SMART CONTRACT AUDIT REPORT

for

Fluid Protocol

Prepared By: Xiaomi Huang

PeckShield

November 10, 2023

Document Properties

Client	Instadapp
Title	Smart Contract Audit Report
Target	Fluid
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Jinzhuo Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	November 10, 2023	Xuxian Jiang	Final Release
1.0-rc	October 30, 2023	Xuxian Jiang	Release Candidate

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 `Fluid` 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 Fluid

`Fluid` aims to culminate existing lending protocols and transform the lending and borrowing space. It has a unique base `Liquidity` layer, which serves as the foundation upon which other protocols can be built by solving liquidity fragmentation. Innovative initial protocols are built on top, including `lending market` and `vault`. The former allows users to lend and earn while the latter innovates on the borrowing space with distinct features, e.g., higher LTV and much lower liquidation penalty. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Fluid

Item	Description
Target	Fluid
Website	https://instadapp.io/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 10, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the `Fluid` protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit. And the current code base is still under active revision.

- https://github.com/Instadapp/fluidity-contracts.git (5b3bfd1)

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

- https://github.com/Instadapp/fluidity-contracts.git (7a0cac2)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

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

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the 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) [11], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this 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 1.3: 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

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 `Fluid` implementation. 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	4	■■■■
Medium	4	■■■■
Low	5	■■■■■
Informational	0	
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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 4 high-severity vulnerabilities, 4 medium-severity vulnerabilities, and 5 low-severity vulnerabilities.

Table 2.1: Key Fluid Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Public Exposure of Privileged Functions in vault/admin/main	Security Features	Resolved
PVE-002	High	Incorrect Price Scaling in ChainlinkOracleImpl	Business Logic	Resolved
PVE-003	Low	Incorrect Interest Rate Computation in LiquidityCalcs	Coding Practices	Resolved
PVE-004	Low	Incorrect Rebalance Logic in VaultT1	Business Logic	Resolved
PVE-005	Low	Timely Interest Collection Upon Rate Module Change	Business Logic	Resolved
PVE-006	Low	Precision Issue in Asset Withdrawal Logic	Numeric Errors	Resolved
PVE-007	Medium	Conflicted Reentrancy Protection in iTokenEIP2612Deposits	Time and State	Resolved
PVE-008	Low	Incorrect Vault NFT Minting Logic in VaultT1Factory	Business Logic	Resolved
PVE-009	High	Revisited Collateral Factor Calculation in VaultT1	Coding Practices	Resolved
PVE-010	High	Improper Position Ownership Validation in VaultT1	Business Logic	Resolved
PVE-011	Medium	Improper Branch Debt Liquidity Update in VaultT1	Business Logic	Resolved
PVE-012	Medium	Improved User Debt Liquidation Logic in VaultT1	Business Logic	Resolved
PVE-013	Medium	Trust Issue of 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 Improper Public Exposure of Token-Approving Function

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `VaultAdmin`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Fluid` protocol, there is a `VaultAdmin` contract that is designed to manage the vault administration. However, we notice a number of privileged functions are publicly exposed, which need to be restricted to trusted callers only.

To elaborate, we show below the example admin-related functions from the `VaultAdmin` contract. By design, they are used to configure various aspects of the deployed vaults. These functions should be restricted to trusted callers, instead of being exposed publicly.

```
    modifier _updateExchangePrice() {
        IVault(address(this)).updateExchangePriceOnStorage();
        _;
    }

    function updateSupplyRateMagnifier(
        uint16 supplyRateMagnifier_
    ) public _updateExchangePrice {

        vaultVariables2 =
            (vaultVariables2 & 0
                xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff0000) |
                supplyRateMagnifier_;

        emit LogUpdateSupplyRateMagnifier(supplyRateMagnifier_);
    }

    function updateBorrowRateMagnifier(
```

```
        uint16 borrowRateMagnifier_
    ) public _updateExchangePrice {

        vaultVariables2 =
            (vaultVariables2 & 0
                xffffffffffffffffffffffffffffffffffffffffffffffffffffffff0000ffff) | (
                borrowRateMagnifier_ << 16);

        emit LogUpdateBorrowRateMagnifier(borrowRateMagnifier_);
    }

    function updateCollateralFactor(
        uint16 collateralFactor_
    ) public _updateExchangePrice {
        vaultVariables2 =
            (vaultVariables2 & 0
                xffffffffffffffffffffffffffffffffffffffffffffffffffff0000ffffffff) | (
                collateralFactor_ << 32);

        emit LogUpdateCollateralFactor(collateralFactor_);
    }

    function updateLiquidationThreshold(
        uint16 liquidationThreshold_
    ) public _updateExchangePrice {
        vaultVariables2 =
            (vaultVariables2 & 0
                xffffffffffffffffffffffffffffffffffffffffffffffff0000ffffffffffff) | (
                liquidationThreshold_ << 48);

        emit LogUpdateLiquidationThreshold(liquidationThreshold_);
    }
```

Listing 3.1: Example Administration Functions in `VaultAdmin`

Recommendation Validate the callers to the above-mentioned functions in `VaultAdmin`.

Status The issue has been addressed by applying the following PR: `152`.

3.2 Incorrect Price Scaling in ChainlinkOracleImpl

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `ChainlinkOracleImpl`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

Oracles are a critical component of any lending and borrowing protocol. The vault in `Fluid` utilizes an oracle system that combines `Uniswap` and `Chainlink` to ensure the most reliable and accurate pricing data. Our analysis shows current oracle integration has an issue in computing the price with incorrect scaling.

To elaborate, we show below the related code snippet of the `getChainlinkExchangeRate()` routine. As the name indicates, this routine returns the exchange rate from `Chainlink` oracle. However, when `CHAINLINK_INVERT_RATE` is `true`, the inverted price should be `_invertChainlinkPrice(uint256(exchangeRate_))`, not `_invertChainlinkPrice(uint256(exchangeRate_))* (10 ** CHAINLINK_PRICE_SCALER_DECIMALS)` (line 39). The reason is that the `_invertChainlinkPrice()` helper makes the internal adjustment based on the required price scaling, without the need of external adjustment again.

```
34    function getChainlinkExchangeRate() public view returns (uint256 rate_) {
35        (, int256 exchangeRate_, , , ) = FEED.latestRoundData();
36
37        // Return the price in units of wei
38        if (CHAINLINK_INVERT_RATE) {
39            return _invertChainlinkPrice(uint256(exchangeRate_)) * (10 **
                  CHAINLINK_PRICE_SCALER_DECIMALS);
40        } else {
41            return uint256(exchangeRate_) * (10 ** CHAINLINK_PRICE_SCALER_DECIMALS);
42        }
43    }
```

Listing 3.2: `ChainlinkOracleImpl:getChainlinkExchangeRate()`

Note other two routines in `uniV3OracleImpl`, i.e., `_getPriceFromSqrtPriceX96()` and `_invertUniV3Price()`, can also benefit from similar scaling adjustment.

Recommendation Improve the above-mentioned routines by returning the queried prices with correct scaling.

Status The issue has been addressed in the following commit: `2396cca`.

3.3 Incorrect Interest Rate Computation in LiquidityCalcs

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LiquidityCalcs`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

In the `Fluid` protocol, the `LiquidityCalcs` contract is a library contract that consolidates liquidity-related computation. In the process of examining current interest rate logic, we notice its implementation can be improved.

To illustrate, we show below the affected routine `calcRateV2()`. This routine is designed to calculate the borrow rate based on utilization for rate data version 2 (with two `kinks`) in $1e4$ precision. However, these two `kinks` have 16 bits each, instead of 20 bits (lines 461-462). The incorrect `kinks` may lead to wrongfully calculated borrow rate, which undermines the correctness of both lending and borrowing functionalities.

```
437     function calcRateV2(uint256 rateData_, uint256 utilization_) internal pure returns (uint256 rate_) {
438         /// For rate v2 (two kinks) -----------------------------------------------------
439         /// Next 16 bits =>  4 - 19 => Rate at utilization 0% (in 1e2: 100% = 10_000; 1% = 100 -> max value 65535)
440         /// Next 16 bits =>  20- 35 => Utilization at kink1 (in 1e2: 100% = 10_000; 1% = 100 -> max value 65535)
441         /// Next 16 bits =>  36- 51 => Rate at utilization kink1 (in 1e2: 100% = 10_000; 1% = 100 -> max value 65535)
442         /// Next 16 bits =>  52- 67 => Utilization at kink2 (in 1e2: 100% = 10_000; 1% = 100 -> max value 65535)
443         /// Next 16 bits =>  68- 83 => Rate at utilization kink2 (in 1e2: 100% = 10_000; 1% = 100 -> max value 65535)
444         /// Next 16 bits =>  84- 99 => Rate at utilization 100% (in 1e2: 100% = 10_000; 1% = 100 -> max value 65535)
445         /// Last 156 bits => 100-255 => blank, might come in use in future
446 
447         // y = mx + c.
448         // y is borrow rate
449         // x is utilization
450         // m = slope (m can be 0 but never negative)
451         // c is constant (c can be negative)
452 
453         uint256 y1_;
454         uint256 y2_;
455         uint256 x1_;
456         uint256 x2_;
```

```
        // extract kink1: 16 bits (0xFFFF) starting from bit 20
        // extract kink2: 52 bits (0xFFFF) starting from bit 20
        // kink is in 1e2, same as utilization, so no conversion needed for direct comparison of the two
        uint256 kink1_ = ((rateData_ >> 20) & 0xFFFFF);
        uint256 kink2_ = ((rateData_ >> 52) & 0xFFFFF);
        if (utilization_ < kink1_) {
            // if utilization is less than kink1
            y1_ = ((rateData_ >> 4) & X16);
            y2_ = ((rateData_ >> 36) & X16);
            x1_ = 0; // 0%
            x2_ = kink1_;
        }
        ...
    }
```

Listing 3.3: `LiquidityCalcs::calcRateV2()`

Recommendation Revise the above `calcRateV2()` routine by computing the right `kinks` for borrow rate calculation.

Status The issue has been addressed in the following commit: `22b0144`.

3.4 Incorrect Rebalance Logic in VaultT1

- ID: PVE-004
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: `VaultT1`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The `vault` has a built-in rebalancing logic to synchronize between the underlying liquidity and vault balance. In the process of examining the rebalancing logic, we notice the computed rebalance amount may have a wrong orientation.

To elaborate, we show below the code snippet from the `rebalance()` routine. This code snippet basically checks the balance between liquidity and vault. When the vault has more expected balance than liquidity, there is a need to fetch tokens from rebalancer and supply in liquidity. On the reverse side, when the vault has less balance than liquidity, we need to withdraw from liquidity and send to rebalancer. It comes to our attention that the liquidity withdrawal should be given the amount of `-int256(totalSupplyLiquidity_ - totalSupplyVault_)`, not current `int256(totalSupplyLiquidity_ - totalSupplyVault_)` (line 1289).

```
        if (totalSupplyVault_ > totalSupplyLiquidity_) {
            // Fetch tokens from revenue/rebalance contract and supply in liquidity contract
            // This is the scenario when the supply rewards are going in vault. Hence the vault total supply is increasing at a higher pace than Liquidity contract.
            // We are not transferring rewards right when we set the rewards to keep things clean.
            // Also, this can also happen in case when supply factor is greater than 1.
            LIQUIDITY.operate(
                ILiquidityOperateParams.OperateParams({
                    token: SUPPLY_TOKEN,
                    supplyAmount: int256(totalSupplyVault_ - totalSupplyLiquidity_),
                    borrowAmount: 0,
                    withdrawTo: address(0),
                    borrowTo: address(0),
                    callbackData: abi.encode(rebalancer)
                })
            );
        } else if (totalSupplyLiquidity_ > totalSupplyVault_) {
            // Withdraw from Liquidity contract and send it to revenue contract.
            // This is the scenario when the vault user's are getting less ETH APR then what's going on Liquidity contract.
            // When supplyFactor is less than 1.
            LIQUIDITY.operate(
                ILiquidityOperateParams.OperateParams({
                    token: SUPPLY_TOKEN,
                    supplyAmount: int256(totalSupplyLiquidity_ - totalSupplyVault_),
                    borrowAmount: 0,
                    withdrawTo: rebalancer,
                    borrowTo: address(0),
                    callbackData: new bytes(0)
                })
            );
        }
```

Listing 3.4: `VaultT1::rebalance()`

Recommendation Revise the above `rebalance()` routine to properly withdraw liquidity. The same issue is also applicable to transfer from the rebalance contract and payback on liquidity contract.

Status

3.5 Timely Interest Collection Upon Rate Module Change

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AuthModule
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The Fluid protocol has an unified liquidity layer that enables the deployment of unique features on top. The liquidity layer allows for the adjustment of borrow/supply rate models. While these rate models are being adjusted, we notice the lack of timely refreshment on the fee or interest collection.

To elaborate, we show below an example updateRateDataV1s() routine. This routine allows to adjust the internal kinks as well as associated utilization rates, which may greatly affect the borrow rate computation. Therefore, when they are changed, there is a need to timely refresh the fee collection before the new rate model can be applied. Note this issue also affects other routines, including AuthModule::updateRateDataV2s() and iTokenAdmin::updateRewards().

```
    function updateRateDataV1s(RateDataV1Params[] calldata tokensRateData_) external
        onlyAuths {
        uint256 length_ = tokensRateData_.length;

        for (uint256 i; i < length_; ) {
            if (tokensRateData_[i].token == address(0)) {
                revert AddressZero();
            }

            _rateData[tokensRateData_[i].token] = _computeRateDataPackedV1(
                tokensRateData_[i]);

            unchecked {
                i++;
            }
        }

        emit LogUpdateRateDataV1s(tokensRateData_);
    }
```

Listing 3.5: AuthModule::updateRateDataV1s()

Recommendation Timely collect the fee or interest before the new rate model is deployed and activated.

Status The issue has been addressed in the following commits: b9f8bb6 and bd7c053.

3.6 Precision Issue in Asset Withdrawal Logic

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: iTokenCore
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

Description

The lending market built on top of Fluid is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying users, i.e., mint()/redeem(). While reviewing the redeem logic, we notice the current implementation has a precision issue.

To elaborate, we show below the related _executeWithdraw() routine. As the name indicates, this routine is designed to withdraw assets by burning the owned market share. When the user indicates the underlying asset amount (via assetsWithdrawn_), the respective sharesBurned_ is computed as (assetsWithdrawn_ * EXCHANGE_PRICES_PRECISION)/ newTokenExchangePrice_ (line 260). Unfortunately, the current approach may unintentionally introduce a precision issue by computing the sharesBurned_ amount against the protocol. Specifically, the resulting flooring-based division introduces a precision loss, which may be just a small number but plays a critical role when certain boundary conditions are met – as demonstrated in the recent HundredFinance hack: https://blog.hundred.finance/15-04-23-hundred-finance-hack-post-mortem-d895b618cf33.

```
240     function _executeWithdraw(
241         uint256 assets_,
242         address receiver_,
243         address owner_
244     ) internal virtual validAddress(receiver_) returns (uint256 assetsWithdrawn_,
            uint256 sharesBurned_) {
245         uint256 liquidityExchangePrice_;

247         // withdraw from liquidity directly to _receiver. requires nonReentrant!
                otherwise ERC777s could reenter
248         (liquidityExchangePrice_, assetsWithdrawn_) = _withdrawFromLiquidity(assets_,
                receiver_);

250         // Check for rounding error
251         if (assetsWithdrawn_ == 0) {
252             revert iToken__RoundingError();
253         }

255         // update the exchange prices
256         uint256 newTokenExchangePrice_ = _updateRates(liquidityExchangePrice_, false);
```

```
        // not using previewWithdraw here because we just got newTokenExchangePrice_
        // burn shares for actually withdrawn assets_ amount
        sharesBurned_ = (assetsWithdrawn_ * EXCHANGE_PRICES_PRECISION) /
            newTokenExchangePrice_;

        // Check for rounding error
        if (sharesBurned_ == 0) {
            revert iToken__RoundingError();
        }

        _burn(owner_, sharesBurned_);

        emit Withdraw(msg.sender, receiver_, owner_, assetsWithdrawn_, sharesBurned_);
    }
```

Listing 3.6: `iTokenCore::_executeWithdraw()`

Recommendation Properly revise the above routine to ensure the precision loss needs to be computed in favor of the protocol, instead of the user. In particular, as a precaution, we need to ensure that markets are never empty by minting small shares at the time of market creation so that we can prevent the rounding error being used maliciously.

Status The issue has been addressed in the following commits: `e404534`, `9cb9204`, `dc2de35`, `16587f0`, and `4a1b390`.

3.7 Conflicted Reentrancy Protection in iTokenEIP2612Deposits

- ID: PVE-007
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `iTokenEIP2612Deposits`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [4]

Description

To mitigate potential reentrancy issues, the `Fluid` protocol makes extensive use of `nonReentrant` modifier to detect and block reentrancy attempts. However, we notice the presence of potentially in-conflict reentrancy protection, which should be accordingly improved.

To elaborate, we show below the implementation of the `depositWithSignature()` function. It has a `nonReentrant` modifier and its function body further calls the `iTokenActions::call()` which also has the `nonReentrant` modifier. As a result, the intended `depositWithSignature()` function for the `EIP2612` support does not work as expected.

```
    function depositWithSignature(
        uint256 assets_,
        address receiver_,
        uint256 minAmountOut_,
        uint256 deadline_,
        bytes calldata signature_
    ) external nonReentrant returns (uint256 shares_) {
        // create allowance through signature_ and spend it. ‘nonReentrant‘ modifier present so this is ok to happen
        // after
        (uint8 v_, bytes32 r_, bytes32 s_) = _splitSignature(signature_);

        // EIP-2612 permit for underlying asset from owner (msg.sender) to spender (this contract)
        IERC20Permit(address(ASSET)).permit(msg.sender, address(this), assets_, deadline_, v_, r_, s_);

        shares_ = deposit(assets_, receiver_);
        if (shares_ < minAmountOut_) {
            revert iToken__MinAmountOut();
        }
    }
```

Listing 3.7: `iTokenEIP2612Deposits::depositWithSignature()`

```
    function deposit(
        uint256 assets_,
        address receiver_
    ) public virtual override nonReentrant returns (uint256 shares_) {
```

```
        if (assets_ == type(uint256).max) {
            assets_ = ASSET.balanceOf(msg.sender);
        }

        // @dev transfer of tokens from ‘msg.sender‘ to liquidity contract happens via ‘
            liquidityCallback‘
        (, shares_) = _executeDeposit(assets_, receiver_, abi.encode(msg.sender));
    }
```

Listing 3.8: `iTokenActions::deposit()`

Recommendation Remove the `nonReentrant` modifier from the `depositWithSignature()` function.

Status The issue has been addressed in the following commit: `89a6bb2`.

3.8 Incorrect Vault NFT Minting Logic in VaultT1Factory

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VaultT1Factory`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

As mentioned earlier, the `Fluid` protocol has an unified liquidity layer that enables the deployment of unique features on top. While examining the `vault` deployment via the `VaultT1Factory` contract, we notice the `NFT` tokenization of a user position should be improved.

To elaborate, we show below the related `mint()` function. This routine is designed to mint a new `NFT` to the given user with the associated `vaultId_`. However, it comes to our attention that the internal `_mint()` helper was given a wrong `tokenId_` as `vaultId_` (line 244).

```
240     function mint(uint256 vaultId_, address user_) external returns (uint256 tokenId_) {
241         if (msg.sender != getVaultAddress(vaultId_)) revert VaultT1Factory__InvalidVault();
242
243         // Using _mint() instead of _safeMint() to allow any msg.sender to receive ERC721 without onERC721Received holder.
244         tokenId_ = _mint(user_, tokenId_);
245
246         emit NewPositionMinted(msg.sender, user_, tokenId_);
247     }
```

Listing 3.9: `VaultT1Factory::mint()`

```
    function _mint(address to_, uint256 vaultId_) internal virtual returns(uint256 id_) {
        if (to_ == address(0)) revert ERC721__InvalidParams();

        unchecked {
            totalSupply++;
        }

        id_ = totalSupply;
        if (id_ >= type(uint32).max || _tokenConfig[id_] != 0) revert ERC721__InvalidParams();

        _transfer(address(0), to_, id_, vaultId_);


        emit Transfer(address(0), to_, id_);
    }
```

Listing 3.10: `ERC721::_mint()`

Recommendation Correct the above `mint()` function with the right `vaultId_`.

Status The issue has been addressed in the following commit: `e8672ff`.

3.9 Revisited Collateral Factor Calculation in VaultT1

- ID: PVE-009
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `VaultT1`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

The `vault` support in `Fluid` has a unified entry function `operate()` to perform lending-related operations, i.e., supply, borrow, withdraw, and payback. Naturally, the `vault` needs to enforce an invariant, i.e., a borrower will not be able to borrow more than allowed based on the deposited collateral and associated collateral factor. While assessing this borrow invariant, we notice current implementation incorrectly applies the collateral factor and thus compromise the invariant.

To elaborate, we show below the code snippet from the `operate()` routine. This code snippet basically computes `tickAtCF` based on the specified collateral factor. We notice the collateral factor is extracted from `((o_.vaultVariables2 >> 32)& 0xffff)` (line 346), which is normalized with 4 decimals. As a result, the correct `tickAtCF` should be further scaled down by `10000` (line 349).

```
335     // if debt is greater than 0 & transaction is not just deposit, payback or deposit & payback
336     if (o_.debtRaw > 0 && !(newCol_ >= 0 && newDebt_ <= 0)) {
337         // Oracle returns price at 100% ratio.
338         // converting oracle 160 bits into oracle address
339         // temp_ => debt price w.r.t to col in 1e18
340         temp_ = IOracle(address(uint160(o_.vaultVariables2 >> 96))).getExchangeRate();
341         // Converting price in terms of raw amounts
342
343         temp_ = (temp_ * o_.supplyExPrice) / o_.borrowExPrice;
344         // temp2_ => ratio at CF
345
346         temp2_ = temp_ * ((o_.vaultVariables2 >> 32) & 0xffff);
347         // Price from oracle is in 1e18 decimals. Converting it into (1 << 96) decimals
348
349         temp2_ = (temp2_ * (1 << 96)) / 1e18;
350
351         // temp3_ => tickAtCF_
352         temp3_ = TickMath.getTickAtRatio(temp2_);
353         if (o_.tick > temp3_) {
354             if (o_.oldTick > o_.tick || (o_.debtRaw - o_.dustDebtRaw) > o_.oldNetDebtRaw ) {
```

```
                // Above CF, user should only be allowed to reduce ratio either by
                    paying debt or by depositing more collateral
                // Not comparing collateral as user can potentially use safe/deleverage
                    to reduce tick & debt.
                // On use of safe/deleverage, collateral will decrease but debt will
                    decrease as well making the overall position safer.
                revert VaultT1__PositionAboveCF();
            }
        }
    }
```

Listing 3.11: `VaultT1::operate()`

Recommendation Revise the above `operate()` routine to properly enforce the borrow invariant.

Status The issue has been addressed by applying the following PR: `149`.

3.10 Improper Position Ownership Validation in VaultT1

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `VaultT1`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

As mentioned in Section 3.10, the `vault` has a unified entry function `operate()` to perform lending-related operations, i.e., supply, borrow, withdraw, and payback. Also, each `vault` position is tokenized as an `NFT`. While examining the borrow-related functionality, we notice the enforcement to validate the caller (i.e., it is initiated by the owner) is incorrectly implemented.

To elaborate, we show below the code snippet from the `operate()` routine. This code snippet basically validates the caller to be the `NFT` owner if the operation involves more than deposit and payback (lines 95 – 99). However, the `if`-condition should be `if !(newCol_ >= 0 && newDebt_ <= 0)`, not current `if (newCol_ >= 0 && newDebt_ <= 0)` (line 99).

```
84          {
85              // Fetching user's position
86              if (nftId_ == 0) {
87                  // creating new position.
88                  o_.tick = type(int).min;
89                  // minting new NFT vault for user.
90                  nftId_ = VAULT_FACTORY.mint(VAULT_ID, msg.sender);
91              } else {
92                  // Updating existing position
```

```

                // not checking owner in case of just deposit & payback
                if (newCol_ >= 0 && newDebt_ <= 0) {
                    if (VAULT_FACTORY.ownerOf(nftId_) != msg.sender) {
                        revert VaultT1__NotAnOwner();
                    }
                }

                // temp_ => user's position data
                temp_ = positionData[nftId_];

                if (temp_ == 0) {
                    revert VaultT1__InvalidOperateAmount();
                }

                temp2_ = (temp_ >> 45) & X64;
                // Converting big number into normal number
                o_.colRaw = (temp2_ >> 8) << (temp2_ & 0xff);
                // Converting big number into normal number
                temp2_ = (temp_ >> 109) & X64;
                o_.dustDebtRaw = (temp2_ >> 8) << (temp2_ & 0xff);

                // 1 is supply & 0 is borrow
                if (temp_ & 1 == 1) {
                    // only supply position (has no debt)
                    o_.tick = type(int).min;
                } else {
                    // borrow position (has collateral & debt)
                    o_.tick = temp_ & 2 == 2
                        ? int((temp_ >> 2) & 0x7ffff)
                        : -int((temp_ >> 2) & 0x7ffff);
                    o_.tickId = (temp_ >> 21) & 0xffffff;
                }
            }
        }
```

Listing 3.12: `VaultT1::operate()`

Recommendation Revise the above `operate()` routine to properly validate the `NFT` owner if the operation involves more than deposit and payback.

Status The issue has been addressed in the following commit: `fdc3f77`.

3.11 Improper Branch Debt Liquidity Update in VaultT1

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VaultT1`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The `vault` in `Fluid` innovatively allocates a user's position to a specific tick (determined by their debt-to-collateral ratio). When a user changes its position, there is a need to adjust its allocation accordingly. Our analysis shows the position allocation incorrectly updates the associated branch data.

To elaborate, we show below the related code snippet from the `operate()` routine. This code snippet kicks in when the user position is partially liquidated and there is a need to recompute the latest user position (line 147). Since the user position is being adjusted, we need to withdraw the liquidity from the final branch being liquidated where the position exists and later add the liquidity to the new position (or tick). However, the liquidity removal from the previous branch reduces the branch liquidity (line 155) and the reduction is incorrectly reflected in the `branchData` (line 157) with a wrong mask `0xffffffffffffffffffffffffff000000000000000000000000fffffffffffff`, which should be `0xffffffffffffffffffffffffffffffffff0000000000000000fffffffffffff`.

```
137         // Checking if tick is liquidated OR if the total IDs of tick is greater than
                user's tick ID
138         if (((temp_ & 1) == 1) || (((temp_ >> 1) & 0xffffff) > o_.tickId)) {
139             // User got liquidated
140             (
141                 // returns the position of the user if the user got liquidated then it
                    returns the new position of user.
142                 o_.tick,
143                 o_.debtRaw,
144                 o_.colRaw,
145                 temp2_, // final branch from liquidation where position exist right now
146                 o_.branchData
147             ) = fetchLatestPosition(o_.tick, o_.tickId, o_.debtRaw, temp_);
148
149             if (o_.debtRaw > o_.dustDebtRaw) {
150                 // temp_ => branch's Debt
151                 temp_ = (o_.branchData >> 52) & X64;
152                 temp_ = (temp_ >> 8) << (temp_ & 0xff);
153
154                 // TODO: Make sure to check that debtToRemove_ should always be < branch
                    's Debt (temp_). Else function will fail
155                 temp_ -= o_.debtRaw;
```

```
                temp_ = temp_.toBigNumberPure(56, 8);
                branchData[temp2_] =
                    (o_.branchData & 0
                        xfffffffffffffffffffffffffff000000000000000000000000fffffffffffff
                        ) |
                    (temp_ << 52);

                // Converted positionRawDebt_ in net position debt
                o_.debtRaw -= o_.dustDebtRaw;
            } else {
                // Liquidated 100%
                o_.debtRaw = 0;
                o_.colRaw = 0;
            }
            o_.dustDebtRaw = 0;
        }
```

Listing 3.13: `VaultT1::operate()`

Recommendation Revise the above `operate()` routine to properly removal liquidity from the affected branch.

Status The issue has been addressed in the following commit: `fdc3f77`.

3.12 Improved User Debt Liquidation Logic in VaultT1

- ID: PVE-012
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VaultT1`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The `vault` innovates the borrowing space with a borrower-friendly liquidation mechanism. Specifically, any trader of any size can liquidate any amount of debt and there is no requirement to liquidate the entire bad debt at once. While examining the liquidation-related functionality, we notice there are a number of revisions that can be done to improve current implementation.

The first issue is related to the `minimaTick` extraction from the current branch being liquidated (line 544 – 546). The extraction is based on the ternary operator that makes use of an uninitialized `temp_`, hence yielding always negative `minimaTick`.

```
530         {
531             // ############# Setting current branch in memory #############
532
```

```
            // Updating branch related data
            branch_.id = (vaultVariables_ >> 22) & 0x3fffffff;
            branch_.data = branchData[branch_.id];
            branch_.debtFactor = (branch_.data >> 116) & X50;
            if (branch_.debtFactor == 0) {
                // Initializing branch debt factor. 35 | 15 bit number. Where full 35
                    bits and 15th bit is occupied.
                // Making the total number as (2**35 - 1) << 2**14.
                branch_.debtFactor = ((0x7fffffff << 15) | (1 << 14));
            }
            // If branch is liquidated then only it'll have minima tick
            if ((vaultVariables_ & 2) == 2) {
                branch_.minimaTick = (temp_ & 4) == 4
                    ? int256((branch_.data >> 3) & 0x7ffff)
                    : -int256((branch_.data >> 3) & 0x7ffff);
            } else {
                branch_.minimaTick = type(int).min;
            }
        }
```

Listing 3.14: `VaultT1::liquidate()`

The second issue involves the `tickHasDebt_.nextTick` assignment when the top tick is not liquidated (lines 646 – 650). Specifically, when current tick in liquidation is a perfect tick, the same tick is used to fetch next perfect tick as `tickHasDebt_.nextTick = currentData_.tick`, not `tickHasDebt_.nextTick == currentData_.tick` (line 650).

```
633         if (currentData_.debtRemaining > 0) {
634             // Stores liquidated debt & collateral in each loop
635             uint debtLiquidated_;
636             uint colLiquidated_;
637             uint debtFactor_ = 1e18;
638 
639             TickHasDebt memory tickHasDebt_;
640             tickHasDebt_.mapId = (currentData_.tick < 0)
641                 ? (((currentData_.tick + 1) / 256) - 1)
642                 : (currentData_.tick / 256);
643 
644             tickInfo_.ratio = TickMath.getRatioAtTick(int24(tickInfo_.tick));
645 
646             if (currentData_.tickStatus == 1) {
647                 // top tick is not liquidated. Hence it's a perfect tick.
648                 currentData_.ratio = tickInfo_.ratio;
649                 // if current tick in liquidation is a perfect tick then fetching this
                        will allow to fetch next perfect tick
650                 tickHasDebt_.nextTick == currentData_.tick;
651             } else {
652                 // top tick is liquidated. Hence it's has partials.
653                 tickInfo_.ratioOneLess = (tickInfo_.ratio * 10000) / 10015;
654                 tickInfo_.length = tickInfo_.ratio - tickInfo_.ratioOneLess;
655                 tickInfo_.partials = (branch_.data >> 22) & X30;
```

```
                currentData_.ratio = tickInfo_.ratioOneLess + ((tickInfo_.length * tickInfo_.partials) / X30);
            }
            ...
        }
```

Listing 3.15: `VaultT1::liquidate()`

The third issue is about the branch data update with a wrong `connect factor` offset. It occurs when the debt is being liquidated so that the associated branch will be adjusted for respective liquidity removal. In particular, the correct `connect factor` offset should be 116, not current 112 (line 890).

```
876     {
877         uint newBranchDebtFactor_ = (temp2_ >> 116) & X50;
878 
879         // connectionFactor_ = baseBranchDebtFactor / currentBranchDebtFactor
880         uint connectionFactor_ = BigMath.divBigNumber(
881             newBranchDebtFactor_,
882             branch_.debtFactor,
883             35,
884             15,
885             96, // precision
886             16384 // decimals
887         );
888 
889         // Updating current branch in storage
890         branchData[branch_.id] = (((branch_.data >> 166) << 166) | (connectionFactor_ << 112) | 2);
891 
892         // Storing base branch in memory
893         // Updating branch ID to base branch ID
894         branch_.id = temp_;
895         // Updating branch data with base branch data
896         branch_.data = temp2_;
897         // Remove next branch connection from base branch
898         branch_.debtFactor = newBranchDebtFactor_;
899         // minima tick of base branch
900         branch_.minimaTick = (temp2_ & 4) == 4
901             ? int256((temp2_ >> 3) & 0x7ffff)
902             : -int256((temp2_ >> 3) & 0x7ffff);
903     }
```

Listing 3.16: `VaultT1::liquidate()`

Recommendation Resolve the above-mentioned issues in the debt liquidation logic.

Status The issue has been addressed by following the above suggestion.

3.13 Trust Issue of Admin Keys

- ID: PVE-013
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Fluid` protocol, there are a series of privileged accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters and update price oracle). In the following, we show the representative functions potentially affected by the privilege of the accounts.

```
    function updateAuths(AddressBool[] calldata authsStatus_) external onlyGovernance {
        uint256 length_ = authsStatus_.length;
        for (uint256 i; i < length_; ) {
            if (authsStatus_[i].addr == address(0)) {
                revert AddressZero();
            }

            uint256 setStatus_ = authsStatus_[i].value ? 1 : 0;

            _isAuth[authsStatus_[i].addr] = setStatus_;

            unchecked {
                i++;
            }
        }

        emit LogUpdateAuths(authsStatus_);
    }

    /// @inheritdoc ILiquidityAdmin
    function updateGuardians(AddressBool[] calldata guardiansStatus_) external
        onlyGovernance {
        uint256 length_ = guardiansStatus_.length;
        for (uint256 i; i < length_; ) {
            if (guardiansStatus_[i].addr == address(0)) {
                revert AddressZero();
            }

            uint256 setStatus_ = guardiansStatus_[i].value ? 1 : 0;

            _isGuardian[guardiansStatus_[i].addr] = setStatus_;

            unchecked {
                i++;
```

```
            }
        }

        emit LogUpdateGuardians(guardiansStatus_);
    }
```

Listing 3.17: Example Privileged Operations in `GovernanceModule`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. The `multi-sig` mechanism could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest to introduce the `multi-sig` mechanism to manage all the privileged accounts to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks. Note the same issue is also applicable to the proxy upgrade as the current protocol is deployed behind a proxy.

Status The issue has been confirmed by the team. The teams intends to make use of `multi-sig` to mitigate this issue.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Fluid` protocol, which aims to culminate existing lending protocols and transform the lending and borrowing space. It has a unique base `Liquidity` layer, which serves as the foundation upon which other protocols can be built by solving liquidity fragmentation. Innovative initial protocols are built on top, including `lending market` and `vault`. The former allows users to lend and earn while the latter innovates on the borrowing space with distinct features, e.g., higher LTV and lowest liquidation penalty. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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