Bitflow HODLMM Audit Report

Version 1.0

Lead Auditors

neumo

100proof

8 November 2025

Contents

1 About Greybeard Security

We are Greybeard Security, a team of independent security researchers focused on high-impact, high-integrity smart contract audits and exploit research. Our work emphasizes depth, precision, and transparency, with a track record of identifying critical vulnerabilities across major DeFi protocols.

Greybeard Security is:

100proof

- X
- Blog
- Immunefi Profile

neumo

- X
- Blog
- Immunefi Profile

2 Disclaimer

This audit is a limited security assessment of Bitflow's HODLMM smart contracts based on the code and documentation provided to the audit team. While every effort was made to identify vulnerabilities, bugs, and potential risks, no audit can guarantee the complete absence of issues.

This report does not constitute legal, financial, or investment advice. It should not be considered a warranty or certification of safety, nor a recommendation to use or interact with the protocol.

Smart contracts are inherently risky and may be subject to unexpected behaviors, including those introduced by third-party dependencies, network conditions, or economic exploits. Bitflow and any users of the HODLMM contracts are solely responsible for the continued testing, monitoring, and safe deployment of the protocol.

The audit team disclaims all liability for any loss or damage resulting from the use of, or reliance on, the findings presented in this report.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Protocol Summary

Bitflow's HODLMM (High-throughput, Orderbook-style Decentralized Liquidity Market Maker) is a concentrated liquidity engine purpose-built to expand Bitcoin capital markets. It brings orderbook-like precision to AMMs, allowing liquidity providers to allocate capital within specific price bins — enabling tighter spreads, zero-slippage trades, and more capital-efficient yield generation.

Inspired by the **Liquidity Book architecture by LFJ**, HODLMM adapts the concept of discrete price bins to Bitcoin markets, optimizing capital allocation for both traders and LPs. It offers a highly efficient alternative to traditional AMMs, which spread liquidity across an infinite price curve, leaving most of it unused and unproductive.

Key features of HODLMM include:

- **Price Bins**: Liquidity is segmented into discrete price ranges, enabling slippage-free trades within active bins.

- **Custom Strategy Shapes**: LPs can select from pre-built configurations (Spot, Curve, Bid-Ask) or define custom bin layouts aligned with their market views.
- **Keeper Automation**: Bitflow Keepers automate position rebalancing, fee harvesting, and strategy optimization — reducing the complexity of active liquidity management.
- **Dynamic Fee Logic**: Fees adjust based on volatility and activity, rewarding LPs for participating during high-value market moments.
- **Strategy Vaults & Auto Modes**: Simplified options for passive LPs who prefer hands-off exposure with set-and-forget strategies.

For traders, HODLMM delivers slippage-free execution, deep liquidity, and efficient routing across bins and pool types. For LPs, it offers high capital efficiency, granular control, and automated management — all tailored for assets users want to HODL.

By combining the capital efficiency of concentrated liquidity with the precision of orderbook mechanics — and building on innovations introduced by LFJ's Liquidity Book — Bitflow's HODLMM sets a new standard for Bitcoin-native liquidity infrastructure.

5 Audit Scope

We conducted a ***best effort*** security review of the DLLM contracts of Bitflow.

The size of the scope to review (see below) is too much for a one-week engagement, but due to lack of availability, we agreed with Bitflow to perform a best effort review.

The contracts in scope for this review are:

File	Blank	Comment	Code
./clarity/contracts/dlmm-core-v-1-1.clar	288	338	1383
./clarity/contracts/dlmm-pool-sbtc-usdc-v-1-1.clar	76	82	519
./clarity/contracts/dlmm-pool-trait-v-1-1.clar	2	3	108
Total	366	423	2010

Only the items listed in the Scope section above are in scope, and the following items are explicitly out of scope:

- Full test coverage for the entire repository or unrelated subsystems
- Frontend, backend, and off-chain infrastructure

6 Executive Summary

The Greybeard Security team performed an audit over 5 days, on the Bitflow HODLMM code provided by Bitflow. A total of 8 issues were found.

This report presents the findings from our audit of Bitflow's HODLMM — a concentrated liquidity engine designed to enhance capital efficiency and yield generation in Bitcoin DeFi markets.

HODLMM introduces orderbook-style mechanics into an AMM framework by segmenting liquidity into discrete price bins. Inspired by LFJ's Liquidity Book architecture, it allows LPs to deploy capital more precisely and earn higher returns with less exposure, while enabling traders to execute swaps with zero slippage inside active ranges. The system also includes an automation layer (Bitflow Keepers) that handles rebalancing, fee harvesting, and strategy optimization.

Our audit evaluated the correctness, security, and economic soundness of the core and pool HODLMM smart contracts.

Overall, the HODLMM codebase demonstrates a thoughtful design, strong modularity, and alignment with current best practices in DeFi protocol architecture. No critical vulnerabilities were identified. Several low- and medium-severity findings were reported, primarily related to edge-case behavior, gas efficiency, and operational clarity. These have been communicated to the Bitflow team along with actionable recommendations.

Bitflow has been responsive throughout the audit process, and has implemented or scheduled appropriate mitigations for the majority of the issues identified. Assuming full resolution of remaining items, the HODLMM contracts are on track for a safe and secure launch.

As with any complex DeFi system, ongoing testing, careful monitoring, and a commitment to security best practices will remain essential post-deployment.

Summary

Project Name	Bitflow HODLMM
Repository	bitflow-dlmm
Commit	59ecb3530279...
Audit Timeline	13 October - 17 October 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	1
Informational	3
Gas Optimizations	0
Total Issues	8

Summary of Findings

[M-1] Authentication via `tx-sender` opens up phishing attack	Resolved
[M-2] No upgrade path for core contract	Resolved
[M-3] After a bin cross an immediate swap in the reverse direction reverts because 0 tokens are required	Resolved
[M-4] With enough funds a user can pay tiny fees for swaps	Resolved
[L-1] Bin steps can be added with wrong values and they cannot be updated/removed	Acknowledged
[I-1] Errors in `transfer` function do not follow `SIP-013`	Resolved
[I-2] Multi-bin swaps using favourable bins can actually lead to worse outcomes	Acknowledged
[I-3] Verified pools can still use malicious tokens	Acknowledged

7 Medium Findings

[M-1] Authentication via `tx-sender` opens up phishing attack

Description

Authenticating the caller using `tx-sender` is generally considered bad practice, as documented in several sources, including the Clarity Book.

Allowing non-whitelisted, user-supplied traits to be passed into functions that rely on `tx-sender` for authentication opens the door to arbitrary (and potentially malicious) code execution on behalf of the initial contract caller. Any function that uses `tx-sender` for authentication must ensure that no user-supplied contracts are called in the entire execution path. Otherwise, an attacker could exploit this behavior.

For example, the `swap` functions in the core contract use `tx-sender` for authentication and may call user-supplied contracts (i.e., token contracts) that could be malicious. In this case, due to the design of token transfers in Stacks, `tx-sender` authentication is necessary, particularly for compatibility with router-based swap calls. Post-conditions protect user funds in this case. However, post-conditions do not prevent malicious tokens from invoking other state-changing functions, so it is critical that admin accounts **never** call `swap` functions with unverified tokens.

Beyond the `swap` functions, the DLMM core contract includes a number of other functions that use `tx-sender` for authentication, even when there is no technical need to do so. This introduces unnecessary risk, particularly in admin-only functions. If an attacker can trick an admin into interacting with a malicious contract, they may hijack execution and invoke sensitive functions using the admin's identity via `tx-sender`.

Admins are intended to be the sole callers of certain pool-level functions. A pool that includes a malicious token could abuse this trust and perform unauthorized actions on behalf of an admin.

Example Attack:

- A malicious actor deploys a contract-hash-verified pool with one legitimate token and one malicious token.
- The pool operates normally, with real liquidity and regular swaps, to create the appearance of legitimacy.
- Eventually, an admin calls `claim-protocol-fees`. The call `(contract-call? pool-trait pool-transfer x-token-trait unclaimed-x-fees fee-address)` invokes a transfer via the malicious token (`x-token-trait`).
- The malicious token then calls a sensitive function like `add-admin` on the core contract, impersonating the admin, and setting a new malicious admin in the contract.

Impact

The likelihood of this attack is low, but the potential impact is high. Therefore, we have assessed the severity as Medium.

Recommended Mitigation

Avoid using `tx-sender` for authentication unless absolutely necessary. Use `contract-caller` where applicable, especially in admin-only and sensitive functions.

Comments

Bitflow

Acknowledged, but still thinking through the u1 STX transfer idea. Thanks!

Implemented 1 uSTX transfer logic in this commit: 54113b6036b737fa2f9156a9365e0edece7e26e2

[M-2] No upgrade path for core contract

Description

The core and pool contracts are critical components that may evolve in future versions. However, there is currently no mechanism to upgrade either of them.

With the current implementation of the `dlmm-pool-sbtc-usdc-v-1-1` contract, pools are permanently tied to the `dlmm-core-v-1-1` contract. All pool funds can only be managed by the current core contract. As a result, upgrading to a new core version would make it impossible to continue using existing pools.

This becomes especially problematic if the core needs to be upgraded to address a serious vulnerability. In such a case, Bitflow would ideally want the ability to pause the current core and migrate all pools to a new core contract. No such mechanism exists in the current setup.

Similarly, if new versions of pool contracts are released in the future, there should be a clear migration path to transfer all balance mappings, funds, and relevant state to the new version.

Impact

Bitflow currently has no way to upgrade the core or pool contracts. The likelihood of needing to roll out a new version is eventually High, and the impact of not having an upgrade path is Medium. Therefore, we assess the severity of this issue as Medium.

Recommended Mitigation

Consider adding functionality to support upgrading the core contract. To enable this, the core contract should include a function to set a new core address. Each pool contract should also implement a setter function, callable only by the current core address, that updates its stored core address.

Additionally, consider implementing a mechanism to migrate pools to new versions. The pool contract should include a function, callable only by the core address, that allows the migration of funds and relevant state to a new pool contract.

Comments

Bitflow

We're planning to add a new function to update the core address for pools - will keep you updated. Thanks!

Core address migration logic has been implemented in this commit: bf11fc201f07093b8f689dca9248001faeb9f904

Greybeard Security

Reviewing the fixes I would propose the following changes:

1.- Set the initial value of `core-migration-address` the core contract itself:

```
-(define-data-var core-migration-address principal tx-sender)
+(define-data-var core-migration-address principal (as-contract tx-sender))
```

Otherwise an admin could accidentally migrate any pool to the core contract deployer by calling `migrate-core-address` and without any cooldown, because `core-migration-execution-time` is initially 0.

2.- Apply the 1 uSTX transfer protection to `set-core-migration-address`, `set-core-migration-cooldown` and `migrate-core-address`:

```
    ;; Transfer 1 uSTX from caller to BURN_ADDRESS (post condition check)
    (try! (stx-transfer? u1 caller BURN_ADDRESS))
```

Bitflow

Updated 1 in this commit: b949aa694378f0136023e2ad533f1b514f2b4824.

2 was already implemented.

[M-3] After a bin cross an immediate swap in the reverse direction reverts because 0 tokens are required

Description

Whenever a user calls `swap-x/y-for-y/x` and they cross into the next bin — and the bin has liquidity — the `active-bin-id` is decremented/incremented.

In the next transaction, if a user immediately calls `swap-y/x-for-x/y` the swap will revert with error `u3` ("amount is non-positive") from SIP-013.

When the `active-bin-id` is updated the liquidity in the bin is one-sided. i.e. completely composed of token X/Y.

The following code (from the `swap-x-for-y` case) is executed in the `(let ...)` block

1. `x-amount` is calculated. Since `(is-eq y-balance u0)` this will evaluate to `u0`

```
(max-x-amount (/ (+ (* y-balance PRICE_SCALE_BPS) (- bin-price u1)) bin-price))
```

2. Next the `updated-max-x-amount` is calculated. It evaluates to `u0` since `max-x-amount` is 0

```
(updated-max-x-amount (if (> swap-fee-total u0) (/ (* max-x-amount FEE_SCALE_BPS) (- FEE_SCALE_BPS swap-fee-total)) max-x-amount))
```

3. Next the `updated-x-amount` is calculated as `u0` since `(>= x-amount 0)` is always `true`

```
(updated-x-amount (if (>= x-amount updated-max-x-amount) updated-max-x-amount x-amount))
```

Next this block of code is called (since the bin has liqudity)

```
(if (not initial-bin-balances-empty)
    (try! (contract-call? x-token-trait transfer updated-x-amount caller pool-contract none))
    false)
```

This will return `(err u3)` as per SIP-013.

A similar argument holds for the `swap-y-for-x` case

Impact

The impact is that if this edge case is ever reached, it will be necessary for *someone* to perform a small swap in the original direction so that a swap in the opposite direction can be enabled. This may well end up being an admin since users will not know what is wrong. This is a Denial of Service.

Proof of Concept

Add the following file in `tests/dlmm-core-immediate-swap-back.test.ts`

```
import {
  bob,
  deployer,
  setupTokens,
  sbtcUsdcPool,
  mockSbtcToken,
  mockUsdcToken,
  createTestPool,
  dlmmCore,
} from "./helpers-greybeard";

import { describe, it, expect, beforeEach } from 'vitest';
import { txErr, txOk } from '@clarigen/test';

const ERR_FT_AMOUNT_NON_POSITIVE = 3n; // from SIP-010:
↪  https://github.com/stacksgov/sips/blob/main/sips/sip-010/sip-010-fungible-token-standard.md


function addLiquidityToBin(pool: string, binId: bigint, xAmount: bigint, yAmount: bigint, caller: string = deployer) {
    const minDlp = 1n;
    const maxXFees = 10000000n;
    const maxYFees = 10000000n;

    txOk(dlmmCore.addLiquidity(
      pool,
      mockSbtcToken.identifier,
      mockUsdcToken.identifier,
      binId,
      xAmount,
      yAmount,
```

```
      minDlp,
      maxXFees,
      maxYFees
    ), caller);
}

const pool = sbtcUsdcPool.identifier;
const sbtc: string = mockSbtcToken.identifier;
const usdc: string = mockUsdcToken.identifier;

describe('Greybeard Security - DLMM Swap Router Favorability', () => {

   beforeEach(async () => {
        setupTokens();
   })

  it('does fail (but should not) when Bob swaps Y-for-X to next bin and immediately swaps back', async () => {
    createTestPool(pool, 1_0000n, 50_000000n);
    addLiquidityToBin(pool, 1n, 1_0000n, 0n);

    // Bob swaps into bin 1 on sbtcUsdcPool
    txOk(dlmmCore.swapYForX(
        pool,
        sbtc,
        usdc,
        0n,
        10000_000000n
      ), bob);

    // Bob immediately tries to swap back to bin 0
    // This fails because the `updated-x-amount` is calculated as 0
    const r0 = txErr(dlmmCore.swapXForY(
        pool,
        sbtc,
        usdc,
        1n,
        1n
      ), bob);
      expect(r0.value).toBe(ERR_FT_AMOUNT_NON_POSITIVE);
  });

  it('does fail (but should not) when Bob swaps X-for-Y to next bin and immediately swaps back', async () => {
    createTestPool(pool, 1_0000n, 50_000000n);
    addLiquidityToBin(pool, -1n, 0n, 50_0000000n);

    // Bob swaps into bin -1 on sbtcUsdcPool
    txOk(dlmmCore.swapXForY(
        pool,
        sbtc,
        usdc,
        0n,
        10_00000000n
      ), bob);

    // Bob immediately tries to swap back to bin 0
    // This fails because the `updated-x-amount` is calculated as 0
    const r0 = txErr(dlmmCore.swapYForX(
        pool,
        sbtc,
        usdc,
        -1n,
        1n
      ), bob);
      expect(r0.value).toBe(ERR_FT_AMOUNT_NON_POSITIVE);
  });


});
```

Recommended Mitigation

For `swap-x-for-y` apply the following diff.

```
      ;; Transfer updated-x-amount x tokens from caller to pool-contract
-     (if (not initial-bin-balances-empty)
+     (if (and (not initial-bin-balances-empty) (> updated-x-amount u0))
          (try! (contract-call? x-token-trait transfer updated-x-amount caller pool-contract none))
          false)

      ;; Transfer dy y tokens from pool-contract to caller
-     (if (not initial-bin-balances-empty)
+     (if (and (not initial-bin-balances-empty) (> dy u0))
          (try! (contract-call? pool-trait pool-transfer y-token-trait dy caller))
          false)
```

Similarly for `swap-y-for-x`

Comments

Bitflow

Fixing, thanks! I also think we should implement these checks at lines 1138 and 1283:

- Line 1138: `(if (and (> updated-x-amount u0) (not initial-bin-balances-empty))`
- Line 1283: `(if (and (> updated-y-amount u0) (not initial-bin-balances-empty))`

Thoughts?

Fixed in this commit: 7ee846161704d8cd6a9f69578115e8e311030c46

Greybeard Security

These extra additions seem safe. We'll run through the logic just for documentation's sake:

In `swap-x-for-y` if `(is-eq update-x-amount u0)` then obviously no token Y is returned, and we don't need need to update the bin balances. Similarly for `swap-y-for-x`

It's a nice addition and saves some gas in that case

[M-4] With enough funds a user can pay tiny fees for swaps

Description

One of the features of Liquidity Book based CLMMs is a fee known as the *composition fee*. This prevents a user from adding and immediately withdrawing liquidity in order to perform a *fee-free swap*.

A flaw in the logic of `add-liquidity` means that a user is given too many bin shares when calling `add-liquidity`. Consequently, as the amount of capital available to them increases, they are able to perform a swap that approaches being fee-free.

This issue was first discovered by Offside Labs and thoroughly explain in this blog post.

Impact

A user can perform a swap in a bin, at the bin's price, for a very low fee. The impact is limited to the protocol and other LPs losing fee revenue.

Recommended Mitigation

Add the composition fee into the denominator of the share calculation.

Proof of Concept

Add a new file: `tests/dlmm-core-fee-free-swap.test.ts`. This test

```
import {
  alice,
  dlmmCore,
  sbtcUsdcPool,
  pool,
  sbtc,
  usdc,
  mockSbtcToken,
  mockUsdcToken,
  addLiquidityToBin,
  toBinId,
  setupTokens,
  createTestPool,
} from "./helpers-greybeard";

import { describe, it, expect, beforeEach } from 'vitest';
import {
  cvToValue,
} from '@clarigen/core';
import { txOk, rovOk } from '@clarigen/test';


describe('DLMM Core Fee free swap', () => {

  beforeEach(async () => {
    setupTokens();
    createTestPool(pool, 10_0000n, 100_000000n);
    const poolData = rovOk(sbtcUsdcPool.getPool());
    expect(poolData.poolCreated).toBe(true);
    expect(poolData.binStep).toBe(25n);
    expect(poolData.activeBinId).toBe(0n);
    addLiquidityToBin(pool, 0n, 500_0000n, 5000_000000n);
  });

  it('Perform low-fee swap but only with massive capital', async () => {
    console.log("--- getPoolForAdd ---");
    const FEE_PERCENTAGE_DECIMALS = 3;
    const r0 = rovOk(sbtcUsdcPool.getPoolForAdd());

    const binId   = 0n; // Active bin
    const minDlp  = 1n;
    const minXAmount = 1n;
    const minYAmount = 1n;

    const price = rovOk(dlmmCore.getBinPrice({ initialPrice: r0.initialPrice, binId: r0.activeBinId, binStep: r0.binStep}));
    console.log("price: ", Number(price) / 1e6);
    const binBalances0 = rovOk(sbtcUsdcPool.getBinBalances(toBinId(binId)));
    console.log("binBalances0", binBalances0);
    console.log("price for swap", Number(binBalances0.yBalance) / Number(binBalances0.xBalance)*100);

    console.log("--- Attack begins ----");

    const initialX = rovOk(mockSbtcToken.getBalance(alice));
    const initialY = rovOk(mockUsdcToken.getBalance(alice));


    const X_AMOUNT = 0;
    const Y_AMOUNT = 50_000_000_000000n;

    const response2 = txOk(dlmmCore.addLiquidity(
      pool,
      sbtc,
      usdc,
      binId,
      X_AMOUNT,
      Y_AMOUNT,
      minDlp,
      2n**127n, // effectively no limit on X fees
      2n**127n  // effectively no limit on Y fees
    ), alice);

    const binBalances1 = rovOk(sbtcUsdcPool.getBinBalances(500n));
    console.log("binBalances1", binBalances1);
    console.log("new price", Number(binBalances1.yBalance) / Number(binBalances1.xBalance)*100);

    const liqReceived2 = cvToValue(response2.result);
    console.log("liqReceived2: ", liqReceived2);
```

```
    txOk(dlmmCore.withdrawLiquidity(
      sbtcUsdcPool.identifier,
      mockSbtcToken.identifier,
      mockUsdcToken.identifier,
      binId,
      liqReceived2,
      minXAmount,
      minYAmount
    ), alice);

    const binBalances2 = rovOk(sbtcUsdcPool.getBinBalances(500n));
    console.log(binBalances2);
    console.log(Number(binBalances2.yBalance) / Number(binBalances2.xBalance)*100);


    const finalX = rovOk(mockSbtcToken.getBalance(alice));
    const finalY = rovOk(mockUsdcToken.getBalance(alice));
    console.log({ finalX, finalY });

    const sign = finalX < initialX ? -1n : 1n;
    const diffX = finalX - initialX;
    const diffY = finalY - initialY;

    const xVal = Number(diffX * price / (10n**8n)) / 1e6;
    const yVal = Number(diffY) / 1e6;

    console.log("X       :", Number(diffX) / 1e8);
    console.log("X (in Y):", xVal);
    console.log("Y       :", yVal);

    const xStr = `${Number(sign * diffX) / 1e8} Xs (valued at ${Number(sign) * xVal} Ys)`;
    const yStr = `${Number(-sign) * yVal} Ys`;

    console.log(`Effectively swapped ${sign == 1n ? yStr : xStr} for ${sign == 1n ? xStr : yStr}`);
    const fee = -(xVal + yVal);
    console.log("Fee", -(xVal + yVal));

    const feePercentage = Math.round(fee / (sign == 1n ? xVal : yVal) * 100 * 10**FEE_PERCENTAGE_DECIMALS)/10**FEE_PERCENTAGE_DECIMALS; //
    ↪  rounded to 3 decimals
    console.log(`Fee percentage: ${feePercentage}%`);
    // Remember, this is a percentage (not a straight ratio)
    expect(feePercentage).lessThan(0.01);

  });
});
```

Comments

Bitflow

The test is now returning "Fee: 20.263021000000663; Fee percentage: 0.4%"

Fixed, thanks! Here's the commit: ffe6d1f8b0018226b11032246af1301ba7bf37ac

Looking back at the issue, shouldn't we also include this updated logic in `move-liquidity`? Is there a reason it wasn't included in the finding?

Greybeard Security

Great catch. We did not consider this. Since `move-liquidity` is essentially `withdraw-liquidity` followed by `add-liquidity` it will also require the fix.

Bitflow

Fixed in commit: 46f4a99e5d6720672fbe926b38b1a2a5da0d9a0f

---

8 Low Findings

[L-1] Bin steps can be added with wrong values and they cannot be updated/removed

Description

If an admin adds a new bin step along with its factors, and these happen to have incorrect values, neither the factors can be modified nor the bin step removed entirely. At the very least, the factors should be editable in case of an emergency.

Impact

The severity is Low, as the likelihood of an error when creating a new bin step is very low. Only admins can call the function that adds a new bin step, and Bitflow calculates the factors off-chain using the same method applied to the initial five steps.

Although the potential impact could be High if an error were to occur, the low likelihood justifies a Low severity rating.

Recommended Mitigation

Add a new function that allows an admin to update the factors of a bin-step.

Comments

Bitflow

Acknowledged. We're planning to add core upgradability. If needed, we can deploy a new core contract with corrected bin factors. Thanks!

---

9 Informational Findings

[I-1] Errors in `transfer` function do not follow `SIP-013`

Description

As defined in SIP-013, specific error codes should be returned for certain events.

Error Code	Description
u1	The sender has insufficient balance.
u2	The sender and recipient are the same principal.
u3	Amount is `u0`.
u4	The sender is not authorized to transfer tokens.

However, the error codes returned by the `SIP-013` transfer function in the `dlmm-pool-sbtc-usdc-v-1-1` contract do not align with the table above in most cases.

For example, when the sender and recipient are the same principal, the function returns `ERR_INVALID_PRINCIPAL_SIP_013` (`u5`) instead of the expected `u2`.

Impact

The discrepancy outlined in the description could lead to issues with third-party integrations, as the actual errors returned may not align with those expected. Since this is not considered a security vulnerability, we have assessed the issue as Informational.

Recommended Mitigation

Align the error codes in the function with the `SIP-013` specification.

Comments

Bitflow

Fixed, thanks! Commit: 9cc772c26116941d04fa1e174b6983271bdf24b2

[I-2] Multi-bin swaps using favourable bins can actually lead to worse outcomes

Description

When using `dlmm-swap-router` for a multi-step swap a user is able to specify a `max-unfavorable-bins` parameter which allows them them to set a cap on the number of bins for which the price is unfavourable to them.

The design of the router implicitly assumes that swap steps in favourable bins will always be good for the user. However, this is not always the case.

Consider a scenario where a user wants to do a two step swap X-for-Y in `pool1` and `pool2` using `swap-x-for-y-same-multi`. They specify `bin-id == 1` followed by `bin-id == 0` with `max-unfavorable-bins == 0`. Also:

- the last time they checked the current `active-bin-id == 1` for `pool1`
- another user front-runs them, with a swap of Y-for-X, pushing `active-bin-id` to `2`

We now consider the result of the swap for the current state of the pool bins vs what the user expected.

The expected state of the pools is:

- `pool1`. `active-bin-id == 1`. `y-balance == 7,000 Y`. Price is `1.01` Y/X
- `pool2`. `active-bin-id == 0`. `y-balance == 10,000 Y`. Price is `1.00` Y/X

The actual state of the pools is:

- `pool1`. `active-bin-id == 2`. `y-balance == 3000 Y`. Price is `1.02` Y/X (favourable!)
- `pool2` is as before

Expected case:

- swap 1: `bin-id == 1` get `7,000 Y` for `6930.69 X`
- swap 2 `bin-id == 0` get `3069.31 Y` for remaining `10,000 - 6,930.69 = 3069.31 X`
- Total out: `10,069.31 Y`

Actual case:

- swap 1: `bin-id == 2`. Get `3,000 Y` for `2,941.18 X`
- swap 2. `bin-id == 0` get `7048.82 Y` for remaining `10,000 - 2941.18 == 7048.82 X`
- Total out: `9990 Y`

Thus, the user has actually done *worse* even though the bins were supposedly favourable.

The root cause is the design. There is more to favourability that price alone. The capacity of the bins must also be taken into account.

Impact

A user can actually do worse when favourable bins are available because the capacity of those bins is not as high as they expected. The result is unexpected funds loss for the user.

However, the user can prevent this by setting `min-received` correctly for each swap step. Thus, this has been assessed as **Informational**.

Recommended Mitigation

The user should be able to specify stricter favourability guarantees on the swaps performed, and on a per-bin basis, not just as a total. One suggestion is to have a `favorability-delta` on each swap step.

Proof of Concept

Add `dlmm-swap-router-favorability.test.ts`.

```
import {
  alice,
  bob,
  toBinId,
  dlmmSwapRouter,
  setupTokens,
  pool1,
  pool2,
  pool3,
  pool4,
  sbtc,
  usdc,
  sbtcUsdcPool1,
  sbtcUsdcPool2,
  sbtcUsdcPool3,
  mockSbtcToken,
  mockUsdcToken,
  createTestPool,
  dlmmCore,
  addLiquidityToBin
} from "./helpers-greybeard";

import { describe, it, expect, beforeEach } from 'vitest';
import { txOk, rovOk } from '@clarigen/test';


describe('Greybeard Security - DLMM Swap Router Favorability', () => {

   beforeEach(async () => {
        setupTokens();
   })

  it('should produce worse result for favorable bins', async () => {
    createTestPool(pool1, 1_0000n, 50_000000n);
    createTestPool(pool2, 1_0000n, 50_000000n);

    createTestPool(pool3, 1_0000n, 50_000000n);
    createTestPool(pool4, 1_0000n, 50_000000n);
```

```
addLiquidityToBin(pool1, 1n, 1_0000n, 0n);
addLiquidityToBin(pool1, 2n, 1_0000n, 0n);


addLiquidityToBin(pool3, 1n, 1_0000n, 0n);
addLiquidityToBin(pool3, 2n, 1_0000n, 0n);

// Pool 2 and pool 4 are set up just the same way
addLiquidityToBin(pool2, 0n, 5000n, 2_500000n);
addLiquidityToBin(pool4, 0n, 5000n, 2_500000n);


// Bob swaps into bin 1 on sbtcUsdcPool1
txOk(dlmmCore.swapYForX( pool1, sbtc, usdc, 0n, 100_000000n ), bob);
// Bob swaps into bin 2 on sbtcUsdcPool1
txOk(dlmmCore.swapYForX(pool1, sbtc, usdc, 1n, 1_00000000n), bob);
// Bob swaps a little further in bin 2
txOk(dlmmCore.swapYForX(pool1, sbtc, usdc, 2n, 1_100000n), bob);

const pool1Bin1Bals = rovOk(sbtcUsdcPool1.getBinBalances({ id: toBinId(1n) }));
const pool1Bin2Bals = rovOk(sbtcUsdcPool1.getBinBalances({ id: toBinId(2n) }));


expect(rovOk(sbtcUsdcPool1.getActiveBinId())).toBe(2n);
expect(rovOk(sbtcUsdcPool2.getActiveBinId())).toBe(0n);


// Bob swaps into bin 1 on sbtcUsdcPool3
txOk(dlmmCore.swapYForX( pool3, sbtc, usdc, 0n, 100_000000n ), bob);
// Bob swaps a little further in bin 1
txOk(dlmmCore.swapYForX( pool3, sbtc, usdc, 1n, 77_500000n ), bob);

const pool3Bin1Bals = rovOk(sbtcUsdcPool3.getBinBalances({ id: toBinId(1n) }));
const pool3Bin2Bals = rovOk(sbtcUsdcPool3.getBinBalances({ id: toBinId(2n) }));
console.log("\n\n-----------------------------------------------");
console.log("Pool 1's bin 2 Y balance < Pool 3' bin 1 Y balance");
console.log("pool1 @ bin 1", pool1Bin1Bals);
console.log("pool1 @ bin 2", pool1Bin2Bals);
console.log("pool3 @ bin 1", pool3Bin1Bals);
console.log("pool3 @ bin 2", pool3Bin2Bals);
console.log("-----------------------------------------------");

expect(Number(pool3Bin1Bals.yBalance)).greaterThan(Number(pool1Bin2Bals.yBalance))

expect(rovOk(sbtcUsdcPool1.getActiveBinId())).toBe(2n);
expect(rovOk(sbtcUsdcPool2.getActiveBinId())).toBe(0n);


const xBefore0 = rovOk(mockSbtcToken.getBalance({ who: alice}));
const yBefore0 = rovOk(mockUsdcToken.getBalance({ who: alice}));

txOk(dlmmSwapRouter.swapXForYSameMulti(
    { swaps: [
        { poolTrait: pool1,
          expectedBinId: 1n,
          minReceived: 1n,
        },
        { poolTrait: pool2,
          expectedBinId: 0n,
          minReceived: 1n,
        },
      ],
      xTokenTrait: sbtc,
      yTokenTrait: usdc,
      amount: 1_5000n,
      minYAmountTotal: 1n,
      maxUnfavorableBins: 0n
    }), alice);

const xAfter0 = rovOk(mockSbtcToken.getBalance({ who: alice}));
const yAfter0 = rovOk(mockUsdcToken.getBalance({ who: alice}));


const xBefore1 = rovOk(mockSbtcToken.getBalance({ who: alice}));
const yBefore1 = rovOk(mockUsdcToken.getBalance({ who: alice}));

txOk(dlmmSwapRouter.swapXForYSameMulti(
    { swaps: [
        { poolTrait: pool3,
          expectedBinId: 1n,
          minReceived: 1n,
        },
        { poolTrait: pool4,
          expectedBinId: 0n,
          minReceived: 1n,
        },
```

```
          ],
          xTokenTrait: sbtc,
          yTokenTrait: usdc,
          amount: 1_5000n,
          minYAmountTotal: 1n,
          maxUnfavorableBins: 0n
        }), alice);

    const xAfter1 = rovOk(mockSbtcToken.getBalance({ who: alice}));
    const yAfter1 = rovOk(mockUsdcToken.getBalance({ who: alice}));

    const deltaX0 = xBefore0 - xAfter0;
    const deltaX1 = xBefore1 - xAfter1;
    const deltaY0 = yAfter0 - yBefore0;
    const deltaY1 = yAfter1 - yBefore1;

    console.log("\n\n-----------------------------------------------");
    console.log("Favorable bin in pool1/expected bin in pool2")
    console.log("X0: ", deltaX0);
    console.log("Y0: ", deltaY0);

    console.log("\n\nExpected bin in pool3/expected bin in pool4");
    console.log("X1: ", deltaX1);
    console.log("Y1: ", deltaY1);

    console.log("You would expect first swap to yield more, but it doesn't");
    expect(deltaX0).toBe(deltaX1);
    expect(Number(deltaY1)).greaterThan(Number(deltaY0));


  });
});
```

Comments

Bitflow

Acknowledged. The unfavorable bin logic only compares the active bin ID to the expected bin ID, not bin balances or other factors. We calculate min-received per swap off-chain and enforce it on-chain.

[I-3] Verified pools can still use malicious tokens

Description

In contract `dlmm-core-v-1-1` the `verified-pool-code-hashes` data variable is used to show whether a pool contract has been verified or not. A similar data variable should be added for `verified-tokens`

Post conditions can prevent many impacts but the one thing they can't prevent is arbitrary smart contract state changes made possible by the authority granted by `tx-sender` based authentication. Anything the user is authorized to do can now be done by the malicious token's logic.

Impact

Users are susceptible to phishing attacks for `tx-sender` based authentication if they interact with malicious tokens. Any tokens they hold are potentially at risk.

Recommended Mitigation

Add a `verified-token` data variable and an accompanying read-only function to see if a token is verified or not.

Comments

Bitflow

Acknowledged. We're planning to handle token verification in a peripheral contract and also off-chain.

---

10 Appendix

All tests used the following `tests/helpers-greybeard.ts` file

```ts
import { project, accounts } from './clarigen-types';
import {
  cvToValue,
  projectErrors,
  projectFactory,
  CoreNodeEventType
} from '@clarigen/core';
import { rovOk, txOk, filterEvents } from '@clarigen/test';

export const contracts = projectFactory(project, "simnet");

export const deployer = accounts.deployer.address;
export const alice = accounts.wallet_1.address;
export const bob = accounts.wallet_2.address;
export const charlie = accounts.wallet_3.address;

export const dlmmCore = contracts.dlmmCoreV11;
export const dlmmSwapRouter = contracts.dlmmSwapRouterV11;
export const dlmmLiquidityRouter = contracts.dlmmLiquidityRouterV11;
export const sbtcUsdcPool = contracts.dlmmPoolSbtcUsdcV11;
export const sbtcUsdcPool1 = contracts.dlmmPoolSbtcUsdc1stV11;
export const sbtcUsdcPool2 = contracts.dlmmPoolSbtcUsdc2ndV11;
export const sbtcUsdcPool3 = contracts.dlmmPoolSbtcUsdc3rdV11;
export const sbtcUsdcPool4 = contracts.dlmmPoolSbtcUsdc4thV11;
export const mockSbtcToken = contracts.mockSbtcToken;
export const mockUsdcToken = contracts.mockUsdcToken;
export const mockPool = contracts.mockPool;
export const mockRandomToken = contracts.mockRandomToken;

export const pool = sbtcUsdcPool.identifier;
export const pool1 = sbtcUsdcPool1.identifier;
export const pool2 = sbtcUsdcPool2.identifier;
export const pool3 = sbtcUsdcPool3.identifier;
export const pool4 = sbtcUsdcPool4.identifier;
export const sbtc: string = mockSbtcToken.identifier;
export const usdc: string = mockUsdcToken.identifier;

export function toBinId(binIndex: bigint) {
  return binIndex + 500n;
}

const _errors = projectErrors(project);

export const errors = {
  dlmmCore: _errors.dlmmCoreV11,
  sbtcUsdcPool: _errors.dlmmPoolSbtcUsdcV11,
  dlmmSwapRouter: _errors.dlmmSwapRouterV11,
  dlmmLiquidityRouter: _errors.dlmmLiquidityRouterV11,
  sbtc: _errors.mockSbtcToken,
  usdc: _errors.mockUsdcToken,
};

export function getPrintEvents(response: any) {
  return filterEvents(
    response.events,
    CoreNodeEventType.ContractEvent
  );
}

export function getEventsDataByAction(action: string, response: any) {
  return getPrintEvents(response)
    .map(printEvent => cvToValue(printEvent.data.value))
    .filter(parsedEvent => parsedEvent.action === action);
}

export function getSwapXForYEventData(response: any) {
  return getEventsDataByAction("swap-x-for-y", response);
}

export function getSwapYForXEventData(response: any) {
  return getEventsDataByAction("swap-y-for-x", response);
}

export function getAddLiquidityEventData(response: any) {
  return getEventsDataByAction("add-liquidity", response);
}

export function getWithdrawLiquidityEventData(response: any) {
  return getEventsDataByAction("withdraw-liquidity", response);
}
```

```
// Common pool setup functionality
export function setupTokens() {
  // Step 1: Mint tokens to required parties
  txOk(mockSbtcToken.mint(10_00000000n, deployer), deployer);  // 10 BTC to deployer
  txOk(mockUsdcToken.mint(50_0000_000000n, deployer), deployer); // 500k USDC to deployer
  txOk(mockSbtcToken.mint(1000_00000000n, alice), deployer);  // 1000 BTC to alice
  txOk(mockUsdcToken.mint(50_000_000_000000n, alice), deployer); // 50M USDC to alice
  txOk(mockSbtcToken.mint(100_00000000n, bob), deployer);  // 100 BTC to bob
  txOk(mockUsdcToken.mint(5_000_000_000000n, bob), deployer); // 5M USDC to bob
}

export function createTestPool(poolIdentifier: string, btcAmount: bigint, usdcAmount: bigint) {

  // Create pool with proper parameters
  txOk(dlmmCore.createPool(
    poolIdentifier,
    mockSbtcToken.identifier,
    mockUsdcToken.identifier,
    btcAmount,    // 0.1 BTC in active bin
    usdcAmount,   // 5000 USDC in active bin
    1000n,        // burn amount
    10n, 30n,     // x fees (0.1% protocol, 0.3% provider)
    10n, 30n,     // y fees (0.1% protocol, 0.3% provider)
    25n,          // bin step (25 basis points)
    900n,         // variable fees cooldown
    false,        // freeze variable fees manager
    new Uint8Array(0),     // dynamic config
    deployer,     // fee address
    "https://bitflow.finance/dlmm", // uri
    true          // status
  ), deployer);
}

export function generateBinFactors(numEntries: number = Number(dlmmCore.constants.NUM_OF_BINS), startValue: bigint = 1000000n): bigint[] {
  return Array.from({ length: numEntries }, (_, i) => startValue + BigInt(i));
}

export function addLiquidityToBin(pool: string, binId: bigint, xAmount: bigint, yAmount: bigint, caller: string = deployer) {
    const minDlp = 1n;
    const maxXFees = 10000000n;
    const maxYFees = 10000000n;

    txOk(dlmmCore.addLiquidity(
      pool,
      mockSbtcToken.identifier,
      mockUsdcToken.identifier,
      binId,
      xAmount,
      yAmount,
      minDlp,
      maxXFees,
      maxYFees
    ), caller);
}
```