IMMUNEFI AUDIT

Immunefi / Microchain™

DATE September 17, 2025

AUDITOR Zealynx: CODESPECT, Braniac

REPORT BY Immunefi

ABOUT IMMUNEFI

Immunefi is the leading onchain security platform, having directly prevented hacks worth more than $25 billion USD. Immunefi security researchers have earned over $120M USD for responsibly disclosing over 4,000 web2 and web3 vulnerabilities, more than the rest of the industry combined.

Through Magnus, Immunefi delivers a comprehensive suite of best-in-class security services through a single command center to more than 300 projects — including Sky (formerly MakerDAO), Optimism, Polygon, GMX, Reserve, Chainlink, TheGraph, Gnosis Chain, Lido, LayerZero, Arbitrum, StarkNet, EigenLayer, AAVE, ZKsync, Morpho, Ethena, USDT0, Stacks, Babylon, Fuel, Sei, Scroll, XION, Wormhole, Firedancer, Jito, Pyth, Eclipse, PancakeSwap and many more.

Magnus unifies SecOps across the entire onchain lifecycle, combining Immunefi's market leading products and community of elite security researchers with a curated set of the very best security products and technologies provided by top security firms — including Runtime Verification, Dedaub, Fuzzland, Nexus Mutual, Failsafe, OtterSec and others.

Magnus is powered by Immunefi's proprietary vulnerabilities dataset — the largest and most comprehensive in web3, ensuring that security leaders and teams have the best possible tools for identifying and mitigating life threats before they cause catastrophic harm, all while reducing operational overhead and complexity.

Learn how you can benefit too at immunefi.com.

TERMINOLOGY

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

LIKELIHOOD	IMPACT		
	HIGH	MEDIUM	LOW
CRITICAL	Critical	Critical	High
HIGH	High	High	Medium
MEDIUM	Medium	Medium	Low
LOW	Low		
NONE	None		

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

EXECUTIVE SUMMARY

Over the course of 15 days in total, Microchain DEX engaged with Immunefi to review the Mira Binned Liquidity functionality. In this period of time a total of 10 issues were identified.

SUMMARY

Name	Microchain DEX
Repository	https://github.com/mira-amm/mira-binned-liquidity
Audit Commit	8800f00c7a8f56544a8e9aac55d187ab9b36e21b
Type of Project	DeFi, Exchange
Audit Timeline	Aug 18th - Sep 5th
Fix Period	Sep 6th - Sep 11th*f*

ISSUES FOUND

Severity	Count	Fixed	Acknowledged
Critical	0	0	0
High	2	2	0
Medium	4	3	1
Low	3	3	0
INSIGHT	1	1	0

CATEGORY BREAKDOWN

Bug	9
Gas Optimization	0
Informational	1

FINDINGS

IMM-HIGH-01

Putting the LP token as an LP to a pool makes it unredeemable #1

Id	IMM-HIGH-01
Severity	High
Category	Bug
Status	Fixed in 65509ec685e322498357504cf75967b06f69314d

Description

We found one issue within the guard against burning the LP token when it is part of the pool/contract reserves here:
https://github.com/mira-amm/mira-binned-liquidity/blob/1e92e243fa41d0ca41a514e742d9a3acfe2e18d1/libraries/burn_utils/src/lib.sw#L33

This is indeed needed, because of the Fuel UTXO model - when burning the token, it needs to be owned by the contract. The token could either be owned by the contract in case it was transferred as part of the LP burning call, or it is held as LP reserves.

The problem lies in how it is checked, because the condition will still prevent burning even if the LP token was once in a pool but was withdrawn already. So the LP token pool is already empty, but the check will prevent burning:

```TypeScript
    require(
        storage_keys
            .contract_reserves
            .get(asset_id)
            .try_read()
            .is_none(),
        PoolCurveStateError::InvalidLPTokenBalance,
```

```
    );
```

The point is that if someone creates a pool using the LP token as an asset and then withdraws that LP token from that pool, this condition will still be reverting, as it will not be None anymore, despite the fact that the pool created is empty already.

The impact is High as a malicious user could "taint" the LP token in this way and e.g. deposit it on some 3rd party system, or sell it, while because of its unredeemability it will be worthless.

Recommendation

Change the above `require` statement so that when the `contract_reserves` return something else than `None` it is zero.

IMM-HIGH-02

Incorrect reserves accounting leads to LP loss during minting or swapping #2

Id	IMM-HIGH-02
Severity	High
Category	Bug
Status	Fixed in `3fe0690c72963b680df064e94ced9ad2daad728c`

Description

After swap, when we update the pool reserves and contract reserves, we subtract the protocol fees from the pool reserves (which is ok - the protocol fees should be separated from the liquidity pool). We don't however subtract the protocol fees from contract reserves (which is also ok, because you need the contract reserves to reflect the actual balance of the contract - it is used e.g. when LP is minted and the diff between actual balance and reserves is used to calculate how much is added).

In short:

- the pool reserves don't contain protocol fees.
- the contract reserves contain protocol fees.

When the privileged account collects fees (by calling the `collect_protocol_fees(...)` function), we don't deduct them from contract reserves. This means that the accounting of contract reserves is lower than the actual amount. When a new LP comes in, sends a token in, that difference will be deducted from the amount he loads in as LP, so he will be at a loss. This will also lead to loss during swapping as the `token_in` amount is calculated in the same way.

Impact is High, as it might amount to high losses for the fresh LP provider and swapper who engage with the contract directly after the fee collection event. The bigger the fee collection amount is, the bigger is the loss of the next interacting user. Additionally if the LP or swapper tries to add a smaller amount than the `balance - contract_reserves` the function will revert with an underflow.

Recommendation

It is recommended to subtract the amounts of collected fees from the `contract_reserves` within the `collect_protocol_fees(...)` function.

IMM-MED-01

Minting To The Active Bin Returns Reduced/Unfair Shares For LPs #5

Id	IMM-MED-01
Severity	Medium
Category	Bug
Status	Acknowledged

Description

When minting liquidity for an active bin, the shares minted for the liquidity provider are overpriced, further disincentivizing liquidity provision to the active bins and causing unfair share distribution for the providers.

TypeScript

```
/// Handle composition fees for active bin
#[storage(read, write)]
fn handle_composition_fees(
    storage_keys: MintStorageKeys,
    internal_bin_id: InternalBinId,
    bin_reserves: Amounts,
    amounts_in: Amounts,
    fees: Amounts,
    price: Price,
    total_shares_in_bin: u256,
) -> (u256, Amounts) {
    let amounts_after_fees = amounts_in.sub(fees);
    let user_liquidity = get_liquidity(amounts_after_fees, price);
   // ... code snippet ..

    let non_protocol_fees = fees.sub(protocol_fees);
    let bin_reserves_with_fees = bin_reserves.add(non_protocol_fees);
    let bin_liquidity = get_liquidity(bin_reserves_with_fees, price);

    let shares = math::uint256x256_math::mul_div_round_down(user_liquidity,
total_shares_in_bin, bin_liquidity);
    (shares, amounts_in)
}
```

Here, `bin_reserves` is topped up with the non-protocol fees, and the calculation for the shares uses this overestimated liquidity, not taking into account that total fees have already been deducted from the user.

While the code for calculating user obtained shares diminishes the LP value, the code follows strictly LFJ V2 formula which has been accepted by the market for a long time already. That's why the chosen severity for it is Medium.

Recommendation

Use the normal `bin_reserves` for the calculation of the shares the user gets when providing liquidity for an active bin.

IMM-MED-02

Lack of maximum liquidity guardrail during swapping #7

Id	IMM-MED-02
Severity	Medium
Category	Bug
Status	Fixed in ef061b3c637fc5ba3bee8b4cf11d2fe69cca9397

Description

The contract contains a guardrail against crossing the maximum allowed liquidity amount within a single bin during LP minting process within the `bin_helper::get_shares_and_effective_amounts_in(...)` function:

```
TypeScript
    require(
        final_liquidity <= MAX_LIQUIDITY_PER_BIN,
        PoolCurveStateError::MaxLiquidityPerBinExceeded,
    );
```

In practice the bin liquidity is also increased during swapping as swap fees are accrued and are added to the overall `bin_reserves`, this is done in the `swap_utils::process_single_bin(...)`:

```
TypeScript
      // Update bin reserves
    let amounts_in_after_fees = amounts_in_with_fees - protocol_fees;
    let new_bin_reserves = calculate_new_bin_reserves(
        bin_reserves,
        amounts_in_after_fees,
        amounts_out_of_bin,
        args.swap_for_y,
    );

    storage_keys.bins.insert(internal_bin_id, new_bin_reserves);
```

Yes, there is no verification here that the `new_bin_reserves` don't cross the `MAX_LIQUIDITY_PER_BIN` threshold.

The impact is that when liquidity crosses the threshold it might result in overflow while doing the `u256` math hence, which in turn would break the swapping, minting and burning functionality on the given bin. This in turn would result in funds locked. The chances of this however are low, due to the high amount of tokens that would need to be deposited as liquidity. While the impact is serious, the conditions where it could happen are difficult to achieve in practice, hence the severity is classified as Medium.

Recommendation

Validate the new bin reserves against crossing the `MAX_LIQUIDITY_PER_BIN` threshold during swaps.

IMM-MED-03

Inconsistency In Functionality & Spec Will Lead To DOS, And Possibly Theft of Funds. #10

Id	IMM-MED-03
Severity	Medium
Category	Bug
Status	Fixed in 96aa6677a491063d37c81e5fbce252aa97f16873

Description

On the call to `burn_liquidity()`, it is stated in the comment spec that multiple LP tokens from different pools can be burned on a single call.

```
TypeScript
   /// * LP tokens for multiple pools can be mixed in the same call.
    #[storage(read, write)]
    fn burn_liquidity(args: BurnLiquidityArgs) -> Vec<Asset> {
```

However, this is not possible where the `required_pool_id` is cached before the loop to burn each LP token in the array subsequently.

```
TypeScript
      //@audit
      let required_pool_id = get_pool_id_from_lp_token(storage_keys,
args.lp_assets.get(0).unwrap());

        let mut reserves_withdrawn: Vec<Asset> = Vec::new();

        for lp_asset in args.lp_assets.iter() {
            let (asset_1, asset_2) = burn_lp_token(storage_keys, lp_asset, args.to,
required_pool_id);

            transfer_reserves(asset_1, asset_2, args.to);
```

```
            if asset_1.amount > 0 {
                reserves_withdrawn.push(asset_1);
            }
            if asset_2.amount > 0 {
                reserves_withdrawn.push(asset_2);
            }
        }
```

Since the `required_pool_id` is locked to the initial LP token's pool to be burned, such that burning an LP token from a different pool will revert on the validations in `verify_lp_token()`

TypeScript

```
/// Verifies that the LP token is valid
#[storage(read)]
fn verify_lp_token(storage_keys: BurnStorageKeys, asset_id: AssetId, required_pool_id:
PoolId) {
    //.. snip ..
    require(pool_id.unwrap() == required_pool_id,
PoolCurveStateError::LPTokenFromWrongPool);//@audit
}
```

This inconsistency between the actual functionality and the commented spec can lead to harmful scenarios for protocols or users trying to integrate with Mira binned liquidity.

For instance,

1. A user sends LP tokens from different pools to the curve state prior to the `burn_liquidity()` call(As this is a required)
2. Trying to call `burn_liquidity()` now reverts due to the actual functionality in the code.
3. Before the user can place a certain number of transactions to burn the share of the different LP tokens they sent to the contract.
4. An attacker can frontrun them to burn those shares for themselves.

Recommendation

Correct the spec and let users know how to properly integrate with the curve state, as every transaction to burn should be bundled in one call.

IMM-MED-04

Dusts/Leftover Reserves In Bins Will Be Forced To Stay Idle #11

Id	IMM-MED-04
Severity	Medium
Category	Bug
Status	Fixed in 96aa6677a491063d37c81e5fbce252aa97f16873

Description

Bins are removed from the tree(which is essential for swaps) when the total shares left for that bin are zero; however, this condition doesn't assure the bin's reserve in turn will be empty, as the shares burned from users are rounded down, allowing for possible leftover reserves in such a bin while having zero total shares.

```TypeScript
        if (new_total_shares == 0) {
            remove_bin_from_tree(storage_keys, internal_bin_id);
        }
```

This will cause some reserves to remain unused when swapping, leading to idle funds left in the curve state

Recommendation

Ensure the shares burned align with the actual reserve returned to the user, or turn to protocol fees.

IMM-LOW-01

Incorrect SRC20 standard implementation #3

Id	IMM-LOW-01
Severity	LOW
Category	Bug
Status	Fixed in 584e378537207d8120da39e746e1da0293da5528

Description

The SRC20 token implementation in the contract for the LP tokens is not following the official Fuel SRC20 standard document.

The Mira AMM creates a new `AssetId` for each LP position that it creates in the `mint_liquidity(...)` method. Each token created in this way needs to have a Symbol, Name and Decimal values assigned and those values must be returned by specific methods.

The contract doesn't fill the `symbol`, `name`, `decimals` storage variables when a new LP token is created. This doesn't entail any security vulnerability, but as per the standard of SRC20:

```
TypeScript
fn name(asset: AssetId) -> Option<String>
```

This function MUST return the name of the asset, such as "Ether". This function MUST return Some for any assets minted by the contract.

In the case of the `name(...)` method the implementation would call the default std-lib function `_name(...)` which will return None.

The current behaviour might become problematic when the tokens are integrated in some 3rd party protocol - not following the standard, might require non-standard handling on their side, so there is a risk of becoming difficult-to-integrate-to.

Recommendation

Assign correct values to `symbol`, `name`, `decimals` storage variables and implement specified event generation when those values change. The full specification is available here: https://docs.fuel.network/docs/sway-standards/src-20-native-asset/

IMM-LOW-02

Pool reserves accounting incorrectly includes composition protocol fees #6

Id	IMM-LOW-02
Severity	LOW
Category	Bug
Status	Fixed in e3e597ada1ef043faf30cf18e75b176a5593891a

Description

During LP minting a user can provide both Token X and Token Y only to the active bin. The active bin has a composition factor which describes the relation between both token reserves. If during LP minting the amounts of Token X and Token Y are provided in a way that alters the composition factor of the active bin, a composition fee is deducted from the user's reserves. This composition fee is composed of the LP fees (which join the pool reserves) and the protocol fees (which don't, because they are claimable by the fee collector).

The protocol fees portion of the composition fees is handled within the `mint_utils::handle_composition_fees(...)` function, i.e. it is deducted from amounts provided to the bin and added to the `pool_protocol_fees` storage variable.

After minting is handled, the pool reserves are updated in the `mint_utils::update_reserves_after_mint(...)` function, but is done based on the difference between `amounts_received` and `amounts_to_refund`:

```
TypeScript
    let new_pool_reserves =
current_pool_reserves.add(amounts_received).sub(amounts_to_refund);
    storage_keys
        .pool_reserves
        .insert(pool_id, new_pool_reserves);
```

This is incorrect, because the difference contains the protocol fees, which are not supposed to be part of the pool reserves.

The `pool_reserves` are not used for any calculation within the contract, yet they are exposed by the `get_pool_reserves(...)` method, hence the impact is low, as it does not affect the protocol internal accounting or operation, yet it could case integration issues with other protocol, as the pool reserves reporting will be shown larger than in reality.

Recommendation

To fix this issue, increment the `pool_reserves` using the value of `amounts_added_to_bins` returned by the `mint_utils::mint_bins(...)` rather than the difference between `amounts_received` and `amounts_to_refund`.

IMM-LOW-03

Overinflation In `mint_bins()` Leads To Incorrect Storage Accounting. #9

Id	IMM-LOW-03
Severity	LOW
Category	Bug
Status	Fixed in 584e378537207d8120da39e746e1da0293da5528

Description

The `total_assets` is incremented twice on the call to mint liquidity to a pool, thereby leading to the wrong evaluation of the total assets handled by the pool curve state.

The `total_assets` variable is used to keep track of the total amount of assets created and managed by the pool curve. The issue stems from when `mint_bins()` is called, during which shares are calculated and the LP token is to be minted.

```
TypeScript
    let sub_id = next_lp_sub_id(storage_keys);
    let lp_asset_id = _mint(
        storage_keys
            .total_assets,
        storage_keys
            .total_supply,
        to,
        sub_id,
        1,
    );
```

The `sub_id` to be used for the new LP token is obtained via `next_lp_sub_id()`, where it increments the `total_assets`.

```
TypeScript
  /// Gets the next LP sub ID
#[storage(read, write)]
pub fn next_lp_sub_id(storage_keys: MintStorageKeys) -> SubId {
    let total_assets = storage_keys.total_assets.read();
    let next_asset_id = total_assets + 1;
    //@audit increment as it is written to after the addition of the previous state with `1`
    storage_keys.total_assets.write(next_asset_id);

    next_asset_id.as_u256().as_b256()
}
```

But when calling `_mint()` it also writes to the `totot_assets`, incrementing again if the LP token is being minted for the first time.

```
TypeScript
#[storage(read, write)]
pub fn _mint(
// .. snip ..
     // Only increment the number of assets minted by this contract if it hasn't been minted
before.
    if supply.is_none() {
        total_assets_key.write(_total_assets(total_assets_key) + 1);
    }
```

This causes the total assets minted in storage to be double the amount that has actually been minted for the curve state.

Recommendation

Remove the extra increment during the call to `_mint()`

IMM-INSIGHT-01

Inconsistent storage variables cleanup during LP burning #8

Id	IMM-INSIGHT-01
Severity	INSIGHT
Category	Informational
Status	Fixed in ff06ba98bcbb6d0f999354eb67293119687c4020

Description

The contract uses the following storage variables to hold the LP data:

- `lp_token_to_sub_id` - maps `AssetId` of the LP token to the `SubId` used to create it.
- `lp_shares_per_token` - maps the `AssetId` of the LP token with the `StorageVec` of `LPSharesPerBin` structs which represents shares owned by the LP token in each bin where liquidity was provided.
- `lp_token_to_pool_id` - maps the `AssetId` of the LP token with the `PoolId` - defines to which pool the given token pertains.

All the above are filled with relevant data during the LP minting process. While burning the following is being removed in the `mint_utils::burn_lp_token(...)`:

```
TypeScript
    assert(storage_keys.lp_shares_per_token.remove(lp_asset_id));
    assert(storage_keys.lp_token_to_sub_id.remove(lp_asset_id));
```

We can see that the `lp_token_to_pool_id` isn't removed, hence the inconsistency.

There is no impact here, as the specific `AssetId` cannot be re-used, yet there is inconsistent approach to data lifecycle here.

Recommendation

Two approaches can be considered here:

1. Removal of all data.
2. leaving the data as is, even after burning. This approach might be considered as it will save some gas.