Fluid Dex V2 Audit Report

mkXploit

February 19, 2026

Fluid Dex V2 Audit Report

Version 0.1

mkXploit

February 23, 2026

Fluid Dex V2 Audit Report

mkXploit

February 19, 2026

Fluid Dex V2 Audit Report

Prepared by: mkXploit Lead Auditors:

- [mxXploit] https://github.com/mkXploit

Assisting Auditors:

- None

Table of contents

See table

About mkXploit

mkXploit is a smart contract security auditor specializing in identifying vulnerabilities and strengthening the security posture of decentralized applications. With deep expertise in Solidity, EVM mechanics, and blockchain security patterns, I conduct comprehensive audits across DeFi protocols, NFT platforms, DAOs, and other Web3 applications.

Disclaimer

The mkXploit team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

`904c2989aa404ecb9cf75eb1efa1a5fa526007b0`

Scope

fluid-contracts @ 904c2989aa404ecb9cf75eb1efa1a5fa526007b0 - DEX V2 Core - Money Market Core - Libraries

Protocol Summary

Fluid is a DeFi ecosystem by Instadapp with a three-layer architecture: Core Architecture 1. Liquidity Layer (Foundation)

Central vault holding all protocol funds Doesn't interact with users directly - only with built-on protocols Provides unified liquidity across the ecosystem

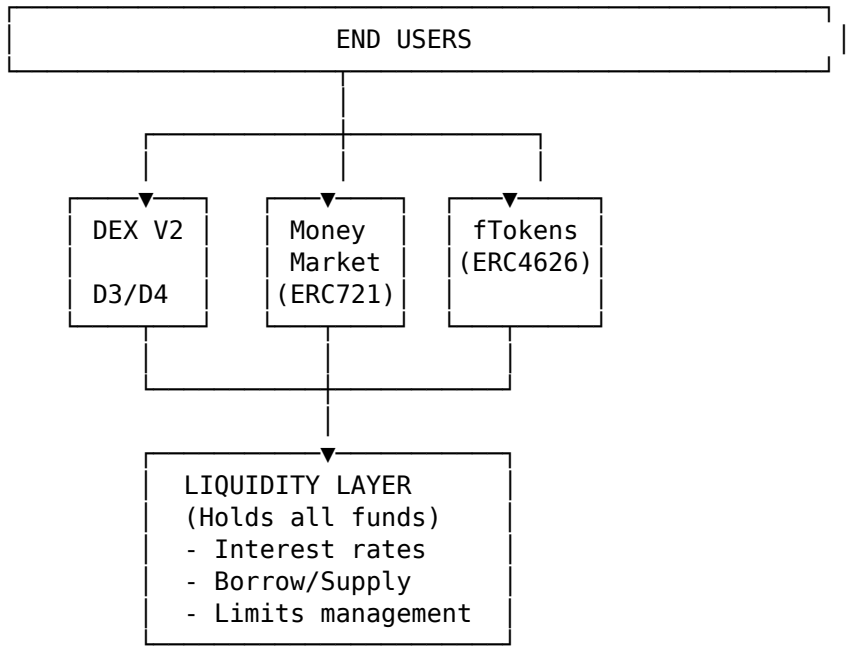
2. DEX V2 (The audit focus)

D3 Pools (Smart Collateral): Concentrated liquidity pools where LP positions can be used as collateral. Uses Uniswap V3-style ticks and will become permissionless. D4 Pools (Smart Debt): Similar to D3 but uses borrowed funds for leveraged liquidity provision. Remains permissioned and integrates with the Money Market. D1/D2 Pools: Basic liquidity pools (out of scope)

3. Money Market (Smart Lending)

ERC721-based positions (each position is an NFT) Supports 4 position types: Normal Supply, Normal Borrow, D3 LP positions, D4 LP positions Features efficiency modes for correlated assets and isolated collateral options Oracle-based liquidations

ARCHITECTURE



Key Design Principles:

1. **Modular Architecture** - Protocols can be added/upgraded independently
2. **Unified Liquidity** - All protocols share the same liquidity pool
3. **Gas Optimization** - Heavy use of bit packing, unchecked blocks, BigMath
4. **Upgradeable** - Uses ERC1967 UUPS proxy pattern
5. **No Direct User Interaction with Liquidity** - Only protocols interact with core

---

Roles

Role Hierarchy

```
GOVERNANCE (Highest Authority)
    │
    ├─► Can do EVERYTHING Auths can do
    ├─► Upgrade contracts (UUPS)
    ├─► Add/Remove Auths
    ├─► Change critical parameters
    ├─► Emergency controls
    │
    └─► AUTHS (Authorized Addresses)
            │
            ├─► Configure pools/markets
            ├─► Update fees and parameters
            ├─► Manage whitelists
            ├─► Execute rebalances (DEX V2)
            └─► Cannot upgrade contracts
```

Role Permissions Summary The protocol has a three-tier permission model: **GOVERNANCE** Storage: Liquidity Layer slot 0xb53...6103 Unique Powers:

Protocol upgrades (upgradeTo, upgradeToAndCall) Add/remove Auth addresses Change admin module implementations Emergency pause/unpause Inherits all Auth permissions

Key Functions: Contract upgrades, oracle updates, token/emode configuration, revenue collection

AUTHS (Operational Admins) Storage: Per-protocol mapping _isAuth[address] Operational Powers:

Configure pool parameters (fees, spreads, ranges) Manage user whitelists Update protocol fees and revenue cuts Execute DEX

rebalancing Stop per-pool accounting

Restrictions:

 Cannot upgrade contracts Cannot modify Auth list

USERS (Public) No Special Permissions Standard Capabilities:

Supply/borrow on Money Market (ERC721 positions) Provide liquidity (when whitelisted or in permissionless phase) Swap tokens Collect LP fees Liquidate undercollateralized positions

Executive Summary

Issues found

Severity	Number of issues found
High	0
Medium	1
Low	2
Info	0
Total	3

Findings

Medium — Inconsistent revert behavior between token0 and token1 fee processing may cause avoidable liquidation failures

Summary

The helper function `_calculateFeeCollectionAmounts()` applies inconsistent handling when accumulated fees are insufficient to satisfy the required withdrawal value:

- **Token0 branch:** consumes all available fees and continues execution
- **Token1 branch:** reverts the entire transaction under the same condition

This asymmetry may cause liquidations to revert even when partial fee coverage is available, reducing liquidation reliability and intro-

ducing non-deterministic execution outcomes based solely on token distribution.

Vulnerability Details

Location:

`contracts/protocols/moneyMarket/liquidateModule/helpers.sol`

When processing token0:

```
if (feeValue_ < withdrawValue_) {
    feeCollectionAmountToken0_ = feeAmountToken0_;
    withdrawValue_ -= feeValue_;
}
```

Execution continues to token1 processing.

However, token1 processing uses different logic:

```
if (feeValue_ < withdrawValue_) {
    revert FluidMoneyMarketError(
        ErrorTypes.LiquidateModule__InvalidParams
    );
}
```

As a result:

- Identical shortfall conditions lead to **continuation for token0**
- But **transaction revert for token1**

The outcome of liquidation therefore depends on how fees are distributed across tokens rather than on total available value.

Impact

This behavior may:

1. Cause otherwise viable liquidations to revert
2. Reduce liquidation success rate under stressed market conditions
3. Introduce unpredictable execution outcomes for liquidators
4. Increase gas expenditure due to failed attempts

While not directly causing loss of funds, reduced liquidation reliability can contribute to delayed bad-debt resolution and degraded protocol risk management.

Proof of Concept

Example scenario:

Parameter	Value
Required withdrawal value	100
Token0 fee value	60
Token1 fee value	30

Execution:

1. Token0 branch consumes 60 → remaining = 40
2. Token1 branch sees `30 < 40`
3. Transaction reverts

Despite 90% coverage, liquidation fails entirely.

Recommendation

Fee processing logic should be made consistent between tokens.

Allow partial coverage for both tokens Mirror token0 behavior:

```
if (feeValue_ < withdrawValue_) {
    feeCollectionAmountToken1_ = feeAmountToken1_;
    withdrawValue_ -= feeValue_;
}
```

Low — Missing transaction deadline validation in DEX V2 deposit and withdraw operations

Summary

The `deposit()` and `withdraw()` functions in the DEX V2 user modules do not enforce transaction expiry or deadline validation. As a result, transactions may be executed significantly later than intended by the user, exposing them to stale execution risk and MEV timing uncertainty.

Vulnerability Details

Affected locations:

```
contracts/protocols/dexV2/dexTypes/d3/core/userModule.sol
contracts/protocols/dexV2/dexTypes/d4/core/userModule.sol
```

Functions:

```
function deposit(DepositParams calldata params_) external
function withdraw(WithdrawParams calldata params_) external
```

The parameter structs lack a deadline field:

```
struct DepositParams {
    ...
    uint256 amount0Min;
    uint256 amount1Min;
}

struct WithdrawParams {
    ...
    uint256 amount0Min;
    uint256 amount1Min;
}
```

No validation similar to:

```
require(block.timestamp <= deadline);
```

is performed before execution.

Although minimum amount parameters provide price protection, they do not constrain execution timing.

Impact

Absence of deadline validation may:

1. Allow transactions to execute long after submission
2. Increase exposure to stale market conditions
3. Reduce user control over execution timing
4. Increase MEV timing uncertainty for liquidity operations

This does not directly enable theft of funds, but may degrade user execution quality and predictability.

Recommendation

Introduce optional deadline enforcement.

Struct update

```
struct DepositParams {
    ...
    uint256 deadline;
}

struct WithdrawParams {
    ...
    uint256 deadline;
}
```

Validation

```
if (block.timestamp > params_.deadline) {
    revert TransactionExpired();
}
```

This aligns behavior with established DEX interface patterns and improves execution predictability.

Low — Swap operations lack transaction expiry protection

Summary

The `swapIn()` and `swapOut()` functions in the DEX V2 swap modules do not implement deadline validation. Transactions may therefore execute at arbitrary future times, exposing users to stale execution and timing-based MEV uncertainty.

Vulnerability Details

Affected locations:

`contracts/protocols/dexV2/dexTypes/d3/core/swapModule.sol`
`contracts/protocols/dexV2/dexTypes/d4/core/swapModule.sol`

Functions:

```
swapIn(SwapInParams calldata params_)
swapOut(SwapOutParams calldata params_)
```

Parameter structs:

```
struct SwapInParams {
    ...
    uint256 amountOutMin;
}

struct SwapOutParams {
    ...
    uint256 amountInMax;
}
```

No deadline field is present, and no timestamp validation is performed prior to execution.

While slippage bounds constrain price execution, they do not limit transaction lifetime.

Impact

This may lead to:

1. Execution under stale market conditions
2. Reduced user control over transaction timing
3. Increased exposure to timing-based MEV selection
4. Less predictable trade execution outcomes

This issue does not directly compromise protocol funds but affects execution guarantees typically expected in modern swap interfaces.

Recommendation

Add deadline support consistent with common DEX implementations.

Struct update

```
struct SwapInParams {
    ...
    uint256 deadline;
}

struct SwapOutParams {
    ...
```

```
    uint256 deadline;
}
```

Validation

```
if (block.timestamp > params_.deadline) {
    revert TransactionExpired();
}
```

Providing expiry protection improves user control and aligns with industry-standard swap protections.
