from minted stablecoins. With that, there is a need to compute the current interest rate, which is implemented in the `updateRates()` function.

To elaborate, we show below the full implementation of the `updateRates()` function. It has a rather straightforward logic in iterating each supported collateral asset, fetching the current interest rate, and then updating the new interest rate in the accounting contract `Handle`. It comes to our attention that when a new interest rate becomes effective, it is suggested to timely accumulate the due interest with the previous interest rate. Otherwise, when the interest is collected after the new interest rate becomes effective, the calculated interest may not be accurate. For proper interest accounting, there is a need to timely collect interest before applying the new interest rate.

```
    /**
     * @dev Updates the interest rate via the data source
     */
    function updateRates() public override notPaused {
        address[] memory collateralTokens = handle.getAllCollateralTypes();
        uint256 j = collateralTokens.length;
        IHandle.CollateralData memory data;
        uint256 interestRate;
        for (uint256 i = 0; i < j; i++) {
            data = handle.getCollateralDetails(collateralTokens[i]);
            interestRate = fetchRate(collateralTokens[i]);
            // Update collateral with fetched interest.
            handle.setCollateralToken(
                collateralTokens[i],
                data.mintCR,
                data.liquidationFee,
                // New interest rate as a per mille ratio (1/1000th, 1 decimal).
                interestRate
            );
        }
        // Update the fetch time for caching purposes.
        interestRateLastUpdated = block.timestamp;
    }
```

Listing 3.7: `Interest::updateRates()`

Recommendation Revise the `updateRates()` implementation to ensure proper interest computation and collection.

Status This issue has been resolved as the function `Handle::setCollateralToken()`, called by `Interest::updateRates()`, writes the current cumulative `R` value to the `Interest` contract by calling `Interest::charge()` before updating the interest rates.

3.7 Incorrect Calculation of getNewMinimumRatio()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `VaultLibrary`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

To facilitate the code organization and contain contract code size for deployment, the `handle.fi` protocol has a library contract named `VaultLibrary`. This library contract provides read-only functions to calculate vault data such as the collateral ratio, the equivalent ETH value of collateral/debt at the current exchange rates, weighted fees, etc.

In the following, we examine a specific library function `getNewMinimumRatio()`. This function is designed to returns the new minimum vault ratio due to a collateral deposit or withdraw. It is mainly used for checking the collateralization ratio is valid before performing an operation. Our analysis shows the current implementation is only appropriate when there is a collateral deposit. However, it is flawed when there is a collateral withdraw.

```
    function getNewMinimumRatio(
        address account,
        address fxToken,
        address collateralToken,
        uint256 collateralAmount,
        uint256 collateralQuote,
        bool isDeposit
    )
        public
        view
        override
        returns (uint256 ratio, uint256 newCollateralAsEther)
    {
        uint256 currentMinRatio = getMinimumRatio(account, fxToken);
        uint256 vaultCollateral =
            getTotalCollateralBalanceAsEth(account, fxToken);
        // Calculate new vault collateral from deposit amount.
        newCollateralAsEther = isDeposit
            ? vaultCollateral.add(
                collateralQuote.mul(collateralAmount).div(
                    getTokenUnit(collateralToken)
                )
            )
            : vaultCollateral.sub(
                collateralQuote.mul(collateralAmount).div(
                    getTokenUnit(collateralToken)
                )
            );
        uint256 depositCollateralMintCR =
            handle.getCollateralDetails(collateralToken).mintCR;
        if (currentMinRatio == 0) {
            ratio = depositCollateralMintCR.mul(1 ether).div(100);
        } else {
            /* Ratio for the current share of minimum collateral ratio due
            to the deposit amount (i.e. if vault holds $50 and the new
            deposit is $50, this value is 50% expressed as 0.5 ether). */
            uint256 oldCollateralMintRatio =
```

```
559                 vaultCollateral.mul(1 ether).div(newCollateralAsEther);
560             // Calculate new minimum ratio using the CR ratio above.
561             ratio = currentMinRatio
562                 .mul(oldCollateralMintRatio)
563                 .div(1 ether)
564                 .add(
565                 uint256(1 ether)
566                     .sub(oldCollateralMintRatio)
567                     .mul(depositCollateralMintCR)
568                     .div(1 ether)
569             );
570         }
571     }
```

Listing 3.8: `VaultLibrary::getNewMinimumRatio()`

Specifically, the internal variable `oldCollateralMintRatio` is used to represent the ratio for the current share of minimum collateral ratio due to the deposit change. In a collateral deposit scenario, this variable is no larger than `1 eth`, which yields the proper ratio. However, in a collateral withdraw scenario, this variable is no smaller than `1 eth`, which can easily result in reverting the execution due to the `SafeMath` operation on `uint256(1 ether).sub(oldCollateralMintRatio)` (lines 565-566).

Recommendation Accommodate both scenarios of collateral changes in `getNewMinimumRatio()`.

Status The issue has been fixed by the following commits: `84aaa97` and `d31e141`.

3.8 Improved Arithmetic Calculation

- ID: PVE-008
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [3]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `Comptroller::burn()` as an example. This routine is used to burn the `fxToken` debt from the sender's vault. We notice the fee calculation (lines 337 – 344) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `fee = amount.mul(handle.burnFeePerMille()).mul(quote)div(1000)..div (vaultLibrary.getTokenUnit(token))`. Similarly, the calculation of `getMinimumRatio()` in `VaultLibrary` contract (lines $373 - 379$) can be accordingly adjusted. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

```
    function burn(
        uint256 amount,
        address token,
        uint256 deadline
    ) external override dueBy(deadline) validFxToken(token) notPaused {
        require(amount > 0, "IA");
        // Token balance must be higher or equal than burn amount.
        require(IfxToken(token).balanceOf(msg.sender) >= amount, "IA");
        uint256 quote = handle.getTokenPrice(token);

        {
            // Treasury debt must be higher or equal to burn amount.
            uint256 maxAmount = handle.getDebt(msg.sender, token);
            if (amount > maxAmount) amount = maxAmount;
            if (amount != maxAmount)
                _ensureMinimumMintingAmount(
                    msg.sender,
                    token,
                    quote,
                    amount,
                    false
                );
        }

        // Update interest rates according to cache time.
        IInterest(handle.interest()).tryUpdateRates();

        // Store balance for assertion purposes.
        uint256 balanceBefore = IfxToken(token).balanceOf(msg.sender);

        // Charge burn fee as collateral Ether equivalent of fxToken amount.
        uint256 fee =
            amount
                .mul(handle.burnFeePerMille())
            // Cancel out fee ratio unit after fee multiplication.
                .div(1000)
                .mul(quote)
            // Cancel out token unit after price multiplication.
                .div(vaultLibrary.getTokenUnit(token));
        // Withdraw any available collateral type for fee.
        treasury.forceWithdrawAnyCollateral(
            msg.sender,
            handle.FeeRecipient(),
            fee,
```

```
            token,
            true
        );

        // Burn tokens
        IfxToken(token).burn(msg.sender, amount);
        assert(
            IfxToken(token).balanceOf(msg.sender) == balanceBefore.sub(amount)
        );

        // Update debt position
        uint256 debtPositionBefore = handle.getDebt(msg.sender, token);
        handle.updateDebtPosition(msg.sender, amount, token, false);
        assert(
            handle.getDebt(msg.sender, token) == debtPositionBefore.sub(amount)
        );

        emit BurnToken(amount, token);
    }
```

Listing 3.9: `Comptroller::burn()`

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed by this commit: `2d5e04c`.

3.9 Strengthened Reentrancy Prevention in Comptroller

- ID: PVE-009
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

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 is an occasion where the `checks-effects-interactions` principle is violated. Using the `fxKeeperPool` as an example, the `_withdrawCollateralRewardFrom()` 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`.

Apparently, the interaction with the external contract (line 194) starts before effecting the update on internal states (lines 196−198), 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.

```
177     function _withdrawCollateralRewardFrom(address account, address fxToken)
178         private
179     {
180         if (pools[fxToken].snapshot.P == 0) return;
181         // Withdraw all collateral rewards.
182         (
183             address[] memory collateralTokens,
184             uint256[] memory collateralAmounts
185         ) = balanceOfRewards(account, fxToken);
186         assert(collateralTokens.length > 0);
187         uint256 j = collateralTokens.length;
188         for (uint256 i = 0; i < j; i++) {
189             if (collateralAmounts[i] == 0) continue;
190             uint256 collateralBalance =
191                 pools[fxToken].collateralBalances[collateralTokens[i]];
192             // If the reward is greater than the pool amount, ignore loop iteration.
193             if (collateralBalance < collateralAmounts[i]) continue;
194             IERC20(collateralTokens[i]).transfer(account, collateralAmounts[i]);
195             // Update total collateral balance.
196             pools[fxToken].collateralBalances[
197                 collateralTokens[i]
198             ] = collateralBalance.sub(collateralAmounts[i]);
199         }
200         emit Withdraw(account, fxToken);
201     }
```

Listing 3.10: `fxKeeperPool::_withdrawCollateralRewardFrom()`

In the meantime, we should mention that 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`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`. Note similar issues exist in other contracts, including `Pool::deposit()` and the adherence of `checks-effects-interactions` best practice is recommended in a number of related routines, e.g., `mintWithoutCollateral()`, `burn()`, `buyCollateral()`, `buyCollateralFromManyVaults()` etc.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `d93d22e`.

3.10 Proper Debt Absorb in absorbDebt()

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `fxKeeperPool`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

Within the `handle.fi` protocol, there is a scalable pool that is designed to collectively fund liquidations. The pool holders share potential loss from the liquidation and are also potentially rewarded with liquidated collaterals. While examining the current debt-socializing logic, we notice the current implementation can be improved.

To elaborate, we show below the full implementation of the `absorbDebt()` function. It is designed to update various pool parameters after performing a liquidation. It comes to our attention that the `totalDeposits` is not updated until the debt loss has been socialized to all share holders. For better accuracy, it is suggested to reduce the `totalDeposits` before socializing the debt.

```
    function absorbDebt(
        uint256 debt,
        address[] memory collateralTypes,
        uint256[] memory collateralAmounts,
        address fxToken
    ) private {
        if (pools[fxToken].totalDeposits == 0  debt == 0) return;
        // Increase pool collateral balances.
        uint256 l = collateralTypes.length;
        for (uint256 i = 0; i < l; i++) {
            if (collateralAmounts[i] == 0) continue;
            pools[fxToken].collateralBalances[collateralTypes[i]] = pools[
                fxToken
            ]
                .collateralBalances[collateralTypes[i]]
                .add(collateralAmounts[i]);
        }
        _updateFxLossPerUnitStaked(
            debt,
            collateralTypes,
            collateralAmounts,
            fxToken
        );
        _updateCollateralGainSums(collateralTypes, collateralAmounts, fxToken);
        _updateSnapshotValues(debt, fxToken);
        pools[fxToken].totalDeposits = pools[fxToken].totalDeposits.sub(debt);
```

```
    }
```

Listing 3.11: `fxKeeperPool::absorbDebt()`

Recommendation Revise the `absorbDebt()` implementation to properly socialize the debt loss to all share holders.

Status The issue has been fixed by this commit: `5df058c`.

3.11 Proper Debt Absorb in absorbDebt()

- ID: PVE-011
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Treasury`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

Within the `handle.fi` protocol, there is a core contract `Treasury` that holds all protocol funds and supports user opertions on depositing and withdrawing their collaterals. While examining the current collateral-withdrawing logic, we notice the current implementation needs to be improved.

To elaborate, we show below the full implementation of the `withdrawCollateral()` function. It is designed to withdraw collateral from the sender's account. It comes to our attention that the current implementation does not timely invoke `IInterest(handle.interest()).tryUpdateRates()` to update interest rates according to cache time, which may lead to an inaccurate calculation of user balance.

```
    /**
     * @dev Withdraws collateral from the sender's account
     * @param collateralToken The collateral token to withdraw
     * @param to The address to remit to
     * @param amount The amount of collateral to withdraw
     * @param fxToken The vault fxToken
     */
    function withdrawCollateral(
        address collateralToken,
        address to,
        uint256 amount,
        address fxToken
    ) external override nonReentrant {
        _withdrawCollateralFrom(
            msg.sender,
            collateralToken,
            to,
```

```
            amount,
            fxToken
        );
    }
```

Listing 3.12: `Treasury::withdrawCollateral()`

Recommendation Revise the `withdrawCollateral()` implementation to timely update the interest rate.

Status The issue has been fixed by this commit: `4abdff2`.

3.12 Improved Logic in PCT::ensureUpperBoundLimit()

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `PCT`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

To improve the capital efficiency, the `handle.fi` protocol further supports the integration of external protocols for increased return. Accordingly, it maintains necessary accounting to keep track of the investment and gains about each external investment protocol. While examining the current accounting logic, we notice a specific function fails to properly maintain the accounting.

To elaborate, we show below the full implementation of the `ensureUpperBoundLimit()` function, which checks the `Treasury`'s collateral balance and total invested funds against maximum upper bound and withdraws from external protocol into `Treasury` if needed. Our analysis shows that it properly withdraws the funds from the external investment protocol and reduce the `totalInvestments` state. However, it fails to update the `protocolInvestments` state and it may corrupt the execution of a number of related functions, e.g., `withdrawProtocolFunds()`.

```
    function ensureUpperBoundLimit(
        IPCTProtocolInterface pi,
        address collateralToken
    ) private {
        Pool storage pool = pools[collateralToken];
        uint256 totalInvested = pool.protocolInvestments[address(pi)];
        uint256 totalFunds =
            IERC20(collateralToken).balanceOf(address(treasury)).add(
                totalInvested
            );
        uint256 upperBound = handle.pctCollateralUpperBound();
```

```
        uint256 maxInvestmentAmount = totalFunds.mul(upperBound).div(1 ether);
        if (totalInvested <= maxInvestmentAmount) return;
        // Upper bound limit has been exceeded; withdraw from external protocol.
        uint256 diff = totalInvested.sub(maxInvestmentAmount);
        pi.withdraw(diff);
        pool.totalInvestments = pool.totalInvestments.sub(diff);
    }
```

Listing 3.13: `PCT::ensureUpperBoundLimit()`

Recommendation Revise the `ensureUpperBoundLimit()` implementation to properly keep track of the investment-related accounting.

Status The issue has been fixed by this commit: `351005b`.

3.13 Inconsistent Deposit Fee Calculation And Collection

- ID: PVE-011
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Comptroller`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

As mentioned in Section 3.3, DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `handle.fi` protocol also supports a number of fee-related parameters, such as `withdrawFeePerMille` and `depositFeePerMille`. During the analysis of the related fee collection logic, we notice unnecessary inconsistency on the use of `depositFeePerMille`.

To elaborate, we show below the related `_mintAndDeposit()` function. The purpose of this function is to mint the requested amount (with necessary accounting for associated mint fees) and deposit collateral into the vault via the `Treasury`. It comes to our attention that the `feeCollateral` is calculated as `feeCollateral = collateralAmount.mul(handle.depositFeePerMille()).div(1000)` (lines 176-177).

```
162     function _mintAndDeposit(
163         uint256 tokenAmount,
164         address token,
165         address collateralToken,
166         uint256 collateralAmount,
167         address referral
168     ) private {
169         assert(
170             IERC20(collateralToken).approve(address(treasury), collateralAmount)
171         );
172
```

```
173         // Calculate fee with current amount and increase token amount to include fee.
174         uint256 feeTokens = tokenAmount.mul(handle.mintFeePerMille()).div(1000);
175 
176         uint256 feeCollateral =
177             collateralAmount.mul(handle.depositFeePerMille()).div(1000);
178         collateralAmount = collateralAmount.sub(feeCollateral);
179 
180         uint256 tokenQuote = handle.getTokenPrice(token);
181 
182         _ensureMinimumMintingAmount(
183             msg.sender,
184             token,
185             tokenQuote,
186             tokenAmount,
187             true
188         );
189 
190         require(
191             vaultLibrary.canMint(
192                 msg.sender,
193                 token,
194                 collateralToken,
195                 collateralAmount,
196                 tokenAmount.add(feeTokens),
197                 tokenQuote,
198                 handle.getTokenPrice(collateralToken)
199             ),
200             "CR"
201         );
202 
203         // Deposit in the treasury
204         treasury.depositCollateral(
205             msg.sender,
206             collateralAmount.add(feeCollateral),
207             collateralToken,
208             token,
209             referral
210         );
211 
212         _mint(tokenAmount, token, tokenQuote, feeTokens);
213     }
```

Listing 3.14: `Comptroller::_mintAndDeposit()`

From another perspective, if we follow the execution logic and analyze the invoked `Treasury::depositCollateral()` function, the same `feeCollateral` is calculated as `fee = depositAmount.mul(handle.depositFeePerMille()).div(1000)` (line 177) where `depositAmount=collateralAmount.add(feeCollateral)`. In other words, the `feeCollateral` amount is counted as part of `depositAmount` for the fee calculation again!

```
142     function _depositCollateral(
143         address from,
```

```
        address to,
        uint256 depositAmount,
        address collateralToken,
        address fxToken
    ) private {
        require(handle.isCollateralValid(collateralToken), "IC");

        // Ensure Treasury has self-allowance on ERC20 to wrap for the user.
        // This is needed on Arbitrum for ETH->WETH deposits.
        if (
            from == address(this) &&
            IERC20(collateralToken).allowance(address(this), address(this)) <
            depositAmount
        ) IERC20(collateralToken).approve(address(this), 2**256 - 1);

        // Ensure that this deposit won't result in the total ETH cap being
        uint256 newTotalEthDeposits =
            totalCollateralDeposited.add(
                depositAmount.mul(handle.getTokenPrice(collateralToken)).div(
                    vaultLibrary.getTokenUnit(collateralToken)
                )
            );
        require(
            maximumTotalDepositAllowed == 0
                newTotalEthDeposits <= maximumTotalDepositAllowed,
            "IA"
        );
        totalCollateralDeposited = newTotalEthDeposits;

        // Update interest rates according to cache time.
        IInterest(handle.interest()).tryUpdateRates();

        // Calculate fee and actual deposit amount.
        uint256 fee = depositAmount.mul(handle.depositFeePerMille()).div(1000);
        depositAmount = depositAmount.sub(fee);

        // Transfer collateral into the treasury
        require(
            IERC20(collateralToken).transferFrom(
                from,
                address(this),
                depositAmount
            ),
            "FT"
        );

        handle.updateCollateralBalance(
            to,
            depositAmount,
            fxToken,
            collateralToken,
            true
```

```
        );

        // Transfer fee.
        IERC20(collateralToken).transferFrom(from, handle.FeeRecipient(), fee);

        // Stake into PCT.
        handleStakingPCT(to, fxToken, collateralToken, depositAmount, true);
    }
```

Listing 3.15: `Treasury::_depositCollateral()`

Recommendation Be consistent in the above related functions for proper fee computation and collection.

Status The issue has been fixed by this commit: `67db24b`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `handle.fi` protocol, which is a decentralized multi-currency stablecoin creation and exchange protocol. The protocol allows users to create (borrow) and convert between multi-currency stablecoins called `fxTokens`. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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

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

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

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

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