

STATE MIND

Instadapp Liquidity Layer Update

10-09-2025 - 14-10-2025



1. Project brief		2
2. Finding severity breakdown		4
3. Summary of findings		5
4. Conclusion		5
5. Findings report		6
High	An incorrect net transfer calculation leads to overpayment to the user	6
Informational	Inconsistent callbackData_ parsing creates compatability and validation risks	7
	Redundant conditions in if statements	8
	FluidLiquidityUserModule.operate() implicitly limits revenue contributions to 1%, restricting integration flexibility	9
	SafeTransfer.safeTransferNative() implicit 20000 gas limit may cause unexpected reverts in receive/fallback	10

1. Project brief



Title	Description
Client	Instadapp
Project name	Instadapp Liquidity Layer Update
Timeline	10-09-2025 - 14-10-2025

Project Log

Date	Commit Hash	Note
10-09-2025	23692b02dc20aa87aa59f2bcd8bb8ec4ad9234bf	Initial Commit
02-10-2025	7f86e96dbbd0f3569c638d198f567c392ed9ba64	Reaudit commit
14-10-2025	151ced3ff5d6079e9bc598b71ccce25933a3580b	Reaudit commit 2

Short Overview

Liquidity Layer is the foundational layer of Fluid. It functions as a central hub that consolidates liquidity from all integrated protocols. The Liquidity Layer provides deposit, withdrawal, borrowing, and repayment operations. End users interact with the protocols, and the protocols interact with the Liquidity Layer.

New functionality in the updated version:

- Decay of limits. After a supply, the withdrawal limit gradually decreases by the portion of the supplied amount that extended the withdrawal limit beyond its maximum expansion.
- Net and skip transfers. These mechanisms optimize the work with transfers. When, within a single interaction with the Liquidity Layer, tokens both arrive and are sent out, only the net difference is transferred. If the amounts are equal, no transfer occurs.


Project Scope

The audit covered the following files:

 [bigMathMinified.sol](#)

 [safeTransfer.sol](#)

 [variables.sol](#)

 [dummyImpl.sol](#)

 [events.sol](#)

 [liquidityCalcs.sol](#)

 [main.sol](#)

 [structs.sol](#)


 [helpers.sol](#)

 [proxy.sol](#)

 [liquiditySlotsLink.sol](#)

 [main.sol](#)

 [errorTypes.sol](#)

 [events.sol](#)

 [error.sol](#)

2. Finding severity breakdown



All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings



Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	1 (1 fixed, 0 acknowledged)
Medium	0 (0 fixed, 0 acknowledged)
Informational	4 (1 fixed, 3 acknowledged)
Total	5 (2 fixed, 3 acknowledged)

4. Conclusion



During the audit of the codebase, 5 issues were found in total:

- 1 high severity issue (1 fixed)
- 4 informational severity issues (1 fixed, 3 acknowledged)

The final reviewed commit is `151ced3ff5d6079e9bc598b71ccce25933a3580b`

5. Findings report



HIGH-01

An incorrect net transfer calculation leads to overpayment to the user

Fixed at:
[f13ee79](#)

Description

Lines: [liquidity/userModule/main.sol#L664-L666](#)

In **NET_TRANSFERS** mode, the **FluidLiquidityUserModule.operate()** function is expected to perform netting of **operateIn** and **operateOut**, with **netTransfersOut = operateOut - operateIn**. However, due to the order of operations in the current implementation, **operateIn (memVar2_)** is zeroed before the net calculation. As a result, **netTransfersOut = operateOut - 0**, causing users to receive an excess amount equal to the previously unaccounted **operateIn**. Internally, the LiquidityLayer accounting remains correct, but users receive more tokens than intended.

Recommendation

We recommend adjusting the sequence of operations so that **operateIn** is accounted for before setting it to zero. This ensures that **netTransfersOut** reflects the correct netted value.

```
o_.netTransfersOut = int256(memVar_ - memVar2_);  
memVar2_ = 0;
```

INFORMATIONAL-01	Inconsistent <code>callbackData_</code> parsing creates compatability and validation risks	Acknowledged
------------------	--	--------------

Description

The `callbackData_` parameter is parsed inconsistently across different functions:

`CoreInternals._isInOutBalancedOut()` and `CoreInternals._isNetTransfers()` read the last 64 bytes as `[ID, inFrom]`

`CoreInternals._checkEnforceTotalInputAmount()` read first word as amount/identifier, third word as amount when identifier present.

DEX V1: `[amount][bool][inFrom]`

DEX V2: `[PROTOCOL][ACTION][amount][...]`

When `callbackData_.length > 95`, `FluidLiquidityUserModule.operate()` enforces prefix parsing via

`CoreInternals._checkEnforceTotalInputAmount()`, but the same data must pass suffix parsing in

`CoreInternals._isInOutBalancedOut()` or `CoreInternals._isNetTransfers()`. Constructing callback data that satisfies both incompatible parsing schemes is effectively hard for integrators.

Recommendation

We recommend introducing an explicit, per-user layout version in user configuration and decoding strictly by that version, keeping DEX V1 unchanged.

- Add `uint8 dataLayoutVersion` in user configuration variable, in user config (`0 = DEX_V1`, `1 = DEX_V2`, ... other values reserved for future data layouts).
- In `operate()`, select the decode path based on `dataLayoutVersion`:
 - For `DEX_V1`: preserve the existing schema `[flag, inFrom]` and skip amount enforcement.
 - For `DEX_V2/UNIFIED`: adopt a documented schema and decode consistently.
- If the version is unset, default to `DEX_V1` for backward compatibility, or proactively set versions for known legacy users.

Client's comments

We have control over which protocols get to interact with the LL so the integrators is the team itself. We already have plans to simplify how transfers are handled on the LL in a future version. We want to standardize that all newly developed protocols will follow the structure of `[PROTOCOL][ACTION]` in the first 2 words of `callbackData`, the rest is protocol-specific.

We had thought of a per-user layout version or key mapping etc., but ultimately decided for going only with this `callbackData` way because it is the simplest.

Description

Lines:

- [liquidity/userModule/main.sol#L470-L478](#)
 - [liquidity/adminModule/main.sol#L613](#)
- 1.

```
operateAmountOut_ =
  (borrowAmount_ > 0 ? borrowAmount_ : 0) +
  (supplyAmount_ < 0 ? -supplyAmount_ : 0)
if (
  supplyAmount_ == 0 ||
  borrowAmount_ == 0 ||
  operateAmountOut_ == 0 ||
  (supplyAmount_ > 0 && borrowAmount_ < 0) ||
  (supplyAmount_ < 0 && borrowAmount_ > 0)
){
```

Given the formula used for computing **operateAmountOut_**, the condition **operateAmountOut_ == 0** is satisfied only when **borrowAmount_ <= 0 && supplyAmount_ >= 0**. However, this case is already fully covered by other parts of the if statement: **supplyAmount_ == 0 || borrowAmount_ == 0 || (supplyAmount_ > 0 && borrowAmount_ < 0)**.

Furthermore, the condition **(supplyAmount_ > 0 && borrowAmount_ < 0) || (supplyAmount_ < 0 && borrowAmount_ > 0)** can be simplified to **(supplyAmount_ ^ borrowAmount_) < 0**, which directly checks for different signs.

2.

```
if (newLimit_ == type(uint256).max || newLimit_ > userSupply_) {
  newLimit_ = userSupply_;
}
```

The condition **newLimit_ == type(uint256).max** is unnecessary, as **newLimit_ > userSupply_** already captures all relevant cases, except when **userSupply_ == type(uint256).max**. In that scenario, executing the branch has no meaningful effect.

Recommendation

We recommend changing the if statements to save gas:

1. liquidity/userModule/main.sol

```
if (
  supplyAmount_ == 0 ||
  borrowAmount_ == 0 ||
- operateAmountOut_ == 0 ||
- (supplyAmount_ > 0 && borrowAmount_ < 0) ||
- (supplyAmount_ < 0 && borrowAmount_ > 0)
+ (supplyAmount_ ^ borrowAmount_) < 0
){
```

2. liquidity/adminModule/main.sol

```
if (newLimit_ == 0 || newLimit_ < maxExpansionLimit_) {
  newLimit_ = maxExpansionLimit_;
- } else if (newLimit_ == type(uint256).max || newLimit_ > userSupply_) {
+ } else if (newLimit_ > userSupply_) {
  newLimit_ = userSupply_;
}
```

INFORMATIONAL-03	FluidLiquidityUserModule.operate() implicitly limits revenue contributions to 1%, restricting integration flexibility	Acknowledged
------------------	---	--------------

Description

Lines:

- [liquidity/userModule/main.sol#L527-L528](#)
- [liquidity/userModule/main.sol#L542-L543](#)
- [liquidity/userModule/main.sol#L677-L678](#)
- [liquidity/userModule/main.sol#L702-L704](#)

The current **FluidLiquidityUserModule.operate()** design allows the caller to transfer an **amountIn** up to 1% above the operation amount. Any excess is routed to the LiquidityLayer's revenue. Implicitly, this enforces a 1% cap on revenue contributions.

This design restricts both current operations and future integrations:

- The calling protocol cannot send the full intended revenue along with the operation if it exceeds 1% of the operation amount.
- Integrated protocol revenue may exceed 1% in certain scenarios (e.g., high protocol fees), requiring additional logic in the integration contract.
- Patterns where the calling protocol accumulates revenue/fees and sends the full amount later (not on every **FluidLiquidityUserModule.operate()** call) complicate the caller's logic.

Recommendation

We recommend encoding a **uint256 revenueAmount** in **bytes callbackData**. This allows the LiquidityLayer to precisely handle both the operation amount and revenue, without implicitly limiting the revenue to 1%. For backward compatibility with dexV1, in the absence of a **revenueAmount**, the contract can accept revenue 1% from the operated amount.

Client's comments

We will indirectly do the same in DexV2 by passing in the total amount to be transferred, through which in combination with operateAmounts we could derive the exact revenue amount

Standardization of these things for future protocol development is something we are working on. We will keep this in mind for the next upgrades on the LL.

The 1% cap is intended for now as there is no case where this would be crossed. Protocols that interact with the LL are under our control so we don't need to be supporting every possible scenario as if it would be permissionless. The LL extends trust to the protocol and the current goal is to restrict at the LL as much as possible without impeding any realistic use-case of existing protocols. If the requirements on protocol side change over time, we will adjust the LL accordingly e.g. widening this cap to 3%

INFORMATIONAL-04	SafeTransfer.safeTransferNative() implicit 20000 gas limit may cause unexpected reverts in receive/fallback	Acknowledged
------------------	---	--------------

Description

Lines:

- [libraries/safeTransfer.sol#L8](#)
- [libraries/safeTransfer.sol#L90](#)

The **SafeTransfer.safeTransferNative()** function implicitly restricts the gas limit to 20000 when transferring native tokens. During calls to **FluidLiquidityUserModule.operate()**, this results in the calling contract being implicitly limited to 20000 gas in its **receive** or **fallback** functions when receiving native tokens from the **LiquidityLayer**.

While this restriction reduces the potential surface for reentrancy attacks, it is not a complete protection from all potential cases. Contracts can **deposit** or **payback** native tokens to the **LiquidityLayer** without gas limitations. However, **withdraw** or **borrow** operations may unexpectedly revert if the receiving contract includes internal logic in its native token handlers. This also limits the implementation of the revenue collector contract [liquidity/adminModule/main.sol#L388](#).

Recommendation

We recommend extending the user configuration for native tokens to include a configurable maximum gas limit. This allows administrators to set flexible restrictions depending on the user contract's logic.

Client's comments

We are actually planning to increase this limit to 50k gas to support the case where the receiving contract does a WETH deposit in receive().
Configurable maximum gas limit adds too much complexity, as the receiver usually is the end user (not the protocol like Vault protocol etc.)

**STATE
MIND**