

# INSTADAPP FLUID SECURITY AUDIT REPORT

Jun 25, 2024

MixBytes()

# TABLE OF CONTENTS

<b>1. INTRODUCTION</b>	2
1.1 Disclaimer	2
1.2 Security Assessment Methodology	2
1.3 Project Overview	6
1.4 Project Dashboard	7
1.5 Summary of findings	10
1.6 Conclusion	11
<b>2. FINDINGS REPORT</b>	13
2.1 Critical	13
2.2 High	13
2.3 Medium	13
M-1 Lack of supply limit	13
M-2 Insufficient reentrancy protection in <code>FluidVaultT1</code>	15
2.4 Low	17
L-1 <code>FluidVaultT1.liquidate()</code> reverts if <code>actualColAmt_</code> is zero	17
L-2 Fee-on-transfer and rebasable tokens are not supported	18
L-3 Cross-vault borrowing may cause liquidations to fail	19
L-4 Risk of revert in rate calculation	20
<b>3. ABOUT MIXBYTES</b>	21

# 1. INTRODUCTION

## 1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

## 1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

### 1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

#### Stage goals

- Build an independent view of the project's architecture.
- Identifying logical flaws.

### 2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

#### Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

### 3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

#### Stage goal

Detect inconsistencies with the desired model.

### 4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

#### Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

### 5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

### Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

## 6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

### Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

## 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.
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 funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer 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 Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

## 1.3 Project Overview

VaultT1 is a core component of the Fluid architecture, operating as a borrow/lending protocol. It enables users to establish collateral/borrow positions, with all funds managed via interactions with the central Liquidity contract. Each position is represented by an NFT minted through the VaultFactory. The deployment of VaultT1 involves linking it with the VaultT1 AdminModule and FluidVaultT1Secondary contracts. Furthermore, VaultT1 is connected to an oracle that assesses the value of collateral and debt, ensuring accurate calculations and risk management.

## 1.4 Project Dashboard

### Project Summary

Title	Description
Client	Instadapp
Project name	Fluid
Timeline	March 25 2024 - June 21 2024
Number of Auditors	4

### Project Log

Date	Commit Hash	Note
25.03.2024	75784793203c4be2cfa5476953ba0a7905ee79e1	Commit for the audit
25.04.2024	d3d9fd9e5d8a5bd1841dac600c92c00db8267775	Commit for the reaudit
14.05.2024	c4b49eb902eee75810b16e83aa9cf50cc357f86e	Commit with updates
17.06.2024	b466ffd4b69761be50851094ff624edaf1fa6c07	Commit with updates

### Project Scope

The audit covered the following files:

File name	Link
contracts/protocols/vault/vaultT1/adminModule/events.sol	events.sol
contracts/protocols/vault/vaultT1/adminModule/main.sol	main.sol

File name	Link
contracts/protocols/vault/vaultT1/common/variables.sol	variables.sol
contracts/protocols/vault/vaultT1/coreModule/constantVariables.sol	constantVariables.sol
contracts/protocols/vault/vaultT1/coreModule/events.sol	events.sol
contracts/protocols/vault/vaultT1/coreModule/helpers.sol	helpers.sol
contracts/protocols/vault/vaultT1/coreModule/main2.sol	main2.sol
contracts/protocols/vault/vaultT1/coreModule/main.sol	main.sol
contracts/protocols/vault/vaultT1/coreModule/structs.sol	structs.sol
(review) contracts/libraries/bigMathMinified.sol	bigMathMinified.sol
(review) contracts/libraries/bigMathVault.sol	bigMathVault.sol
(review) contracts/libraries/liquidityCalcs.sol	liquidityCalcs.sol
(review) contracts/libraries/liquiditySlotsLink.sol	liquiditySlotsLink.sol
(review) contracts/libraries/storageRead.sol	storageRead.sol
(review) contracts/libraries/tickMath.sol	tickMath.sol

## Deployments

File name	Contract deployed on mainnet	Comment
main.sol	0x0C8C77...E3DD6cB3	Vault_ETH_USDC
main.sol	0xE16A6f...E00B00eB	Vault_ETH_USDT
main.sol	0x82B27f...aA2548De	Vault_wstETH_ETH

File name	Contract deployed on mainnet	Comment
main.sol	0x1982CC...B28fdcc3	Vault_wstETH_USDC
main.sol	0xb4F3bf...E95Cc946	Vault_wstETH_USDT
main.sol	0xeAEf56...C35e028D	Vault_weETH_wstETH
main.sol	0x399646...2c8744Dd	Vault_sUSDe_USDC
main.sol	0xBc3452...3BA38Da5	Vault_sUSDe_USDT
main.sol	0xF2c8F5...552d0FA4	Vault_weETH_USDC
main.sol	0x92643E...acA6D5F1	Vault_weETH_USDT

## 1.5 Summary of findings

Severity	# of Findings
Critical	0
High	0
Medium	2
Low	4

ID	Name	Severity	Status
M-1	Lack of supply limit	Medium	Acknowledged
M-2	Insufficient reentrancy protection in <code>FluidVaultT1</code>	Medium	Fixed
L-1	<code>FluidVaultT1.liquidate()</code> reverts if <code>actualColAmt_</code> is zero	Low	Fixed
L-2	Fee-on-transfer and rebasable tokens are not supported	Low	Acknowledged
L-3	Cross-vault borrowing may cause liquidations to fail	Low	Acknowledged
L-4	Risk of revert in rate calculation	Low	Acknowledged

## 1.6 Conclusion

The Vault protocol by Fluid is a lending protocol that achieves high capital efficiency due to a highly efficient liquidation algorithm, which operates with ticks and branches.

This audit focused on the protocol's general architecture, interactions across vaults, big number math, tick math, user operations, and the liquidation algorithm. Special attention was given to handling edge cases during liquidations, position creation, and managing overflows and underflows when working with packed storage variables.

Key activities included:

1. Confirming the reduction of a position's Health Factor over time.
2. Conducting experiments with rounding errors in large numbers to ensure they do not lead to DOS or excessive accumulation of bad debt.
3. Testing the possibility of overflow in individual packed variables to alter subsequent variables in the slot.
4. Checking edge cases with ticks to ensure the correct handling of zero ticks and determining the minimum and maximum ticks a user's position can have.
5. Ensuring that the `absorb()` and `liquidate()` functions do not overlap in ticks.
6. Confirming that the health index of a user's position improves after partial liquidation, and that re-liquidation of the position is possible when the price falls further.
7. Confirming that interest accumulates on the branch and can be properly liquidated.
8. Fuzzing the creation of multiple branches and verifying that users can properly close their positions after liquidations, ensuring that the algorithm does not accumulate unaccounted debt, and confirming that debt accrues to the user and can be correctly liquidated.

Key Observations and Recommendations:

- General Architecture and Cross-Vault Interactions: The Vault lacks a supply cap, potentially rendering the liquidation unprofitable.
- Reentrancy Concerns: Cross-contract reentrancy is possible due to the violation of the check-effect-interaction pattern in Vault functions. Neither read-only functions nor the admin module have reentrancy protections. Strengthening these areas is crucial to safeguard certain types of DAO proposals and potential external integrations.
- Error Handling in Core Functions: The functions `absorb()`, `liquidate()`, and `operate()` in `FluidVaultT1` sometimes revert without clear error messages, complicating troubleshooting and user interaction. Enhancing error messaging will improve usability and facilitate debugging.

A thorough review against a detailed checklist revealed no dangerous vulnerabilities. The checklist covered aspects such as proxy and delegatecall risks, access control, calculation flaws, gas issues, assembly pitfalls and potential compiler bugs.

The protocol demonstrates a high degree of security. However, the code's optimization for maximum gas efficiency has necessitated sacrifices in clarity, significantly complicating the audit process. Developers should weigh optimization against maintainability and clarity to ease future audits and maintenance.

# 2. FINDINGS REPORT

## 2.1 Critical

Not Found

## 2.2 High

Not Found

## 2.3 Medium

<b>M-1</b>	Lack of supply limit
<b>Severity</b>	Medium
<b>Status</b>	Acknowledged

### Description

To profit from liquidations, liquidators must be able to sell collateral at a favorable price. However, selling a large amount of collateral may significantly impact the selling price, rendering liquidation unprofitable and generating bad debt.

A supply limit, configured to match market liquidity, could mitigate this issue. However, the supply limit has not been implemented yet. Although, the borrow limit is in place and may indirectly limit supply, it may not effectively address this issue due to substantial price fluctuations often associated with large volumes of liquidations.

### Recommendation

We recommend implementing a supply limit and tracking its value as per market conditions.

### Client's commentary

Interesting, we thought on this when designing the collateral, we believe with debt ceiling, supply will auto limit itself according to the max borrow limit. Adding supply limit adds another kind of risk for user which is when collateral price goes down user cannot add more collateral to make their position safer. Note: We will explore adding supply limit on liquidity layer. Liquidity layer is upgradable contract so it can be added any time in future if needed.

M-2

Insufficient reentrancy protection in `FluidVaultT1`

**Severity**

Medium

**Status**

Fixed in `c7f553b0`

## Description

Although the `FluidVaultT1` smart contract code includes reentrancy protection, some functions remain unprotected:

- admin functions
- read-only functions

### Admin functions

Admin functions can be called by the DAO. Once the DAO proposal is approved, it can be executed by *anyone*.

Here is a possible attack vector:

- An attacker calls `operate()`.
- The attacker receives a call to their malicious smart contract from `FluidVaultT1` while `operate()` is not finalized.
- The attacker's contract calls `DAO.executeProposal()` with, for example, an `absorbDustDebt()` call.
- The attacker allows the initial `operate()` to finish.

The issue is that `operate()` works with a copy of `Variables.vaultVariables`:

- `main.sol#L53`

and stores them back at the end of the function:

- `main.sol#L553`.

Thus, any modifications to `vaultVariables` in `absorbDustDebt()` will be overwritten by the later update in `operate()`, as the copied memory value `vaultVariables_` in `operate()` will remain unchanged after the update in `absorbDustDebt()`.

### Read-only functions

The same vector applies here, but it affects the reading of certain variables like `positionData` and `branchData`, which undergo some modifications before the call to the malicious address. Vulnerabilities

in this area typically appear in more complex smart contract integrations, for instance, when external protocols use read-only functions to integrate with Fluid. See an example at <https://rekt.news/midas-capital-rekt/>.

## Reentrancy spots

The `safeTransfer` library only requires success on `msg.value` transfer without gas limitations. As a result, it is possible to receive a call to a malicious smart contract in the line:

- [safeTransfer.sol#L88](#).

This function is used in the following lines:

- [main.sol#L515](#)
- [main.sol#L1122](#)
- [main.sol#L533-L540](#).

Additionally, tokens with hooks (like ERC-777) can open up new spots for reentrancy during fund transfers from a user.

## Recommendation

We recommend:

- developing a dedicated function to verify the current reentrancy status, replacing repeated lines preceding key functions;
- implementing this function across all functions, including admin and read-only functions (e.g., `fetchLatestPosition`), provided it does not disrupt the contract's logic.

A perfect solution would be to have reentrancy protection at a higher Fluid level – on the Liquidity smart contract. This would also protect against cross-vault and cross-protocol reentrancies.

## Client's commentary

Make sense! Added the re-entrancy check here: [PR-343](#)

## 2.4 Low

L-1

`FluidVaultT1.liquidate()` reverts if `actualColAmt_` is zero

**Severity** Low

**Status** Fixed in 10fbd201

### Description

- [main.sol#L1106](#)

If there are no ticks available for liquidation in the vault, `FluidVaultT1.liquidate()` reverts due to division by zero, which is not the expected behavior of the function.

### Recommendation

For better clarity in such cases, we recommend returning a custom error (e.g., «VaultT1\_\_NoDebt»).

### Client's Commentary

Make sense! A better clarity revert would be good. Applied it here: [PR-343](#)

<b>L-2</b>	Fee-on-transfer and rebasable tokens are not supported
<b>Severity</b>	Low
<b>Status</b>	Acknowledged

**Description**

The vault does not support fee-on-transfer and rebasable tokens. It means that vaults with such tokens can be created, but they may operate incorrectly.

**Recommendation**

Ensure that such tokens are not permitted or implement code to address special cases of these tokens.

**Client's Commentary**

This is known and codebase should not use these kind of tokens. This responsibility is of governance.

<b>L-3</b>	Cross-vault borrowing may cause liquidations to fail
<b>Severity</b>	Low
<b>Status</b>	Acknowledged

### Description

Multiple vaults can be created sharing the same `Liquidity` contract and storing funds there. Each vault operates as an isolated lending entity with a chosen pair of `BORROW_TOKEN` and `SUPPLY_TOKEN`.

As a result, it is possible to have two vaults where, for example, ETH is supplied in the first vault and borrowed in the second vault. If enough ETH is borrowed in the second vault, the first vault may be left with insufficient collateral, which is critical for executing liquidations.

### Recommendation

This design is likely intentional to ensure capital efficiency.

The simplest solution is to consider current borrow limits with this risk in mind. A more complex solution could involve introducing some reserved amounts that are guaranteed to be present on the Liquidity smart contract.

### Client's Commentary

The similar risk exists in almost all other lending protocols. The goal is to keep a rate curve such that when utilization is crossing certain kink the borrow rate should spike heavily which will force borrowers to payback their debt.

**L-4**

Risk of revert in rate calculation

**Severity**

Low

**Status**

Acknowledged

### Description

- [liquidityCalcs.sol#L566-L568](#)

In `liquidityCalcs.calcRateV1()`, it is required that the slope is non-negative:

```
if (slope_ < 0) {
    revert FluidLiquidityCalcsError(
        ErrorTypes.LiquidityCalcs__BorrowRateNegative);
}
```

If the protocol transitions to a state with a negative slope, then the `calcRateV1()` function and, consequently, the `userModule` function `operate()` that uses it will revert. This will temporarily prevent users from repaying debts, potentially leading to liquidation risks.

### Recommendation

We recommend capping the slope to zero value instead of reverting.

## 3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

### Contacts



[https://github.com/mixbytes/audits\\_public](https://github.com/mixbytes/audits_public)



<https://mixbytes.io/>



[hello@mixbytes.io](mailto:hello@mixbytes.io)



<https://twitter.com/mixbytes>