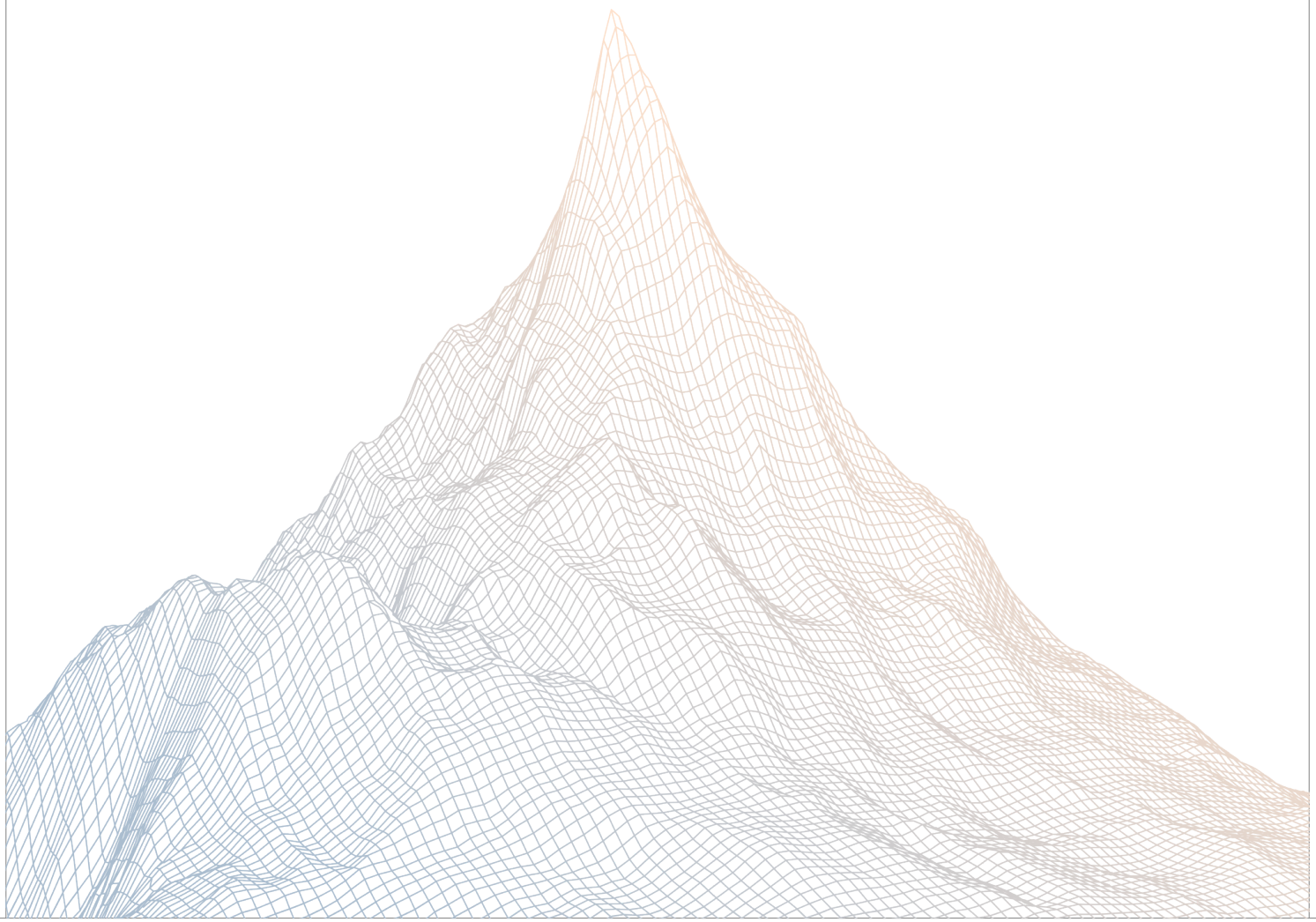
Zenith

Fluid

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: April 14th to May 11th, 2026

AUDITED BY: Matte
peakbolt

Contents

1 Introduction 2

1.1 About Zenith 3

1.2 Disclaimer 3

1.3 Risk Classification 3

2 Executive Summary 3

2.1 About Fluid 4

2.2 Scope 4

2.3 Audit Timeline 7

2.4 Issues Found 7

3 Findings Summary 7

4 Findings 9

4.1 High Risk 10

4.2 Medium Risk 15

4.3 Low Risk 21

4.4 Informational 42

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Fluid

Earn, Swap, and Borrow. All with maximum capital efficiency.

2.2 Scope

The engagement involved a review of the following targets:

Target	fluid-contracts-solana
Repository	https://github.com/Instadapp/fluid-contracts-solana/pull/138
Commit Hash	`c724a7d400a99a550ea7d93e0fb76f03f6a38e60`

Files	`Diff from 9a4fe285095db8f3d74d7d94f34eb2aec2bf5c34`
	`crates/library/src/math/dex.rs`
	`crates/library/src/math/mod.rs`
	`crates/library/src/math/scale.rs`
	`crates/library/src/math/u256.rs`
	`programs/dex/src/constants.rs`
	`programs/dex/src/errors.rs`
	`programs/dex/src/events.rs`
	`programs/dex/src/lib.rs`
	`programs/dex/src/invokes/liquidity_layer.rs`
	`programs/dex/src/invokes/mod.rs`
	`programs/dex/src/invokes/oracle.rs`
	`programs/dex/src/module/admin.rs`
	`programs/dex/src/module/mod.rs`
	`programs/dex/src/module/user.rs`
	`programs/dex/src/state/context.rs`
	`programs/dex/src/state/dex.rs`
	`programs/dex/src/state/mod.rs`
	`programs/dex/src/state/seeds.rs`
	`programs/dex/src/state/state.rs`
	`programs/dex/src/state/structs.rs`
	`programs/dex/src/utils/arbitrage.rs`
	`programs/dex/src/utils/borrow.rs`
	`programs/dex/src/utils/deposit.rs`
	`programs/dex/src/utils/helpers.rs`
	`programs/dex/src/utils/mod.rs`
	`programs/dex/src/utils/payback.rs`
	`programs/dex/src/utils/swap.rs`
	`programs/dex/src/utils/validate.rs`
	`programs/dex/src/utils/withdraw.rs`
	`programs/oracle/src/lib.rs`

Target	Mitigation Review
Repository	https://github.com/Instadapp/fluid-contracts-solana/pull/138
Commit Hash	`7068fb4c1e9e5deb72eadb211e03e62213d53369`

Files	`Diff from 9a4fe285095db8f3d74d7d94f34eb2aec2bf5c34`
	`crates/library/src/math/dex.rs`
	`crates/library/src/math/mod.rs`
	`crates/library/src/math/scale.rs`
	`crates/library/src/math/u256.rs`
	`programs/dex/src/constants.rs`
	`programs/dex/src/errors.rs`
	`programs/dex/src/events.rs`
	`programs/dex/src/lib.rs`
	`programs/dex/src/invokes/liquidity_layer.rs`
	`programs/dex/src/invokes/mod.rs`
	`programs/dex/src/invokes/oracle.rs`
	`programs/dex/src/module/admin.rs`
	`programs/dex/src/module/mod.rs`
	`programs/dex/src/module/user.rs`
	`programs/dex/src/state/context.rs`
	`programs/dex/src/state/dex.rs`
	`programs/dex/src/state/mod.rs`
	`programs/dex/src/state/seeds.rs`
	`programs/dex/src/state/state.rs`
	`programs/dex/src/state/structs.rs`
	`programs/dex/src/utils/arbitrage.rs`
	`programs/dex/src/utils/borrow.rs`
	`programs/dex/src/utils/deposit.rs`
	`programs/dex/src/utils/helpers.rs`
	`programs/dex/src/utils/mod.rs`
	`programs/dex/src/utils/payback.rs`
	`programs/dex/src/utils/swap.rs`
	`programs/dex/src/utils/validate.rs`
	`programs/dex/src/utils/withdraw.rs`
	`programs/oracle/src/lib.rs`

2.3 Audit Timeline

April 14, 2025	Audit start
May 11, 2025	Audit end
June 5, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	3
Low Risk	7
Informational	6
Total Issues	**18**

3

Findings Summary

ID	Description	Status
H-1	Single-smart-side liquidity ops never persist the new pool price	Resolved
H-2	swap_out enforces amount_in_max on the wrong quantity	Resolved
M-1	“Same block” detection is implemented with unix seconds instead of block/slot identity	Resolved
M-2	swap_out can deliver less than the user’s requested exact output	Resolved
M-3	swap_out rounds exact-out input down instead of up	Resolved
L-1	Unsafe U256Muldiv::add() call sites silently wrap overflow	Resolved
L-2	U256 div() correction path can panic on valid inputs via out-of-bounds high-word read	Resolved
L-3	Unsafe U256Muldiv::mul() call sites silently truncate overflow	Resolved
L-4	Invalid Chainlink Data Streams keeper lists are accepted at cache initialization	Resolved
L-5	Balanced withdraw and borrow legs round shares down	Resolved
L-6	swap_in can be globally frozen by admin-allowed fee and revenue settings	Acknowledged
L-7	Perfect share-based operations can become non-proportional after native rounding	Resolved
I-1	init_dex_admin authorization invariant break	Acknowledged
I-2	One-token payback returns a withdraw-specific error code	Resolved

ID	Description	Status
I-3	payback_perfect docs describe one-token max semantics	Resolved
I-4	swap_out quote/debug path returns the wrong result type for ADDRESS_DEAD	Resolved
I-5	Consider removing message logging from compute_deposit_shares()	Resolved
I-6	Users cannot use exact token amount operations at the current pool or debt ratio	Acknowledged

4

Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] Single-smart-side liquidity ops never persist the new pool price

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- user.rs#86

Description:

This affects `deposit`, `withdraw`, `withdraw_perfect_in_one_token`, `borrow`, `payback`, and `payback_perfect_in_one_token` when only one smart side is enabled and the operation can change the active pool price through an imbalanced liquidity change, but the handler only persists the new price when `arb.routing ≠ 0`.

When only one smart side is enabled, a non-perfect liquidity operation can move the live pool price by changing the active pool reserves in an imbalanced way. `run_arbitrage(...)` then computes the updated price from that changed state, but returns `routing = 0` because no cross-pool rebalance is possible.

The handlers only call `update_oracle_prices(...)` when `arb.routing ≠ 0`, so they skip the price update even though the live pool price may have changed.

As a result, the DEX can keep stale stored price history after a single-smart-side liquidity operation changes the active pool reserves and moves the live pool price.

Example Scenario:

Assume only smart collateral is enabled.

A user makes a liquidity change that is not perfectly balanced for the active pool, such as a one-sided deposit or one-sided withdrawal. That changes the collateral-side reserve ratio and moves the live pool price.

Afterward, `run_arbitrage(...)` computes an updated `new_price`, but returns `routing = 0`

because there is no cross-pool arbitrage to execute.

The handler then does:

```
if arb.routing != 0 {
    dex.update_oracle_prices(arb.new_price, arb.center_price)?;
}
```

Since `routing == 0`, the new price is not stored.

The live pool price has changed, but the DEX continues using stale stored price history.

Recommendations:

Persist `arb.new_price` whenever `run_arbitrage(...)` returns a valid updated price, even if `routing == 0`.

`routing` only indicates whether a balancing CPI was needed. It should not decide whether price history must be updated.

Fluid: Resolved with `@9e3c8a4f12...`

Zenith: Verified.

[H-2] `swap_out` enforces `amount_in_max` on the wrong quantity

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- swap.rs#429
- user.rs#L1008
-

Description:

`swap_out` computes a gross fee-inclusive input amount, but later uses only the revenue-cut-reduced amount as the user's required input.

In `execute_swap_out`, the route input is first computed with fees included:

```
amt_in_col = safe_multiply_divide_up(raw_in, SIX_DECIMALS, fee)?;
amt_in_debt = safe_multiply_divide_up(raw_in, SIX_DECIMALS, fee)?;
```

However, the function then applies `revenue_cut` and derives `total_in` from only the net booked amounts:

```
let col_deposit =
    amt_in_col.safe_mul(revenue_cut)?.safe_div(EIGHT_DECIMALS)?;
let debt_payback =
    amt_in_debt.safe_mul(revenue_cut)?.safe_div(EIGHT_DECIMALS)?;

let total_in = col_deposit_native.safe_add(debt_payback_native)?;
```

As a result, `amount_in_max` is checked against the reduced amount instead of the actual gross input required for the exact-out swap.

The same reduced amount is also used during settlement. `swap_out` calls `deposit_or_payback_liquidity` with `None` as the transfer override:

```
cpi_in.deposit_or_payback_liquidity(
    result.col_deposit,
    result.debt_payback,
```

```
    source,
    destination,
    ctx.accounts.signer.to_account_info(),
    signer_seeds,
    None,
)?;
```

When the override is `None`, the liquidity CPI transfers only `supply_amount + payback_amount`, so the user transfers only the net `col_deposit + debt_payback` amount rather than the gross input computed by the swap.

Example Scenario:

1. An exact-out swap computes a gross required input of `110`.
2. After applying `revenue_cut`, the net booked amount is `100`.
3. The current code checks `amount_in_max` against `100`.
4. It also transfers only `100` into Liquidity.
5. The user therefore pays less than the gross input required by the swap math.

Recommendations:

Use the gross exact-out input as `SwapResult.amount_in`.

Enforce `amount_in_max` against the gross input amount.

```
let total_in = col_deposit_native.safe_add(debt_payback_native)?;
// Exact-out user cost is the gross
   fee-inclusive input, before revenue cut.
let total_in = descale_up(amt_in_col.safe_add(amt_in_debt)?, in_decimals)?;

 if total_in > amount_in_max.cast()? {
     return Err(error!(ErrorCodes::DexExceedsAmountInMax));
 }
```

Then pass the gross amount as the liquidity transfer override:

```
cpi_in.deposit_or_payback_liquidity(
    result.col_deposit,
    result.debt_payback,
    source,
    destination,
```

```
    ctx.accounts.signer.to_account_info(),
    signer_seeds,
-   None, // Here we pass None as the ideal amount that we want to transfer
        to the LL is the col_deposit + debt_payback.
+   Some(result.amount_in),
    )?;
```

Keep `result.col_deposit` and `result.debt_payback` as the revenue-cut-reduced liquidity operation amounts.

Fluid: Resolved with `@69f7e11da5 ...`

Zenith: Verified.

4.2 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] “Same block” detection is implemented with unix seconds instead of block/slot identity

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- dex.rs#746

Description:

The price-guard path is responsible for validating external and internal price movement before swaps proceed.

However, the code decides whether it is in the “same block” path by checking `time_diff == 0`. On Solana, multiple blocks/slots can share the same unix second, so separate swaps in different blocks are misclassified as a same-block sequence.

Many distinct blocks can be incorrectly subjected to the strict same-second rules above, greatly amplifying the freeze window.

```
pub fn update_oracle_prices(&mut self, new_price: u128, center_price: u128)
    → Result<()> {
    let last_update = self.last_update_timestamp.cast()?;
    let current_timestamp: u128 = Clock::get()?.unix_timestamp.cast()?;
    let time_diff = current_timestamp.safe_sub(last_update)?;

    if time_diff == 0 {
        // doesn't matter if oracle is on or off when timediff = 0 code for
    both is same
```

Recommendations:

Track slot/block identity for same-block logic rather than unix-second timestamps.

Fluid: Resolved with `@74fbbcc167 ...`

Zenith: Verified.

[M-2] `swap_out` can deliver less than the user's requested exact output

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- swap.rs#425

Description:

`swap_out` is expected to guarantee that the native output actually transferred equals the caller's requested `amount_out`.

However, the modeled output is split into `amt_out_col` and `amt_out_debt`, each leg is independently floored to native units, and the function still returns the original `amount_out` as if the full exact output was delivered.

This lets the program complete an "exact-out" swap while transferring fewer native output units than promised.

Example Scenario:

1. A dual-route exact-out swap requests `1001` units of a 6-decimal token.
2. Routing splits the modeled output into `500.5` and `500.5` native units.
3. The code floors the legs to `500` and `500`, so only `1000` units are transferred.
4. The event still reports success for the original `1001` exact-out request.

```
let col_withdraw_native = amt_out_col.descale(out_decimals)?;
let debt_borrow_native = amt_out_debt.descale(out_decimals)?;
...
Ok(SwapResult {
    amount_out,
    col_withdraw: col_withdraw_native.cast()?,
    debt_borrow: debt_borrow_native.cast()?,
    ...
})
```

Recommendations:

Guarantee that the native output transferred matches the caller's requested `amount_out`, or revert the swap.

Fluid: Resolved with `@7068fb4c1e ...`

Zenith: Verified.

[M-3] `swap_out` rounds exact-out input down instead of up

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- swap.rs#424

Description:

`execute_swap_out` converts the modeled exact-out input into native units before enforcing payment.

However, it computes the fee-inclusive input with `safe_multiply_divide_up( ... )` and then floors the result with `descale( ... )` when deriving `col_deposit_native` and `debt_payback_native`.

This lets an exact-out swap charge less than the mathematically required native input.

Example Scenario:

1. The input token has 6 decimals.
2. A swap needs `0.001000001` input tokens after the fee-inclusive exact-out math.
3. Internally, the program represents amounts at 9 decimals, so this amount is `1,000,001` internal units.
4. `descale(6)` divides by `10^(9 - 6) = 1000` and floors, converting `1,000,001` to `1,000` native units.
5. The user is therefore charged `0.001000` tokens instead of the required `0.001001`.
6. The user still receives the requested output while underpaying by 1 native unit.

```
amt_in_col = safe_multiply_divide_up(raw_in, SIX_DECIMALS, fee)?;
...
let col_deposit_native = col_deposit.descale(in_decimals)?;
let debt_payback_native = debt_payback.descale(in_decimals)?;
let total_in = col_deposit_native.safe_add(debt_payback_native)?;
```

Recommendations:

Implement a `descale_up`-style conversion for `col_deposit_native` and `debt_payback_native` so the final exact-out input is rounded up instead of down when converting to native units.

This ensures the user pays at least the mathematically required amount for the requested exact-out swap.

Fluid: Ackknowledged and resolved with `@2e3161bc84 ...`

Zenith: Verified.

4.3 Low Risk

A total of 7 low risk findings were identified.

[L-1] Unsafe `U256Muldiv::add()` call sites silently wrap overflow

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- u256.rs#L263-L278
- dex.rs#L99
- dex.rs#L156
- dex.rs#L173
- dex.rs#L229
- dex.rs#L366
- dex.rs#L383
- dex.rs#L438
- dex.rs#L450
- dex.rs#L500
- dex.rs#L561-L562
- dex.rs#L575

Description:

`U256Muldiv::add()` adds two 256-bit values but returns only a `U256Muldiv`. The implementation keeps the low 256 bits and silently discards the final carry when the true sum exceeds `2^256 - 1`.

```
pub fn add(&self, other: U256Muldiv) -> Self {
    let mut result = U256Muldiv::new(0, 0);

    let mut carry = 0;
    for i in 0..NUM_WORDS {
        let x = self.get_word_u128(i);
        let y = other.get_word_u128(i);
        let t = x + y + carry;
```

```
            result.update_word(i, t.lo());

            carry = t.hi_u128();
        }

        result
    }
```

Several DEX math paths add already-wide U256 intermediates before a division, square-root, comparison, or subtraction:

```
let positive = mul_u256(t, xy_root).add(mul_u256(y, x2y2_root));
let p2_numer = gp_rx.add(ry_prec);
let under_sqrt = p2_sq.add(p3_u256);
let under_sqrt = p1_sq_u256.add(p2_u256);
let positive = cd.add(df).add(ei);
let under_sqrt = four_ce.add(b_sq);
let b_numer = de.add(cd2).add(cf);
let (c_over_a_u256, _) = cfg.add(cdg).div(U256Muldiv::new(0, d), false);
let b_numer = cf.add(de).add(dg);
let positive = cf.add(de);
let negative = fg.add(dg);
let under_sqrt = four_eg.add(b_sq);
```

If any of these sums exceed `2^256 - 1`, the subsequent math is performed on the wrapped low 256 bits. That can make later comparisons, square roots, divisions, and `try_into_u128()` checks operate on a smaller value than the true mathematical result.

The issue does not appear immediately reachable under normal DEX reserve magnitudes, but the unsafe arithmetic behavior is still present and should be guarded where `U256Muldiv::add()` is used on wide intermediates.

Recommendations:

Add a checked addition wrapper around the existing `add()` implementation. For unsigned addition, overflow is detectable by monotonicity: a non-overflowing sum must be greater than or equal to both operands. If `add()` wrapped modulo `2^256`, the sum becomes smaller than at least one operand.

```
impl U256Muldiv {
    pub fn checked_add(&self, other: U256Muldiv) -> Option<Self> {
        let sum = self.add(other);

        if sum.gte(*self) && sum.gte(other) {
            Some(sum)
```

```
        } else {
            None
        }
    }
}
```

Then use it at the DEX math call sites:

```
let under_sqrt = p2_sq
    .checked_add(p3_u256)
    .ok_or(error!(ErrorCodes::LibraryMathError))?;
```

Fluid: Resolved with @4038699e61 ...

Zenith: Verified.

[L-2] U256 `div()` correction path can panic on valid inputs via out-of-bounds high-word read

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- u256.rs#L739-L765

Description:

`U256Muldiv::div()` can panic on valid inputs inside `div_loop()` when the post-subtraction correction branch runs with `use_carry == true`.

In that case, `index + num_divisor_words == NUM_WORDS`, so the logical high word lives in `dividend_carry_space` rather than in `dividend.items`. The function handles that correctly earlier through `div_hi` and `d_head`, but the correction branch later unconditionally reads `dividend.get_word_u128(index + num_divisor_words)` when computing `new_carry`.

If `use_carry` is true, that becomes `get_word_u128(NUM_WORDS)`, which is out of bounds and panics on otherwise valid division inputs.

The buggy code is in `div_loop()`:

```
fn div_loop(
    index: usize,
    num_divisor_words: usize,
    mut dividend: U256Muldiv,
    dividend_carry_space: &mut u64,
    divisor: U256Muldiv,
    mut quotient: U256Muldiv,
) -> (U256Muldiv, U256Muldiv) {
    let use_carry = (index + num_divisor_words) == NUM_WORDS;

    // ...

    if k > d_head {
        qhat -= 1;
        k = 0;
        for i in 0..num_divisor_words {
            t = dividend
```

```
                .get_word_u128(index + i)
                .wrapping_add(divisor.get_word_u128(i))
                .wrapping_add(k);
            dividend.update_word(index + i, t.lo());
            k = t >> U64_RESOLUTION;
        }

        //@audit When use_carry is true, this reads
    dividend.items[NUM_WORDS] instead of dividend_carry_space and panics.
        let new_carry = dividend
            .get_word_u128(index + num_divisor_words)
            .wrapping_add(k)
            .lo();
        if use_carry {
            *dividend_carry_space = new_carry
        } else {
            dividend.update_word(
                index + num_divisor_words,
                dividend
                    .get_word_u128(index + num_divisor_words)
                    .wrapping_add(k)
                    .lo(),
            );
        }
    }

    quotient.update_word(index, qhat.lo());

    (quotient, dividend)
}
```

Proof of Concept:

The following unit test confirms the panic with the witness `dividend = 2^192` and `divisor = 2^128 + 1`:

```
#[test]
#[should_panic]
fn test_div_use_carry_correction_path_panics() {
    let dividend = U256Muldiv::new(1u128 << 64, 0); // 2^192
    let divisor = U256Muldiv::new(1, 1); // 2^128 + 1

    let _ = dividend.div(divisor, true);
}
```

Running this test panics with `index out of bounds: the len is 4 but the index is 4`, confirming the out-of-bounds read. `2^192` is the smallest dividend that occupies all four internal `u64` words, which is required to hit the `use_carry` path.

The trigger requires all of the following conditions:

- The division enters the multi-word long-division path.
- The normalized dividend needs `dividend_carry_space`.
- The current `div_loop()` step satisfies `index + num_divisor_words == NUM_WORDS`.
- The quotient-estimate correction branch runs with `k > d_head`.

These conditions require adversarial full-width U256 inputs and seem unlikely to be reachable from the DEX program.

Recommendations:

Use the same head-word source in the correction branch that the function already uses earlier in the subtraction path.

When `use_carry` is true, compute `new_carry` from `*dividend_carry_space`; otherwise compute it from `dividend.get_word_u128(index + num_divisor_words)`. Then write the result back through the matching destination.

```
diff --git a/crates/library/src/math/u256.rs
    b/crates/library/src/math/u256.rs
@@
        let new_carry = dividend
            .get_word_u128(index + num_divisor_words)
            .wrapping_add(k)
            .lo();
        let carry_head = if use_carry {
            *dividend_carry_space as u128
        } else {
            dividend.get_word_u128(index + num_divisor_words)
        };

        let new_carry = carry_head.wrapping_add(k).lo();

         if use_carry {
             *dividend_carry_space = new_carry
         } else {
            dividend.update_word(
                index + num_divisor_words,
```

```
            dividend
                .get_word_u128(index + num_divisor_words)
                .wrapping_add(k)
                .lo(),
        );
        dividend.update_word(index + num_divisor_words, new_carry);
    }
```

Add a regression test that exercises the `use_carry == true` correction path without panicking.

Fluid: Resolved with `@11f25ac97b ...`

Zenith: Verified.

[L-3] Unsafe `U256Muldiv::mul()` call sites silently truncate overflow

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- u256.rs#L297-L323
- u256.rs#L642-L646
- dex.rs#L158-L160
- dex.rs#L210-L212
- dex.rs#L227
- dex.rs#L366
- dex.rs#L420
- dex.rs#L432-L433
- dex.rs#L489-L490
- dex.rs#L553

Description:

`U256Muldiv::mul()` multiplies two 256-bit values but returns only a `U256Muldiv`. The implementation silently discards any product limbs above 256 bits instead of returning an error:

```
pub fn mul(&self, other: U256Muldiv) -> Self {
    let mut result = U256Muldiv::new(0, 0);

    let m = self.num_words();
    let n = other.num_words();

    for j in 0..n {
        let mut k = 0;
        for i in 0..m {
            let x = self.get_word_u128(i);
            let y = other.get_word_u128(j);
            if i + j < NUM_WORDS {
                let z = result.get_word_u128(i + j);
                let t = x.wrapping_mul(y).wrapping_add(z).wrapping_add(k);
                result.update_word(i + j, t.lo());
```

```
                k = t.hi_u128();
            }
        }

        // Don't update the carry word
        if j + m < NUM_WORDS {
            result.update_word(j + m, k as u64);
        }
    }

    result
}
```

This is safe for `mul_u256(a, b)` because a `u128 * u128` product always fits into 256 bits. It is not safe once the resulting `U256Muldiv` is multiplied again, because a `U256 * U256` product can require up to 512 bits. In those cases, the current implementation keeps the low 256 bits and silently drops the high bits.

Several DEX math paths call `U256Muldiv::mul()` on already-wide intermediates:

- `sqrt_mul3(a, b, c)` multiplies the exact `u128 * u128` result by a third value:

```
let ab = mul_u256(a, b);
let abc = ab.mul(U256Muldiv::new(0, c));
```

- `calculate_reserves_outside_range()` multiplies `rx * ry * price_precision`:

```
let rx_ry = mul_u256(rx, ry);
let rx_ry_prec = rx_ry.mul(U256Muldiv::new(0, price_precision));
```

- `calculate_debt_reserves()` multiplies `dx * dy * pb` and `ry * ry * price_precision`:

```
let dx_dy = mul_u256(dx, dy);
let dx_dy_pb = dx_dy.mul(U256Muldiv::new(0, pb));
let ry_sq_prec = mul_u256(ry, ry).mul(U256Muldiv::new(0, price_precision));
```

- `get_swap_and_deposit()` computes `4 * c * e` through `U256Muldiv::mul()`:

```
let four_ce = mul_u256(c, e).mul(U256Muldiv::new(0, 4));
```

- `get_withdraw_and_swap()` computes `2 * c * d`, `c * f * g`, and `c * d * g`:

```
let cd2 = mul_u256(c, d).mul(U256Muldiv::new(0, 2));
```

```
let cfg = mul_u256(c, f).mul(U256Muldiv::new(0, g));
let cdg = mul_u256(c, d).mul(U256Muldiv::new(0, g));
```

- `get_borrow_and_swap()` computes `c * f * g`:

```
let cfg = mul_u256(c, f).mul(U256Muldiv::new(0, g));
```

- `get_swap_and_payback()` computes `4 * e * g`:

```
let four_eg = mul_u256(e, g).mul(U256Muldiv::new(0, 4));
```

If any of these full products exceed `2^256 - 1`, the later division or square-root is performed on the truncated product rather than the true mathematical value. This makes the math helpers unsound across their full input domain.

The issue does not appear immediately reachable under normal DEX reserve magnitudes, but the unsafe arithmetic behavior is still present and should be guarded where `U256Muldiv::mul()` is used for triple products or scaled products.

Recommendations:

Add a checked multiplication wrapper around the existing `mul()` and verify the wrapped product by dividing it back. If the multiplication did not overflow, `product / self == other` exactly and the remainder is zero. If `mul()` silently truncated, the division check fails.

```
impl U256Muldiv {
    pub fn checked_mul(&self, other: U256Muldiv) -> Option<Self> {
        if self.is_zero() || other.is_zero() {
            return Some(U256Muldiv::new(0, 0));
        }

        let product = self.mul(other);

        // If no overflow occurred, product / self == other exactly.
        let (quotient, remainder) = product.div(*self, true);
        if quotient.eq(other) && remainder.is_zero() {
            Some(product)
        } else {
            None
        }
    }
}
```

Then replace unsafe `.mul( ... )` calls on `U256Muldiv` intermediates with checked calls, returning `LibraryMathError` on overflow:

```
let abc = ab
    .checked_mul(U256Muldiv::new(0, c))
    .ok_or(error!(ErrorCodes::LibraryMathError))?;
```

Apply the same pattern to the DEX math call sites listed above.

Fluid: Resolved with `@c5976c942f ...` and `@2baa77bd0e ...`

Zenith: Verified.

[L-4] Invalid Chainlink Data Streams keeper lists are accepted at cache initialization

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- programs/oracle/src/lib.rs#L183-L228
- programs/oracle/src/lib.rs#L233-L265
- programs/oracle/src/state/context.rs#L105-L115

Description:

`init_chainlink_data_streams_cache` accepts invalid keeper lists because it only checks that `keepers.len()` does not exceed `MAX_FEED_KEEPER_COUNT` before storing the list.

This allows the cache to be initialized with an empty keeper list, duplicate keepers, or `Pubkey::default()`. That is inconsistent with `update_chainlink_data_streams_cache_keepers`, which rejects the default pubkey and rejects updates that leave the cache with zero keepers.

```
pub fn init_chainlink_data_streams_cache(
    ctx: Context<InitChainlinkDataStreamsCache>,
    nonce: u16,
    feeds: Vec<FeedEntry>,
    keepers: Vec<Pubkey>,
) → Result<()> {
    if keepers.len() > MAX_FEED_KEEPER_COUNT {
        return
    err!(ErrorCodes::ChainlinkDataStreamsMaxFeedKeeperCountReached);
    }

    // ...

    chainlink_ds_cache.keepers = keepers;
}
```

The update path applies stricter validation:

```
pub fn update_chainlink_data_streams_cache_keepers(
    ctx: Context<UpdateCache>,
    keeper_status: Vec<AddressBool>,
) → Result<()> {
    for keeper in keeper_status.iter() {
        if keeper.addr == default_pubkey {
            return err!(ErrorCodes::InvalidParams);
        }
        keeper_map.insert(keeper.addr, keeper.value);
    }

    // ...

    if cache.keepers.len() == 0 {
        return err!(ErrorCodes::ChainlinkDataStreamsNoFeedKeepers);
    }
}
```

The keeper list is used by `chainlink_data_streams_feed_access_controller` to authorize suspend/resume actions. If a cache is initialized with no effective keeper, a later XStocks suspension cannot be resumed by the normal keeper controller until an admin updates the keeper list.

Additionally, because `Pubkey::default()` is accepted at init but rejected by the update instruction before applying `value: false`, a default keeper entry cannot be removed through the normal update path and can permanently consume one of the limited keeper slots.

Recommendations:

Validate keepers during initialization using the same or stricter rules as the update path.

Reject empty keeper lists, reject `Pubkey::default()`, and reject duplicates before storing `chainlink_ds_cache.keepers`.

```
    if keepers.is_empty() {
        return err!(ErrorCodes::ChainlinkDataStreamsNoFeedKeepers);
    }

     if keepers.len() > MAX_FEED_KEEPER_COUNT {
         return
err!(ErrorCodes::ChainlinkDataStreamsMaxFeedKeeperCountReached);
     }
```

```
let default_pubkey = Pubkey::default();
let mut keeper_set = HashSet::new();
for keeper in keepers.iter() {
    if *keeper == default_pubkey || !keeper_set.insert(*keeper) {
        return err!(ErrorCodes::InvalidParams);
    }
}
```

Fluid: Resolved with @b56e45bad5 ...

Zenith: Verified.

[L-5] Balanced withdraw and borrow legs round shares down

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- programs/dex/src/utils/withdraw.rs#L44-L66
- programs/dex/src/utils/borrow.rs#L45-L67

Description:

`compute_withdraw_shares` and `compute_borrow_shares` calculate the balanced proportional share component for exact-amount user withdrawals and borrows.

However, both paths use floor division for the balanced component even though exact withdrawals should burn at least the proportional supply shares and exact borrows should mint at least the proportional debt shares.

This can under-burn withdrawal shares or under-mint borrow shares by rounding dust before the one-sided excess component is added.

```
fn compute_withdraw_shares(...) -> Result<WithdrawResult> {
    let shares_from_t0 = safe_multiply_divide(t0_adj, SHARE_RATIO_PRECISION,
    token0_reserves)?;
    let shares_from_t1 = safe_multiply_divide(t1_adj, SHARE_RATIO_PRECISION,
    token1_reserves)?;

    shares = safe_multiply_divide(shares_from_t1, total_supply_shares,
    SHARE_RATIO_PRECISION)?;
}
```

```
fn compute_borrow_shares(...) -> Result<BorrowResult> {
    let shares_from_t0 = safe_multiply_divide(t0_adj, SHARE_RATIO_PRECISION,
    token0_debt)?;
    let shares_from_t1 = safe_multiply_divide(t1_adj, SHARE_RATIO_PRECISION,
    token1_debt)?;

    shares = safe_multiply_divide(shares_from_t1, total_borrow_shares,
    SHARE_RATIO_PRECISION)?;
```

```
}
```

Recommendations:

Round the balanced proportional component up for exact withdrawals and exact borrows.

```
fn compute_withdraw_shares(...) → Result<WithdrawResult> {
    shares = safe_multiply_divide_up(shares_from_t1, total_supply_shares,
    SHARE_RATIO_PRECISION)?;
}
```

```
fn compute_borrow_shares(...) → Result<BorrowResult> {
    shares = safe_multiply_divide_up(shares_from_t1, total_borrow_shares,
    SHARE_RATIO_PRECISION)?;
}
```

Fluid: Resolved with `@0d3bb6160f...`

Zenith: Verified.

[L-6] `swap_in` can be globally frozen by admin-allowed fee and revenue settings

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- user.rs#891

Description:

`swap_in` should remain operable for every admin-valid fee and revenue configuration.

However, the handler transfers the full user input to LL while the booked `col_deposit + debt_payback` amount is reduced by `revenue_cut`, and LL only tolerates at most 1% excess vault inflow over the booked amount.

This means fully admin-valid settings can create a transfer-book mismatch large enough to make every exact-in swap revert at the LL bound check.

Example Scenario:

1. Admin sets the allowed maximum fee of `10%` and a `100%` revenue cut.
2. A user swaps `100` units exact-in.
3. The handler transfers all `100` units into the vault but only books `90` units after fee/revenue slicing.
4. LL sees `11.1%` excess input relative to `90` and rejects the swap because `MAX_INPUT_AMOUNT_EXCESS` is only `1%`.

```
cpi_in.deposit_or_payback_liquidity(..., Some(amount_in))?;
...
let col_deposit =
    amt_in_col.safe_mul(revenue_cut)?.safe_div(EIGHT_DECIMALS)?;
let debt_payback =
    amt_in_debt.safe_mul(revenue_cut)?.safe_div(EIGHT_DECIMALS)?;
...
if net_amount_in < operate_amount_in || net_amount_in > operate_amount_in *
    101 / 100 {
```

```
    return Err(ErrorCodes::TransferAmountOutOfBounds.into());
}
```

Recommendations:

Align the transferred input with the amount actually booked into LL, or cap DEX fee and revenue settings so they can never violate LL's excess-input bound.

Exact-in revenue should follow an explicit accounting path rather than relying on LL to absorb a larger vault inflow than it books.

Fluid: Acknowledged. This is a known consideration when integrating downstream protocols that charge fees. The same pattern exists in the flashloan program, where we already cap the maximum fees.We'll apply a similar approach here and avoid setting excessively high fees (e.g. 10%). In practice, the fee is typically around 0.05% with a 25% revenue share, which should not create this issue, especially since we only intend to launch this for pegged pairs.

[L-7] Perfect share-based operations can become non-proportional after native rounding

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- withdraw.rs#L355-L376
- user.rs#L337-L347
- borrow.rs#L349-L375
- user.rs#L589-L599

Description:

The perfect share-based liquidity paths compute proportional token amounts in internal precision, round each leg independently into native units, and then continue using the perfect path even when rounding destroys proportionality.

This affects both:

- `withdraw_perfect()`, where supply shares are burned from a proportional share request but the realized native outputs can become `0/0`, `0/x`, or `x/0`
- `borrow_perfect()`, where debt shares are minted from a proportional share request but the realized native outputs can become `0/0`, `0/x`, or `x/0`

The most severe case is `0/0`, where share accounting changes even though no liquidity movement occurs at all. But the issue is broader than that: even when only one leg rounds to zero, the instruction is no longer truly proportional, yet it still executes through the perfect path instead of reverting or falling back to the ordinary imbalanced path.

For `withdraw_perfect()`:

```
let token0_amt: u64 = t0_adj
    .safe_sub(1)?
    .descale(dex.token_0_decimals)?
    .safe_sub(1)?
    .cast()?;
let token1_amt: u64 = t1_adj
    .safe_sub(1)?
    .descale(dex.token_1_decimals)?
```

```
    .safe_sub(1)?
    .cast()?;

let new_total = total_supply_shares.safe_sub(shares)?;
dex.update_supply_shares(new_total.cast()?)?;
position.supply_or_withdraw(-(shares.cast::<i128>()?))?;
```

The wrapper then conditionally executes each withdrawal leg:

```
if result.token0_amt > 0 {
    let cpi = ctx.accounts.get_token0_cpi_accounts()?;
    cpi.withdraw_or_borrow_liquidity(result.token0_amt, 0, user, user,
    signer_seeds)?;
}

if result.token1_amt > 0 {
    let cpi = ctx.accounts.get_token1_cpi_accounts()?;
    cpi.withdraw_or_borrow_liquidity(result.token1_amt, 0, user, user,
    signer_seeds)?;
}
```

For `borrow_perfect()`:

```
let token0_amt: u64 = t0_adj
    .safe_sub(1)?
    .descale(dex.token_0_decimals)?
    .safe_sub(1)?
    .cast()?;

let token1_amt: u64 = t1_adj
    .safe_sub(1)?
    .descale(dex.token_1_decimals)?
    .safe_sub(1)?
    .cast()?;

let new_total = total_borrow_shares.safe_add(shares)?;
dex.update_borrow_shares(new_total.cast()?)?;
position.borrow_or_payback(shares.cast()?)?;
```

And the wrapper likewise conditionally executes each borrow leg:

```
if result.token0_amt > 0 {
    let cpi = ctx.accounts.get_token0_cpi_accounts()?;
    cpi.withdraw_or_borrow_liquidity(0, result.token0_amt, user, user,
```

```
    signer_seeds)?;
}

if result.token1_amt > 0 {
    let cpi = ctx.accounts.get_token1_cpi_accounts()?;
    cpi.withdraw_or_borrow_liquidity(0, result.token1_amt, user, user,
    signer_seeds)?;
}
```

As a result, these instructions can:

- mutate share accounting while transferring or borrowing no tokens at all when rounding produces `0/0`
- execute a realized one-sided `0/x` or `x/0` operation even though the request was priced as a proportional perfect-shares action

Recommendations:

Treat zero-valued rounded outputs as invalid for these perfect share-based paths and revert the instruction instead of silently skipping the liquidity call. More generally, if native rounding destroys proportionality, either revert or reroute through the ordinary imbalanced path so the realized operation is priced and validated according to what actually executes.

Fluid: Acknowledged and resolved with `@fc533d70d6 ...`, but we think the likelihood is really low and it only happens in edge cases (tiny share amounts / reserve splits where internal proportional math survives but aggressive per-leg floor rounding wipes a native leg or both legs before CPI). We now revert with DEX_PERFECT_NATIVE_AMOUNTS_ROUND_TO_ZERO whenever any leg has positive internal proportion but rounds to zero native, so we never burn or mint perfect-path shares without matching two-sided liquidity.

Zenith: Verified.

4.4 Informational

A total of 6 informational findings were identified.

[I-1] `init_dex_admin` authorization invariant break

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- context.rs
- admin.rs
- constants.rs

Description

`DexAdmin` should enter existence only under a trusted bootstrap identity, but `init_dex_admin` lets any signer create the singleton PDA and install any nonzero `authority`. I traced `InitDexAdmin` in `state/context.rs` to `init_dex_admin` in `module/admin.rs`; the account constraint fixes only the PDA seeds and payer, while the handler never binds `signer` to `authority` or to `PROTOCOL_INIT_AUTH`. The constant already exists.

```
if liquidity ≠ LIQUIDITY_PROGRAM_ID {
    return Err(error!(ErrorCodes::DexAdminLiquidityProgramMismatch));
}
if authority == Pubkey::default() {
    return Err(error!(ErrorCodes::DexAdminAddressZeroNotAllowed));
}
dex_admin.authority = authority;
dex_admin.auths.push(authority);
```

`UpdateDexAdminAuthority` and `UpdateDexAdminAuths` later require the stored `dex_admin.authority`, so the first writer becomes the only party that can rotate or repopulate the admin set. Put numerically, 1 successful initializer > 0 later honest initializers because `init` runs once and that same transaction assigns both control fields: `authority` and `auths`. I did not find code support for the broader stale-oracle accumulation claim; the verified consequence is persistent takeover of every DEX admin-gated path.

Recommendation

Bind `InitDexAdmin.signer` to `PROTOCOL_INIT_AUTH` or another explicit bootstrap governance key, and reject any `authority` value that falls outside that model. Making deployment and admin initialization atomic reduces the race window, but the missing state-transition check on the singleton initializer is the fix that restores the intended authorization invariant.

Fluid: Acknowledged. This is a known pattern across the Solana ecosystem and should be considered informational. We always verify the post-init state to ensure the correct admin is configured, and have not encountered issues with this approach in any previous deployments.

[I-2] One-token payback returns a withdraw-specific error code

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- programs/dex/src/utils/payback.rs#L321-L322
- programs/dex/src/errors.rs#L135
- programs/dex/src/errors.rs#L207

Description:

`payback_perfect_in_one_token` returns a withdraw-specific error code for invalid payback token selection.

When both `max_token0` and `max_token1` are set, or when neither is set, the function returns `DexInvalidWithdrawAmts`. This path is a debt payback operation, and the codebase already has a payback-specific `DexInvalidPaybackAmounts` error.

Use `DexInvalidPaybackAmounts`, or add a more specific one-token payback error if distinct handling is desired.

```
pub fn execute_payback_perfect_in_one_token(
    dex: &mut Dex,
    position: &mut DexPosition,
    params: ExecutePaybackPerfectInOneTokenParams,
    borrow_amounts: BorrowAmounts,
    exchange_prices: ExchangePrices,
    external_center_price: u128,
    current_timestamp: u128,
) -> Result<PaybackPerfectInOneTokenResult> {
    // ...

    if (max_token0 > 0 && max_token1 > 0) || (max_token0 == 0 && max_token1
    == 0) {
        return Err(error!(ErrorCodes::DexInvalidWithdrawAmts));
        return Err(error!(ErrorCodes::DexInvalidPaybackAmounts));
    }
}
```

Fluid: Resolved with `@2f83d1b629 ...`

Zenith: Verified.

[I-3] `payback_perfect` docs describe one-token max semantics

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- programs/dex/src/module/user.rs#L648-L659
- programs/dex/src/utils/payback.rs#L221-L240

Description:

The `payback_perfect` documentation incorrectly describes `max_token0` and `max_token1` as if setting one side to zero selects a one-token repayment.

Update the `payback_perfect` documentation so it does not describe one-token repayment semantics.

```
/// Repays debt for a fixed amount of borrow shares, splitting payment
   across both tokens.
///
/// # Arguments
///
/// * `shares` — borrow shares to burn
/// * `max_token0` — maximum token0 to pay (use `0` if repaying only in
   token1)
/// * `max_token1` — maximum token1 to pay (use `0` if repaying only in
   token0)
/// * `max_token0` — maximum token0 to pay in the proportional two-token
   payback
/// * `max_token1` — maximum token1 to pay in the proportional two-token
   payback
///
/// # Returns
///
/// `(token0_amt, token1_amt)` — amounts of token0 and token1 paid.
pub fn payback_perfect<'info>(
```

Fluid: Resolved with @63d037d9c6 ...

Zenith: Verified.

[I-4] `swap_out` quote/debug path returns the wrong result type for `ADDRESS_DEAD`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- user.rs#L1086-L1093

Description:

The `ADDRESS_DEAD` quote/debug branch in `swap_out()` uses the wrong error code.

This branch is part of the exact-out quote path, but it returns `DexSwapInResult` instead of a dedicated exact-out result code. That makes the quote/debug interface inconsistent and can cause off-chain consumers to classify the response as a `swap_in` result even though it came from `swap_out()`.

```
if recipient == ADDRESS_DEAD {
    msg!(
        "DexSwapInResult: [{}, {}, {}]",
        swap0to1,
        result.amount_in,
        amount_out
    );
    return Err(error!(ErrorCodes::DexSwapInResult));
}
```

Recommendations:

Use a dedicated exact-out result code and matching log label in the `swap_out()` `ADDRESS_DEAD` branch.

Fluid: Resolved with `@6652b8b5ad ...`

Zenith: Verified.

[I-5] Consider removing message logging from `compute_deposit_shares()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- deposit.rs#23

Description:

`compute_deposit_shares()` still contains several unconditional `msg!` calls that appear to be leftover debugging output.

This helper runs on the user deposit path and logs normalized deposit inputs, reserve values, intermediate share ratios, and the provisional share result on every call.

These logs are not needed for correctness and add avoidable runtime overhead and noisy transaction output on a hot path.

The current messages are:

```
msg!("t0_adj: {}", t0_adj);
msg!("t1_adj: {}", t1_adj);
msg!("c.token_0_real_reserves: {}", c.token_0_real_reserves);
msg!("c.token_1_real_reserves: {}", c.token_1_real_reserves);
msg!("total_supply_shares: {}", total_supply_shares);
msg!("fee: {}", fee);
msg!("shares_from_t0: {}", shares_from_t0);
msg!("shares_from_t1: {}", shares_from_t1);
msg!("shares: {}", shares);
```

Recommendations:

Remove these `msg!` calls, or gate them behind an explicit debug-only mechanism if they are still useful during development.

Fluid: Resolved with `@7a736261fd ...`

Zenith: Verified.

[I-6] Users cannot use exact token amount operations at the current pool or debt ratio

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- deposit.rs#44
- withdraw.rs#45
- borrow.rs#50
- payback.rs#39

Description:

The exact token amount operations `deposit()`, `withdraw()`, `borrow()` and `payback()` reject the exact proportional case instead of treating it as the simplest zero-excess path.

Across collateral and debt flows, the exact token amount operations first compute the share ratio implied by each token leg. If one side implies a larger share ratio, the helpers process the balanced portion first and then route the excess leg through the swap-style path. However, when both token legs imply the same share ratio, the helpers return an error instead of accepting the operation as a pure proportional share change.

As a result, users and integrations cannot use these operations when the provided token amounts already match the current collateral ratio or current debt ratio, even though those are the cleanest no-swap cases.

Example Scenario:

Assume the collateral pool currently holds:

- `token0_reserves = 100`
- `token1_reserves = 200`

A user calls `deposit( ... )` with:

- `token0_amt = 10`
- `token1_amt = 20`

This matches the current pool ratio exactly:

- `10 / 100 = 10%`
- `20 / 200 = 10%`

So both token legs represent the same proportional deposit. There is no excess token on either side, no swap is needed, and the operation should succeed as a pure proportional mint.

Instead, because both sides imply the same share fraction, the helper reaches the equality branch and reverts with `DexInvalidDepositAmounts`.

The same pattern applies to `withdraw( ... )`, `borrow( ... )`, and `payback( ... )`: if both token legs match the current collateral ratio or debt ratio exactly, the call should succeed as a pure proportional share change, but instead reverts on the equality branch.

```
if t0_adj > 0 && t1_adj > 0 {
    let shares_from_t0 =
        safe_multiply_divide(t0_adj, SHARE_RATIO_PRECISION,
    c.token_0_real_reserves)?;
    let shares_from_t1 =
        safe_multiply_divide(t1_adj, SHARE_RATIO_PRECISION,
    c.token_1_real_reserves)?;

    let (excess_t0, excess_t1) = if shares_from_t0 > shares_from_t1 {
        ...
    } else if shares_from_t1 > shares_from_t0 {
        ...
    } else {
        return Err(error!(ErrorCodes::DexInvalidDepositAmounts));
    };
}
```

```
if t0_adj > 0 && t1_adj > 0 {
    let shares_from_t0 = safe_multiply_divide(t0_adj, SHARE_RATIO_PRECISION,
    token0_reserves)?;
    let shares_from_t1 = safe_multiply_divide(t1_adj, SHARE_RATIO_PRECISION,
    token1_reserves)?;

    if shares_from_t0 > shares_from_t1 {
        ...
    } else if shares_from_t1 > shares_from_t0 {
        ...
    } else {
        return Err(error!(ErrorCodes::DexInvalidWithdrawAmounts));
    }
}
```

```
if t0_adj > 0 && t1_adj > 0 {
    let shares_from_t0 = safe_multiply_divide(t0_adj, SHARE_RATIO_PRECISION,
    token0_debt)?;
    let shares_from_t1 = safe_multiply_divide(t1_adj, SHARE_RATIO_PRECISION,
    token1_debt)?;

    if shares_from_t0 > shares_from_t1 {
        ...
    } else if shares_from_t1 > shares_from_t0 {
        ...
    } else {
        return Err(error!(ErrorCodes::DexInvalidBorrowAmounts));
    }
}
```

```
if t0_adj > 0 && t1_adj > 0 {
    let shares_from_t0 = safe_multiply_divide(t0_adj, SHARE_RATIO_PRECISION,
    d.token_0_debt)?;
    let shares_from_t1 = safe_multiply_divide(t1_adj, SHARE_RATIO_PRECISION,
    d.token_1_debt)?;

    if shares_from_t0 > shares_from_t1 {
        ...
    } else if shares_from_t1 > shares_from_t0 {
        ...
    } else {
        return Err(error!(ErrorCodes::DexInvalidPaybackAmounts));
    }
}
```

Recommendations:

Treat the equality case as a successful pure proportional path across all four helpers:

- mint or burn the corresponding balanced share amount directly
- set both excess legs to zero
- skip the swap-derived share logic

This would allow exact token amount operations to succeed when the provided amounts already match the current pool ratio or debt ratio.

Fluid: Acknowledged. This was an intentional design choice to maintain consistency with our EVM DEX codebase, where the two operations are handled through separate methods. We consider this an informational finding.